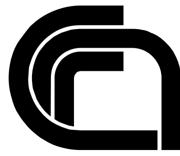
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PUblication MAnagement



Consiglio Nazionale delle Ricerche

Implementation of an IP table lookup algorithm based on an Adaptive Stratified Tree

M. Pellegrini, G. Fusco

IIT TR-17/2002

Technical report

Agosto 2002

Istituto di Informatica e Telematica

Implementation of an IP table lookup algorithm based on an Adaptive Stratified Tree

Giordano Fusco and Marco Pellegrini
Institute for Informatics and Telematics of CNR*

IIT-???

July 2002

Abstract

IP Table Lookup is one of the bottlenecks operation of a router. M. Pellegrini, G. Fusco and G. Vecchiocattivi in *Adaptive Stratified Search Trees for IP Table Lookup* (manuscript) presented a new solution for this problem based on a data structure called *Adaptive Stratified Tree* (AST). The algorithm has been implemented and compared with the state of the art software solution, notably the *LC-Trie* of S. Nilsson and G. Karlson. Here we present some experimental results and the complete C code of our program.

*Area della Ricerca. Via Moruzzi 1, 56124 Pisa ITALY. giordano.fusco, marco.pellegrini@iit.cnr.it

Contents

1 Introduction **1**

2 Esperimental results **2**
2.1 Empirical comparison 2
2.2 Test hardware 2
2.3 Routing tables used in the tests 2
2.4 Three kind of traffic 2
2.5 Parameters 3
2.6 Dimension of the data sets 3
2.7 Time comparison 3
2.8 Maximum depth of the primary search structure 3
2.9 Memory consumption 4
2.10 Memory consumption for IPv6 4

3 About the code **5**
3.1 Little-endian 5
3.2 Copyright 5
3.3 No Warranty 5

4 Basic definitions **6**
4.1 Boolean type 7
4.2 Words of memory 7
4.3 IP addresses 8
4.4 Miscellaneous macros 9

5 Routing table based on the AST **11**
5.1 Next-hop table 11
5.2 Base vector 14
5.3 Anchor table 16
5.4 Nodes of AST 19
5.4.1 Kind of nodes 20
5.4.2 Internal nodes 22
5.4.3 Full leaves 25
5.4.4 Special leaves 27
5.4.5 Covered empty leaves 29
5.4.6 Uncovered empty leaves 31
5.5 Routing table based on the AST 32

6 Entry vector **34**
6.1 Entry vector: header 34
6.2 Entry vector: implementation 36

7 Prefixes **41**
7.1 Segment associated to a prefix 42

7.2 Macros to operate on prefixes 43

8 Macros to explore buckets **45**
8.1 Compute the step 46
8.2 Explore buckets from anchor to extremes 47
8.3 Explore buckets from left to right 49

9 Priority queue **52**
9.1 Priority queue: header 52
9.2 Priority queue: implementation 55

10 Parameters **63**

11 Building the AST **65**
11.1 Building the AST: header 66
11.2 Building the AST: implementation 67

12 Building the routing table **80**
12.1 Building the routing table: header 81
12.2 Building the routing table: implementation 82

13 Searching the next-hop of an address **92**
13.1 Searching the next-hop of an address: header 93
13.2 Searching the next-hop of an address: implementation 98

14 CPU time measurement **105**
14.1 CPU time measurement: header 105
14.2 CPU time measurement: implementation 106

15 Statistics **108**
15.1 Statistics: header 108
15.2 Statistics: implementation 110

16 Testing the routing table based on the AST **121**

17 Check correctness **132**
17.1 Check correctness: header 132
17.2 Check correctness: implementation 134

18 Extract routing prefixes from routing tables of IPMA **142**

19 Randomly generate a traffic file **147**

20 References **151**

21 Index **152**

1 Introduction

In a large network of computers, such as Internet, there are special machines called *routers*. A router is a device that forwards packets between sub-networks. The forwarding decision is based on information about network layer and routing tables. One of the bottlenecks of a router is the *IP table lookup* operation which consists in finding the entry in the routing table matching the address of an incoming packet and decide the appropriate exit route. Between the software solutions the one described by Nilsson and Karlsson in *IP-Address Lookup Using LC-Tries* (in *IEEE Journal on Selected Areas in Communications*, volume 17, number 6, pages 1083–1092, June 1999) is one of the most popular. Pellegrini and Vecchiocattivi in *How Deep is your Search Tree?* (Technical Report IMC B4-01-12) and *Empirical study of search trees for IP address lookup* (Technical Report IMC B4-01-13) proposed a new solution to the IP table lookup problem based on a geometric interpretation. This idea was improved and expained in *Adaptive Stratified Search Trees for IP Table Lookup* (manuscript). The entries of a routing table can be mapped to segments on a real line and the incoming IP address can be mapped to points. Thus the problem can be rephrased in finding the shortest segment that contains a point. The basic instrument used to quickly perform the match between segments and points is the adaptive stratified tree (AST). Here an efficient implementation of the IP table lookup based on an AST and some experimental results are given. Comparison experiments demonstrate that this implementation is competitive with the one of Nilsson and Karlsson.

2 Esperimental results

2.1 Empirical comparison

Stefan Nilsson provided an efficient implementation of an IP address lookup algorithm based on an LC-Trie (his code is freely available on the web at `http://www.nada.kth.se/~snilsson/public/code/router`). Given a routing table and a traffic, Nilsson's program allows to measure the average search time.

We implemented a routing table based on an AST allowing to perform the same search test of Nilsson. We run both programs on the same machine, with the same routing tables and the same traffics in order to perform comparisons about time and memory consumption.

2.2 Test hardware

The machine on which we ran the experiments has the following characteristics:

Model name: AMD Athlon(TM) XP 1600+

CPU MHz: 1400

Cache size: 256 KB

Total memory: 970816 KB

2.3 Routing tables used in the tests

The tests were run on real routing tables downloaded from the Internet Performance Measurement and Analysis (IPMA) project site `http://www.merit.edu/ipma/routing_table`.

2.4 Three kind of traffic

The search time highly depend on the traffic. We tested the two programs on the three following kind of traffics.

Same Probability Hypothesis. This traffic is based on the hypothesis that every prefix has the same probability of being accessed. In other words the traffic per prefix is supposed to be the same for all prefixes. Thus the entries of the routing table are extended to 32 bits by adding zeroes and are permuted to reduce the effects of cache locality.

Uniform Random. The IP address of this traffic are uniform random in all the space.

Mixed. The IP address of this traffic are taken:

- 90% uniform random in the space obtained by concatenating all the segments associated to prefixes,
- 10% uniform random in all the space.

2.5 Parameters

The AST program accepts a certain number of parameters. We ran it on each routing table using the *mixed* traffic and variating the parameters over a large set of choices. Then for each routing table we chose the configuration that minimizes the time and the one that minimizes the memory consumption. We used the same sets of parameters for the other traffics.

2.6 Dimension of the data sets

Each prefix has been mapped into an interval, and the endpoints of the intervals form the data sets. The following table show the dimension of the data sets.

	Paix	MaeEast	AADS	PB	MaeWest
n. of prefixes	17736	18240	31992	44652	74016
n. of points	32271	32838	56297	77206	123509

2.7 Time comparison

The following times are measured in Mega-Lookups/second.

Same Prob. Hypothesis	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	4.46	4.40	3.13	2.74	2.33
AST: min Time	7.50	8.10	4.65	3.74	2.99
AST: min Memory	8.34	8.56	4.72	3.96	3.21

Uniform Random	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	7.89	7.84	6.11	5.26	4.19
AST: min Time	15.5	14.9	11.1	8.91	6.66
AST: min Memory	12.2	12.5	10.4	8.25	5.88

Mixed	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	10.2	9.98	7.68	6.14	4.42
AST: min Time	12.7	12.5	10.4	8.51	6.39
AST: min Memory	10.9	11.1	9.62	8.17	5.91

2.8 Maximum depth of the primary search structure

The following table shows the maximum depth of the primary search structure (trie and AST).

Number of Levels	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	5	6	6	6	6
AST: min Time	3	3	3	3	3
AST: min Memory	5	4	4	4	5

2.9 Memory consumption

We distinguish the storage used essentially to hold the input and the storage used to support searching. The search structures for LC-Trie are trie and prefix vector (see [NK 99]). Search structures for AST are AST and anchor table.

Search Structures	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	347804	349600	434200	515164	711036
AST: min Time	73292	48324	68096	119952	307304
AST: min Memory	22464	23864	43976	61348	88048

Base Next-hop vectors	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	277164	284668	496732	689252	1125256
AST	258280	262816	450520	617656	988244

Total Memory	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	624968	634268	930932	1204416	1836292
AST: min Time	331572	311140	518616	737608	1295548
AST: min Memory	280744	286680	494496	679004	1076292

2.10 Memory consumption for IPv6

When switching to IPv6 the structures containing the input will grow up. The following tables shows an estimate of the memory consumptions.

Search Structures	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	347804	349600	434200	515164	711036
AST: min Time	75596	50364	73268	126456	317192
AST: min Memory	27288	27512	50576	70336	104344

Base Next-hop vectors	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	485280	498412	869596	1206200	1969576
AST	645868	657208	1126516	1544152	2470868

Total Memory	Paix	MaeEast	AADS	PB	MaeWest
LC-Trie	833084	848012	1303796	1721364	2680612
AST: min Time	721464	707572	1199784	1670608	2788060
AST: min Memory	673156	684720	1177092	1614488	2575212

3 About the code

3.1 Little-endian

The heart of this code is based on bit manipulation. Writing the macros, in which the bit manipulation is performed, we assumed a *little-endian* architecture. If you wish to compile this code on a big-endian architecture, please check all the macros.

3.2 Copyright

Copyright (C) 2002, Giordano Fusco.

3.3 No Warranty

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

4 Basic definitions

[basic_defs_h.web]

This section contains some basic definitions needed in all the program.

"tr_body.c" 4 ≡
@O basic_defs.h

[basic_defs_h.web]

Guard

"basic_defs.h" 4.0.1 ≡

```
#ifndef _AST_BASIC_DEFS_H_
#define _AST_BASIC_DEFS_H_
```

[basic_defs_h.web]

Includes

"basic_defs.h" 4.0.2 ≡

```
#include <stdint.h>    // needed for uint32_t
```

[basic_defs_h.web]

4.1 Boolean type

[basic_defs_h.web]

Type: **BOOL**
Description: Boolean type.

"basic_defs.h" 4.1 ≡

```
#ifndef BOOL
  typedef enum
    {
    FALSE = 0, TRUE = 1
    } BOOL;
#endif
```

[basic_defs_h.web]

Constants: *TRUE* and *FALSE*
Description: This makes the names *TRUE* and *FALSE* also available as macros.

"basic_defs.h" 4.1.1 ≡

```
#define TRUE TRUE
#define FALSE FALSE
```

[basic_defs_h.web]

4.2 Words of memory

[basic_defs_h.web]

In *C* the size of an **int** is machine dependent. This code is based on 32-bit unsigned words. The following type definition is necessary to be machine independent. In this way, each time we need a 32-bit word, we use the type **WORD** instead of **int**.

Type: **WORD**
Description: Word of memory.
Important: Make sure that, on your machine, **WORD** is effectively a 32-bit unsigned int.

"basic_defs.h" 4.2 ≡

```
  typedef uint32_t WORD;
```

[basic_defs_h.web]

4.3 IP addresses

[basic_defs_h.web]

IP addresses are string of bits: we store them in unsigned integer variables. In IPv4 IP addresses are 32-bits long, so we define them as a **WORD**. Scaling to IPv6, the size of **IP_ADDRESS** should became 128 bits.

Actually in *C* the longest integer type is **long long**, which size is 8 bytes. So scaling to IPv6 probably requires more than changing a **typedef**.

Constant: *IP_ADDRESS_SIZE*
Description: Number of bits of an IP address.
Important: When scaling to IPv6 change this value to 128.

"basic_defs.h" 4.3 ≡

#define *IP_ADDRESS_SIZE* 32

[basic_defs_h.web]

Type: **IP_ADDRESS**
Description: IP address.
Important: When scaling to IPv6 change the following type definition.

"basic_defs.h" 4.3.1 ≡
typedef WORD IP_ADDRESS;

[basic_defs_h.web]

Constant: *MAX_IP_ADDRESS*
Description: Highest IP address.

"basic_defs.h" 4.3.2 ≡

#define *MAX_IP_ADDRESS* ((**IP_ADDRESS**) −1)

[basic_defs_h.web]

4.4 Miscellaneous macros

[basic_defs_h.web]

Macro: *MIN*( )

Description: It returns the minimum of two numbers.

Parameters:
a ← first number
b ← second number

Return value: $\min(a, b)$

"basic_defs.h" 4.4 ≡

```
#define MIN(a, b) (((a) ≤ (b)) ? (a) : (b))
```

[basic_defs_h.web]

Macro: *MAX*( )

Description: It returns the maximum of two numbers.

Parameters:
a ← first number
b ← second number

Return value: $\max(a, b)$

"basic_defs.h" 4.4.1 ≡

```
#define MAX(a, b) (((a) ≥ (b)) ? (a) : (b))
```

[basic_defs_h.web]

Macro: *CHECK_MEMORY_ALLOCATION*( )

Description: It is used just after a call to ***calloc***, ***malloc*** or ***realloc*** to check if the memory is allocated properly (i.e. if the memory allocation function did not returned *NULL*).

Parameters:
(**void** ∗) *ptr* ← pointer to the just allocated memory

Return value: None.

"basic_defs.h" 4.4.2 ≡

```
#define CHECK_MEMORY_ALLOCATION(ptr) \
  { \
  if ((ptr) ≡ NULL) \
    { \
    fprintf(stderr, "CHECK_MEMORY_ALLOCATION():␣Not␣enough␣memory.\n"); \
    abort( ); \
    } \
  }
```

[basic_defs_h.web]

Macro: *PRINT_WORD*( )

Description: It prints a number (of type **WORD**) placing a "," every 3 digits.

Parameters:

(**FILE** ∗) *out*	↔	file of output
(**WORD**) *n*	←	number to print
(**int**) *minWidth*	←	minimum width of printed stuff

Return value: None.

"basic_defs.h" 4.4.3 ≡

```
#define PRINT_WORD(out, n, minWidth) \
  { \
  int __length__, __i__; \
  WORD __n__; \
  \
  if (minWidth > 0) \
    { \
    __length__ = 1; \
    for (__n__ = (n) / 10; __n__ > 0; __n__ /= 10) \
      { \
      __length__++; \
      if (__length__ % 4 == 0) \
        __length__++; \
      } \
    \
    for (__i__ = __length__; __i__ < minWidth; __i__++) \
      fprintf((out), " "); \
    } \
  \
  __n__ = (n); \
  if (__n__ > 1000000000) \
    fprintf((out), "%u,%03u,%03u,%03u", \
        __n__/1000000000, (__n__ % 1000000000) / 1000000, (__n__ % 1000000) / 1000, __n__ % 1000); \
  else if (__n__ > 1000000) \
    fprintf((out), "%u,%03u,%03u", \
        (__n__ % 1000000000) / 1000000, (__n__ % 1000000) / 1000, __n__ % 1000); \
  else if (__n__ > 1000) \
    fprintf((out), "%u,%03u", (__n__ % 1000000) / 1000, __n__ % 1000); \
  else \
    fprintf((out), "%u", __n__ % 1000); \
  }
```

[basic_defs_h.web]

Guard

"basic_defs.h" 4.4.4 ≡

```
#endif    // _AST_BASIC_DEFS_H_
```

[basic_defs_h.web]

5 Routing table based on the AST

[next_hop_h.web]

This section contains the definition of the routing table based on the AST. It consists of an AST, an *anchor table*, a *base vector* and a *next-hop table*. In the following sub-sections we define each of them (in type dependence order) and, after that, the routing table structure.

5.1 Next-hop table

[next_hop_h.web]

The *next-hop table* is an array containing all the possible next-hops addresses. This table is typically very small: in our experiments less than 60 entries. Each record of the base vector (defined in sub-section 5.2 on page 14) and also some nodes (e.g. full leaves and special leaves) of the AST (defined in sub-section 5.4 on page 19) have a pointer to an entry of the next-hop table.

It may happens that none of the prefixes stored in the routing table is a prefix of a certain given IP address. In this case the search function should return the default next-hop. Often, in the search algorithm, we reach an AST node and we simply return the next-hop pointed by it: in these cases it could be difficult to distinguish if the next-hop to return is either one associated to a prefix or the default one. To handle this situation we store the default next-hop in a special position of the table (position 0).

The next-hop table is built in *BuildNextHopTable*( ) function (page 87) in the following way:

1. *BuildNextHopTable*( ) accepts the routing table entries in an initial temporary format: a vector of **ENTRY** records (see its definition in section 6 on page 34);
2. first a temporary array of **NEXT_HOP**s (bigger then the one required at the end) is allocated, and all the next-hops (one next-hop for each entry) are put inside it;
3. then the entries of the temporary array of **NEXT_HOP**s are sorted and the duplicates are removed;
4. finally the definitive array of **NEXT_HOP**s (of a proper size) is allocated and all the next-hops are put in it.

"basic_defs.h" 5.1 ≡
 @O next_hop.h

[next_hop_h.web]

Guard

"next_hop.h" 5.1.0.1 ≡

#ifndef *_AST_NEXT_HOP_H_*
#define *_AST_NEXT_HOP_H_*

[next_hop_h.web]

Includes

"next_hop.h" 5.1.0.2 ≡

```
#include "basic_defs.h"
```

[next_hop_h.web]

Constants

Constant: *DEFAULT_NEXT_HOP*

Description: Default next-hop to return when we a proper prefix of the searched IP address is not found.

"next_hop.h" 5.1.0.3 ≡

```
#define DEFAULT_NEXT_HOP 0
```

[next_hop_h.web]

Constant: *DEFAULT_NEXT_HOP_INDEX*

Description: The default next-hop is also stored in position 0 of the *next-hop table*: this is useful during the search. In fact when we reach a full or a special leaf, and the prefix pointed by it is not a prefix of the searched IP address, we return the next-hop pointed by that leaf. That next-hop could be either a next-hop associated to a prefix or the default next-hop.

"next_hop.h" 5.1.0.4 ≡

```
#define DEFAULT_NEXT_HOP_INDEX 0
```

[next_hop_h.web]

Type: **MAX_NEXT_HOPS**

Description: Maximum # of next-hops. In our experiments we never encountered more than 60 next-hops; for safeness we set this limit to 2^8. In the fields of AST nodes (see sub-section 5.4.1 on page 20) a next-hop index is represented using 8 bits.

"next_hop.h" 5.1.0.5 ≡

```
#define MAX_NEXT_HOPS (1 << 8)
```

[next_hop_h.web]

Types

Type: **NEXT_HOP**

Description: A *next-hop table* entry in IPv4 is a 32-bit string. The *next-hop table* is an array of **NEXT_HOP**s that contains all possible next-hops addresses.

"next_hop.h" 5.1.0.6 ≡

```
typedef IP_ADDRESS NEXT_HOP;
```

[next_hop_h.web]

Guard

"next_hop.h" 5.1.0.7 ≡

```
#endif    // _AST_NEXT_HOP_H_
```

[next_hop_h.web]

5.2 Base vector

[base_h.web]

The *base vector* is an array of **BASE** records. A prefix has naturally associated a segment (the formula to determine its extremes is in sub-section 7.1 on page 42). The AST is built (see section 11 on page 65) upon points which, in this case, are extremes of segments associated to prefixes: for each prefix we consider the left extreme and the right extreme + 1. The base vector contains all points without repetitions (overlapped points are stored only once) and sorted. Each **BASE** record contains one point and its associated next-hop. The next-hop associated to a point is the one that should be returned for all IP addresses that fall between that point (included) and the next one (excluded).

The full and special leaves of the AST have a pointer to an entry of the base vector (see the fields of the leaves of an AST in sub-section 5.4.1 on page 20).

To build the base vector (in *BuildRoutingTable*( ) function, on page 83) we work as follows:

1. initially all the prefixes are stored in a temporary vector of **ENTRY** records (see its definition in section 6 on page 34);
2. first we sort the entries and we remove the duplicates (*SortAndRemoveDuplicatesEntries*( ) function, on page 37);
3. then we build the next-hop table (*BuildNextHopTable*( ) function, on page 87);
4. at this point we build the base vector using *BuildBaseVector*( ) function (page 88).

"next_hop.h" 5.2 ≡
@O base.h

[base_h.web]

Guard

"base.h" 5.2.0.1 ≡

#ifndef *_AST_BASE_H_*
#define *_AST_BASE_H_*

[base_h.web]

Includes

"base.h" 5.2.0.2 ≡

#include "basic_defs.h"

[base_h.web]

Constants

Type: MAX_BASES

Description: Maximum # of bases. In our experiments we never encountered more than 2^{18} bases; for safeness we set this limit to 2^{20}. In the fields of AST nodes (see sub-section 5.4.1 on page 20) a base index is represented using 20 bits.

"base.h" 5.2.0.3 ≡

```
#define MAX_BASES (1 << 20)
```

[base_h.web]

Types

Type: BASE

Description: Record of a *base vector*, where the extremes of segments associated to prefixes are stored.

"base.h" 5.2.0.4 ≡

```
typedef struct BASE
{
IP_ADDRESS point;    // extreme of the segment associated to a prefix
WORD nextHopIndex;    // index of a next-hop in the next-hop table
} BASE;
```

[base_h.web]

Guard

"base.h" 5.2.0.5 ≡

```
#endif    // _AST_BASE_H_
```

[base_h.web]

5.3 Anchor table

[anchor_h.web]

When we build the sub-AST rooted on a certain node (see section 11 on page 65), we consider the interval associated to that node and we subdivide it into sub-intervals (buckets). The buckets are aligned to a grid which starting point is on an anchor point. The points chosen as anchors are stored in the *anchor table.* In each internal node we store the index of the corresponding anchor point in the anchor table. In all our experiments the number of internal nodes (and thus also the number of anchor points) was always less than 2^{17} (more precisely less than 50,000).

The AST is represented as an array and each node consists of an unsigned word. In the bits of each unsigned word we store one or more fields depending on the kind of the node (see sub-section 5.4.1 on page 20). The nodes which contain a pointer to anchor points are *internal nodes*: these nodes are placed in the root of the sub-AST for which we are subdividing the local interval into buckets. A single unsigned word is not enough to store all the fields necessary to an internal node. Thus together with each anchor point we store an additional unsigned word containing the following two fields:

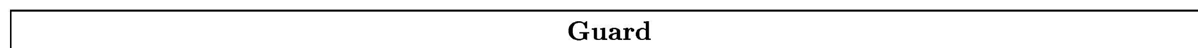
Fields stored near anchor points	*# of bits*	*Pos. in the word*	*# of possibilities*
$\log_2(step)$	7	25–31	$2^7 = 128$
unused	5	20–24	$2^5 = 32$
AST_index	20	0–19	$2^{20} = 1,048,576$

Description of the fields:

- $\log_2(step)$: logarithm of the *step* (length of a regular sub-interval of the interval associated to the internal node); having the logarithm, a division for the *step* can be performed with a shift; the maximum *step* that can be represented is $2^{2^7-1} = 2^{127}$ that is sufficient for IPv6;
- *AST_index*: index, in the AST array (defined in sub-section 5.4 on page 19), of the node which contains the first child on the right of the anchor point; the maximum allowed size of the AST array is 2^{20}, thus 20 bits are enough for this field.

"base.h" 5.3 ≡
  @O anchor.h

[anchor_h.web]



Guard

"anchor.h" 5.3.0.1 ≡

#ifndef *_AST_ANCHOR_H_*
#define *_AST_ANCHOR_H_*

[anchor_h.web]

Includes

"anchor.h" 5.3.0.2 ≡

```
#include "basic_defs.h"
```

[anchor_h.web]

Constants

Type: MAX_ANCHORS

Description: Maximum # of anchor points that can be stored in the anchor table. In our experiments we never encountered more than 2^{17} internal nodes. In the fields of AST nodes (see sub-section 5.4.1 on page 20) an anchor index is represented using 17 bits.

"anchor.h" 5.3.0.3 ≡

```
#define MAX_ANCHORS (1 << 17)
```

[anchor_h.web]

Types

Type: ANCHOR

Description: Record of the *anchor table*, where the anchor points are stored.

"anchor.h" 5.3.0.4 ≡

```
typedef struct ANCHOR
{
IP_ADDRESS point;   // point taken as anchor
WORD fields;   // contains log2(step) and AST_index fields
} ANCHOR;
```

[anchor_h.web]

Macros

Macro: *SET_LOG_STEP*( )

Description: It sets the field $\log_2(step)$ of the **WORD** stored together with the anchor point.

Parameters:
(**int**) *logStep* ← $\log_2(step)$

Return value: (**AST_NODE**) The $\log_2(step)$ in the bits 25–31.

"anchor.h" 5.3.0.5 ≡

```
#define SET_LOG_STEP(logStep) ((logStep) << 25)
```

[anchor_h.web]

Macro: *GET_LOG_STEP*( )
Description: It extracts the field $\log_2(step)$ form the **WORD** stored together with the anchor point.
Parameters: (**WORD**) *fields* ← fields
Return value: (**int**) $\log_2(step)$.

"anchor.h" 5.3.0.6 ≡

```
#define GET_LOG_STEP(fields) ((fields) >> 25)
```

[anchor_h.web]

Macro: *SET_AST_INDEX*( )
Description: It sets the *AST_index* field of the **WORD** stored together with the anchor point.
Parameters: (**WORD**) *ast_index* ← index of a cell of the AST array
Return value: (**AST_NODE**) The *ast_index* in the bits 0–19.

"anchor.h" 5.3.0.7 ≡

```
#define SET_AST_INDEX(ast_index) (ast_index)
```

[anchor_h.web]

Macro: *GET_AST_INDEX*( )
Description: It extracts the *AST_index* field form the **WORD** stored together with the anchor point.
Parameters: (**AST_NODE**) *node* ← node
Return value: (**WORD**) Index of a cell of the AST array.

"anchor.h" 5.3.0.8 ≡

```
#define GET_AST_INDEX(node) ((node) & '3777777)
```

[anchor_h.web]

Guard

"anchor.h" 5.3.0.9 ≡

```
#endif // _AST_ANCHOR_H_
```

[anchor_h.web]

5.4 Nodes of AST

[ast_node_h.web]

The AST is represented as an array and each node consists of an unsigned word. The children of a node are stored in consecutive memory locations. In this way, only a pointer to one of them is needed.

To build the AST (in function *BuildRoutingTable*( ), on page 83) we work as follows:

1. we dynamically allocate an array of **AST_NODE**s which size is bigger then the one required at the end of the construction;
2. we call *BuildAST*( ) function (page 68) which fills in the AST array and returns the effective dimension of the AST;
3. finally we reallocate the array of **AST_NODE**s into one of the proper size, and we store a pointer to it in the final routing table.

The algorithm used to build the AST is described in section 11 on page 65.

"anchor.h" 5.4 ≡
```
@O ast_node.h
```

[ast_node_h.web]

Guard

"ast_node.h" 5.4.0.1 ≡

```
#ifndef _AST_AST_NODE_H_
#define _AST_AST_NODE_H_
```

[ast_node_h.web]

Includes

"ast_node.h" 5.4.0.2 ≡

```
#include "basic_defs.h"
```

[ast_node_h.web]

Constants

Internal constant: *MAX_AST_NODES*

Description: Maximum # of nodes AST can have. In our experiments the # of AST nodes was always less then 2^{19}; for safeness we set this limit to 2^{20}. In the fields of AST nodes (see sub-section 5.4.1 on page 20) the index of a node in the AST array is represented using 20 bits.

"ast_node.h" 5.4.0.3 ≡

```
#define MAX_AST_NODES (1 << 20)
```

[ast_node_h.web]

Internal constant: *MAX_AST_LEVELS*
Description: Maximum # of levels an AST can have.

"ast_node.h" 5.4.0.4 ≡

```
#define MAX_AST_LEVELS IP_ADDRESS_SIZE
```

[ast_node_h.web]

Types

Type: AST_NODE
Description: Node of the AST.

"ast_node.h" 5.4.0.5 ≡

```
typedef WORD AST_NODE;
```

[ast_node_h.web]

5.4.1 Kind of nodes

[ast_node_h.web]

For details about the construction of the AST refer to section 11 on page 65, by now consider that we distinguish five kinds of nodes: *internal nodes*, *full leaves*, *special leaves*, *covered empty leaves* and *uncovered empty leaves*. In the following we first explain the fields of each of them and, after that, we explain the method used to recognize the kind of a node given a word of the AST array.

Internal nodes:

Internal nodes	*# of bits*	*Pos. in the word*	*# of possibilities*
internal_node_flag	1	31–31	2
left_window_width	7	24–30	$2^7 = 128$
right_window_width	7	17–23	$2^7 = 128$
anchor_index	17	0–16	$2^{17} = 131,072$

Description of the fields:

- *internal_node_flag*: set to 1 to recognize that this is an internal node;
- *left_window_width*: number of nodes on the left of the anchor point after which there are only uncovered empty leaves (which are not stored and should not been inspected during the search);

- *right_window_width*: same as *left_window_width* but on the right side;
- *anchor_index*: index in the anchor table of the anchor point used by this node (anchor table is defined in sub-section 5.3 on page 16); more details about the construction of the AST, and the use of anchor points, are given in section 11 on page 65.

The *MAX_WINDOW_WIDTH* constant is the maximum value that can be stored in *left_window_width* and *right_window_width* fields. When the effective value to be stored in one of these two fields exceeds *MAX_WINDOW_WIDTH* we store 0 in that field (this is a way to disable it). During the search, a value of 0 in *left_window_width* or in *right_window_width* means that all the nodes on the left or on the right of the anchor point are stored (also the uncovered empty leaves on the extremes of the interval).

Full leaves:

Full leaves	*# of bits*	*Pos. in the word*	*# of possibilities*
node_kind	3	29–31	$2^3 = 8$
unused	9	20–28	$2^9 = 512$
base_index	20	0–19	$2^{20} = 1,048,576$

Description of the fields:

- *node_kind*: set to 011 (in binary) to recognize that this is a full leaf;
- *base_index*: index in the base vector of the prefix's extreme pointed by this leaf (base vector is defined in sub-section 5.2 on page 14).

Special leaves:

Special leaves	*# of bits*	*Pos. in the word*	*# of possibilities*
node_kind	3	29–31	$2^3 = 8$
unused	1	28–28	2
n_left_bases	4	24–26	$2^4 = 16$
n_right_bases	4	20–23	$2^4 = 16$
base_index	20	0–19	$2^{20} = 1,048,576$

Description of the fields:

- *node_kind*: set to 010 (in binary) to recognize that this is a special leaf;
- *n_left_bases*: number of points on the left of the median point; the maximum number of points that can be pointed by a special leaf is stored in the parameter *parameters→maxSpecialLeaf* (parameters are defined in section 10 on page 63);
- *n_right_bases*: same as *n_left_bases* but after the median child.
- *base_index*: index in the base vector of the median of the prefix's extremes pointed by this leaf (base vector is defined in sub-section 5.2 on page 14).

Covered empty leaves:

Covered empty leaves	*# of bits*	*Pos. in the word*	*# of possibilities*
node_kind	3	29–31	$2^3 = 8$
unused	21	8–28	$2^{21} = 2,097,152$
next_hop_index	8	0–7	$2^8 = 256$

Description of the fields:

- *node_kind*: set to 001 (in binary) to recognize that this is a covered empty leaf;
- *next_hop_index*: index in the next-hop table of the next-hop to return (next-hop table is defined in sub-section 5.1 on page 11); note that *next_hop_index* could also be *DEFAULT_NEXT_HOP_INDEX* which is the index of *DEFAULT_NEXT_HOP* (the next-hop to return when the search fails).

Uncovered empty leaves:

Uncovered empty leaves	*# of bits*	*Pos. in the word*	*# of possibilities*
node_kind	3	29–31	$2^3 = 8$
unused	29	0–28	$2^{23} = 536,870,912$

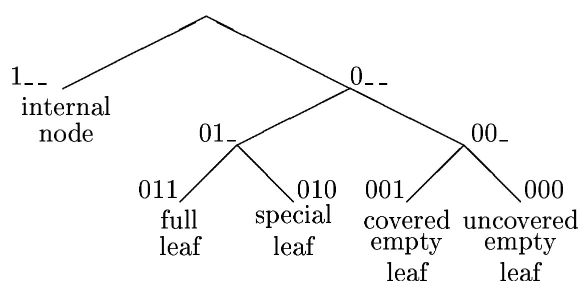
Description of the fields:

- *node_kind*: set to 000 (in binary) to recognize that this is an uncovered empty leaf.

How to recognize the kind of a node during the traverse of the AST:

- *internal nodes* have the *internal_node_flag* field (corresponding to the first bit of the *node_kind* field of the other nodes) set to 1;
- *full leaves* have the *node_kind* filed set to 011 (in binary);
- *special leaves* have the *node_kind* filed set to 010 (in binary);
- *covered empty leaves* have the *node_kind* filed set to 001 (in binary);
- *uncovered empty leaves* have the *node_kind* filed set to 000 (in binary);

When, during the search of the prefix of an IP address (see section 13 on page 92), we start the visit of the AST we assume that its root is not a *special leaf* (as it is natural). When we reach an internal node, we determine the address of one of its child to visit next. Then we need to understand the kind of that child. To determine the kind of a node we use the following condition tree based on the *node_kind* filed:



5.4.2 Internal nodes

[`ast_node_h.web`]

> **Constant:** *SET_INTERNAL_NODE*
>
> **Description:** It sets the *internal_node_flag* field of a node to 1 to specify that it is an internal node.

"ast_node.h" 5.4.2 ≡

```
#define SET_INTERNAL_NODE (1 << 31)
```

[ast_node_h.web]

Macro: *IS_INTERNAL_NODE*( )

Description: It checks if a node is internal by checking if the *internal_node_flag* field is 1.

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is an internal node
FALSE ≡ otherwise

"ast_node.h" 5.4.2.1 ≡

```
#define IS_INTERNAL_NODE(node) ((node) & '20000000000)
```

[ast_node_h.web]

Macro: *FAST_IS_INTERNAL_NODE*( )

Description: It checks if a node is internal by checking if the *internal_node_flag* field is 1.

Note: This macro is the same as *IS_INTERNAL_NODE*( ), but it could take a bit less time.

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is an internal node
FALSE ≡ otherwise

"ast_node.h" 5.4.2.2 ≡

```
#define FAST_IS_INTERNAL_NODE(node) ((signed)(node) < 0)
```

[ast_node_h.web]

Macro: *SET_LEFT_WINDOW_WIDTH*( )

Description: It sets the *left_window_width* field of an internal node.

Parameters:
(**int**) *left_window_width* ← # of nodes on the left after which there are only empty leaves

Return value: (**AST_NODE**) The *left_window_width* in the bits 24–30.

"ast_node.h" 5.4.2.3 ≡

```
#define SET_LEFT_WINDOW_WIDTH(left_window_width) ((left_window_width) << 24)
```

[ast_node_h.web]

Macro: *GET_LEFT_WINDOW_WIDTH*( )

Description: It extracts the *left_window_width* field form an internal node.

Parameters:
(**AST_NODE**) *node* ← node

Return value: (**int**) The number of nodes on the left of the anchor point after which there are only uncovered empty leaves (which are not stored and should not been inspected during the search). A value of 0 means that all the nodes on the left of the anchor point are stored.

"ast_node.h" 5.4.2.4 ≡

```
#define GET_LEFT_WINDOW_WIDTH(node) (((node) >> 24) & '177)
```

[ast_node_h.web]

Macro: *SET_RIGHT_WINDOW_WIDTH*( )

Description: It sets the *right_window_width* field of an internal node.

Parameters:
(**int**) *right_window_width* ← # of nodes on the right after which there are only empty leaves

Return value: (**AST_NODE**) The *right_window_width* in the bits 17–23.

"ast_node.h" 5.4.2.5 ≡

```
#define SET_RIGHT_WINDOW_WIDTH(right_window_width) ((right_window_width) << 17)
```

[ast_node_h.web]

Macro: *GET_RIGHT_WINDOW_WIDTH*( )

Description: It extracts the *right_window_width* field form an internal node.

Parameters:
(**AST_NODE**) *node* ← node

Return value: (**int**) The number of nodes on the right of the anchor point after which there are only uncovered empty leaves (which are not stored and should not been inspected during the search). A value of 0 means that all the nodes on the right of the anchor point are stored.

"ast_node.h" 5.4.2.6 ≡

```
#define GET_RIGHT_WINDOW_WIDTH(node) (((node) >> 17) & '177)
```

[ast_node_h.web]

Constant: *MAX_WINDOW_WIDTH*

Description: Maximum value that can be stored in *left_window_width* and *right_window_width* fields of an internal node.

Note: When the effective value of one of these two fields exceeds *MAX_WINDOW_WIDTH* we store 0 in that field (this is a way to disable it). During the search, a value of 0 in *left_window_width* or in *right_window_width* means that all the nodes on the left or on the right of the anchor point are stored (also the uncovered empty leaves on the extremes of the interval).

"ast_node.h" 5.4.2.7 ≡

```
#define MAX_WINDOW_WIDTH 127
```

[ast_node_h.web]

Macro:	*SET_ANCHOR_INDEX*( )
Description:	It sets the *anchor_index* field of an internal node.
Parameters:	(**WORD**) *anchor_index* ← index of an entry of the anchor table
Return value:	(**AST_NODE**) The *anchor_index* in the bits 0–16.

"ast_node.h" 5.4.2.8 ≡

```
#define SET_ANCHOR_INDEX(anchor_index) (anchor_index)
```

[ast_node_h.web]

Macro:	*GET_ANCHOR_INDEX*( )
Description:	It extracts the *anchor_index* field form an internal node.
Parameters:	(**AST_NODE**) *node* ← node
Return value:	(**WORD**) Index of an entry of the anchor table.

"ast_node.h" 5.4.2.9 ≡

```
#define GET_ANCHOR_INDEX(node) ((node) & '377777)
```

[ast_node_h.web]

5.4.3 Full leaves

[ast_node_h.web]

Constant:	*SET_FULL_LEAF*
Description:	It sets the *node_kind* field of a node to 011 (in binary) to specify that it is a full leaf.

"ast_node.h" 5.4.3 ≡

```
#define SET_FULL_LEAF ('3 << 29)
```

[ast_node_h.web]

Macro: *IS_FULL_LEAF*( )

Description: It checks if a node is a full leaf by checking if the bits of the *node_kind* field are 011.

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is a full leaf
FALSE ≡ otherwise

"ast_node.h" 5.4.3.1 ≡

#define *IS_FULL_LEAF*(*node*) $(((node)\,\&\,{'}34000000000) \equiv {'}14000000000)$

[ast_node_h.web]

Macro: *IS_FULL_OR_SPECIAL_LEAF*( )

Description: It checks if a node is a full or special leaf by checking if the second bit of the *node_kind* is 1.

Important: Before using this macro it should be known that the node in exam is not an internal node (i.e. we have just tested ¬*IS_INTERNAL_NODE*( )).

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is a full or a special leaf
FALSE ≡ otherwise

"ast_node.h" 5.4.3.2 ≡

#define *IS_FULL_OR_SPECIAL_LEAF*(*node*) $((node)\,\&\,{'}10000000000)$

[ast_node_h.web]

Macro: *FAST_IS_FULL_LEAF*( )

Description: It checks if a node is a full leaf by checking if the third bit of the *node_kind* is 1.

Important: This macro is the same as *IS_FULL_LEAF*( ), but it takes a bit less time. Before using this macro it should be known that the node in exam is not an internal node neither an empty leaf (i.e. we have just tested ¬*IS_INTERNAL_NODE*( ) ∧ *IS_FULL_OR_SPECIAL_LEAF*( )).

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is a full leaf
FALSE ≡ otherwise

"ast_node.h" 5.4.3.3 ≡

#define *FAST_IS_FULL_LEAF*(*node*) $((node)\,\&\,{'}4000000000)$

[ast_node_h.web]

Macro: *SET_BASE_INDEX*( )

Description: It sets the *base_index* field of a full or special leaf.

Parameters:
(**WORD**) *base_index* ← index of a cell of the base vector

Return value: (**AST_NODE**) The *base_index* in the bits 0–19.

"ast_node.h" 5.4.3.4 ≡

```
#define SET_BASE_INDEX(base_index) (base_index)
```

[ast_node_h.web]

Macro: *GET_BASE_INDEX*( )

Description: It extracts the *base_index* field form a full or special leaf.

Parameters:
(**AST_NODE**) *node* ← node

Return value: (**WORD**) Index of a cell of the base vector.

"ast_node.h" 5.4.3.5 ≡

```
#define GET_BASE_INDEX(node) ((node) & '3777777)
```

[ast_node_h.web]

5.4.4 Special leaves

[ast_node_h.web]

Constant: *SET_SPECIAL_LEAF*

Description: It sets the *node_kind* field of a node to 010 (in binary) to specify that it is a special leaf.

"ast_node.h" 5.4.4 ≡

```
#define SET_SPECIAL_LEAF ('2 << 29)
```

[ast_node_h.web]

Macro: *IS_SPECIAL_LEAF*( )

Description: It checks if a node is a special leaf by checking if the bits of the *node_kind* field are 010.

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is a special leaf
FALSE ≡ otherwise

"ast_node.h" 5.4.4.1 ≡

#define *IS_SPECIAL_LEAF*(*node*) (((*node*) & ′*34000000000*) ≡ ′*10000000000*)

[ast_node_h.web]

Macro: *SET_N_LEFT_BASES*( )
Description: It sets the *n_left_bases* field of a special leaf.
Parameters:
(**int**) *n_left_bases* ← # of points before the median one
Return value: (**AST_NODE**) The *n_left_bases* in the bits 20–23.

"ast_node.h" 5.4.4.2 ≡

#define *SET_N_LEFT_BASES*(*n_left_bases*) ((*n_left_bases*) << 20)

[ast_node_h.web]

Macro: *GET_N_LEFT_BASES*( )
Description: It extracts the *n_left_bases* field form a special leaf.
Parameters:
(**AST_NODE**) *node* ← node
Return value: (**int**) # of points on the left of the median one.

"ast_node.h" 5.4.4.3 ≡

#define *GET_N_LEFT_BASES*(*node*) (((*node*) >> 20) & ′*17*)

[ast_node_h.web]

Macro: *SET_N_RIGHT_BASES*( )
Description: It sets the *n_right_bases* field of a special leaf.
Parameters:
(**int**) *n_right_children* ← # of points after the median one
Return value: (**AST_NODE**) The *n_right_children* in the bits 24–27.

"ast_node.h" 5.4.4.4 ≡

#define *SET_N_RIGHT_BASES*(*n_right_children*) ((*n_right_children*) << 24)

[ast_node_h.web]

Macro: *GET_N_RIGHT_BASES*( )
Description: It extracts the *n_right_bases* field form a special leaf.
Parameters:
(**AST_NODE**) *node* ← node
Return value: (**int**) # of points on the right of the median one.

"ast_node.h" 5.4.4.5 ≡

```
#define GET_N_RIGHT_BASES(node) (((node) >> 24) & '17)
```

[ast_node_h.web]

5.4.5 Covered empty leaves

[ast_node_h.web]

Constant: *SET_COVERED_EMPTY_LEAF*

Description: It sets the *node_kind* field of a node to 001 (in binary) to specify that it is a covered empty leaf.

"ast_node.h" 5.4.5 ≡

```
#define SET_COVERED_EMPTY_LEAF ('1 << 29)
```

[ast_node_h.web]

Macro: *IS_COVERED_EMPTY_LEAF*( )

Description: It checks if a node is an covered empty leaf by checking if the bits of the *node_kind* field are 001.

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is a covered empty leaf
FALSE ≡ otherwise

"ast_node.h" 5.4.5.1 ≡

```
#define IS_COVERED_EMPTY_LEAF(node) (((node) & '34000000000) ≡ '4000000000)
```

[ast_node_h.web]

Macro: *FAST_IS_COVERED_EMPTY_LEAF*( )

Description: It checks if a node is a covered empty leaf by checking if the third bit of the *node_kind* is 1.

Important: This macro is the same as *IS_COVERED_EMPTY_LEAF*( ), but it takes a bit less time. Before using this macro it should be known that the node in exam is an empty leaf (i.e. we have just tested ¬*IS_INTERNAL_NODE*( ) ∧ ¬*IS_FULL_OR_SPECIAL_LEAF*( )).

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is a covered empty leaf
FALSE ≡ otherwise

"ast_node.h" 5.4.5.2 ≡

#define *FAST_IS_COVERED_EMPTY_LEAF*(*node*) ((*node*) & ´*4000000000*)

[ast_node_h.web]

Macro: *IS_EMPTY_LEAF*( )

Description: It checks if a node is an empty leaf (i.e. a covered empty leaf or an uncovered empty leaf) by checking if the first two bits of the *node_kind* field are 00.

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is a covered empty leaf
FALSE ≡ otherwise

"ast_node.h" 5.4.5.3 ≡

#define *IS_EMPTY_LEAF*(*node*) (((*node*) & ´*30000000000*) ≡ ´*0*)

[ast_node_h.web]

Macro: *SET_NEXT_HOP_INDEX*( )

Description: It sets the *next_hop_index* field of a covered empty leaf.

Parameters:
(**int**) *next_hop_index* ← index of a cell of the next-hop table

Return value: (**AST_NODE**) The *next_hop_index* in the bits 0–7.

"ast_node.h" 5.4.5.4 ≡

#define *SET_NEXT_HOP_INDEX*(*next_hop_index*) (*next_hop_index*)

[ast_node_h.web]

Macro: *GET_NEXT_HOP_INDEX*( )

Description: It extracts the *next_hop_index* field form a covered empty leaf.

Parameters:
(**AST_NODE**) *node* ← node

Return value: (**int**) Index in the next-hop table of the next-hop to return when the prefix's extreme pointed by *base_index* is after the searched IP address (for details about the search algorithm see section 13 on page 92).

"ast_node.h" 5.4.5.5 ≡

#define *GET_NEXT_HOP_INDEX*(*node*) ((*node*) & ´*377*)

[ast_node_h.web]

5.4.6 Uncovered empty leaves

[ast_node_h.web]

Constant: *SET_UNCOVERED_EMPTY_LEAF*

Description: It sets the *node_kind* field of a node to 000 (in binary) to specify that it is an uncovered empty leaf.

"ast_node.h" 5.4.6 ≡

#define *SET_UNCOVERED_EMPTY_LEAF* (´*0* << 29)

[ast_node_h.web]

Macro: *IS_UNCOVERED_EMPTY_LEAF* ( )

Description: It checks if a node is an uncovered empty leaf by checking if the bits of the *node_kind* field are 000.

Parameters:
(**AST_NODE**) *node* ← node

Return value:
TRUE ≡ it is an uncovered empty leaf
FALSE ≡ otherwise

"ast_node.h" 5.4.6.1 ≡

#define *IS_UNCOVERED_EMPTY_LEAF* (*node*) (((*node*) & ´*34000000000*) ≡ 0)

[ast_node_h.web]

Guard

"ast_node.h" 5.4.6.2 ≡

#endif // *_AST_AST_NODE_H_*

[ast_node_h.web]

5.5 Routing table based on the AST

[routing_table_h.web]

This sub-section contains the definition of the routing table structure: it contains an AST, an *anchor table*, a *base vector* and a *next-hop table*

"ast_node.h" 5.5 ≡
```
@O routing_table.h
```

[routing_table_h.web]

Guard

"routing_table.h" 5.5.0.1 ≡

```
#ifndef _AST_ROUTING_TABLE_H_
#define _AST_ROUTING_TABLE_H_
```

[routing_table_h.web]

Includes

"routing_table.h" 5.5.0.2 ≡

```
#include "anchor.h"
#include "ast_node.h"
#include "base.h"
#include "basic_defs.h"
#include "next_hop.h"
```

[routing_table_h.web]

Types

Type: **ROUTING_TABLE_RECORD**
Description: Routing table: it consists of an AST, an *anchor table* a *base vector* and a *next-hop table*. This structure is built in function *BuildRoutingTable*( ) (page 83).

"routing_table.h" 5.5.0.3 ≡
```
typedef struct ROUTING_TABLE_STRUCTURE
{
AST_NODE *ast;    // AST
WORD astSize;
ANCHOR *anchor;    // anchor table
WORD anchorSize;
BASE *base;    // base vector
WORD baseSize;
NEXT_HOP *nextHop;    // next-hop table
```

WORD *nextHopSize*;
} **ROUTING_TABLE_STRUCTURE**;

[routing_table_h.web]

Type: **ROUTING_TABLE**
Description: Pointer to the routing table: this type is used in all the code instead of the previous one.

"routing_table.h" 5.5.0.4 ≡
typedef ROUTING_TABLE_STRUCTURE ∗**ROUTING_TABLE**;

[routing_table_h.web]

Guard

"routing_table.h" 5.5.0.5 ≡

#endif // *_AST_ROUTING_TABLE_H_*

[routing_table_h.web]

6 Entry vector

[entry_h.web]

This section contains the definition of the *entry vector* and some routines to work on it: in particular one to read the routing table entries from a file, and one to sort and remove duplicates.

The *entry vector* is used to temporary store the entries of the routing table before building the final hierarchical structure.

To build a routing table we work as follows:

1. First (function *ReadEntries*( ), page 36) we read the entries of the routing table from a file, and we put them in a simple array (of type **ENTRY**) allocated statically (in function *main*( ), page 123), which size is *MAX_ENTRIES* (the *entry*[ ] vector is an array of pointers to **ENTRY_RECORD**s).
2. Then (function *BuildRoutingTable*( ), page 83) we build a structure of type **ROUTING_TABLE** (defined in sub-section 5.5, page 32): this hierarchical structure contains the same entries of the *entry vector*, but its structure allows very quick searches.

6.1 Entry vector: header

[entry_h.web]

"routing_table.h" 6.1 ≡

@O entry.h

[entry_h.web]

Guard

"entry.h" 6.1.1 ≡

```
#ifndef _AST_ENTRY_H_
#define _AST_ENTRY_H_
```

[entry_h.web]

Includes

"entry.h" 6.1.2 ≡

```
#include <stdio.h>

#include "basic_defs.h"
#include "next_hop.h"
```

[entry_h.web]

Types

Type: **ENTRY_RECORD**

Description: The routing table entries are initially stored in a simple vector of type **ENTRY**: this vector is an array of pointers to **ENTRY_RECORD**s. Each **ENTRY_RECORD** record contains the information of an entry of the routing table.

"entry.h" 6.1.3 ≡

```
typedef struct ENTRY_RECORD
{
IP_ADDRESS string;    // string of bits which contains the prefix in its highest part
WORD length;    // length of the prefix
NEXT_HOP nextHop;    // the corresponding next-hop
} ENTRY_RECORD;
```

[entry_h.web]

Type: **ENTRY**

Description: Pointer to an *entry record* used to define (statically) an array of pointers to *entry records* in function *main*( ) (page 123), and filled in with the entries of the routing table in function *ReadEntries*( ) (page 36).

"entry.h" 6.1.4 ≡

```
typedef ENTRY_RECORD *ENTRY;
```

[entry_h.web]

Prototypes

"entry.h" 6.1.5 ≡

```
long ReadEntries(const char *fileName, ENTRY entry[ ], WORD maxEntries);
WORD SortAndRemoveDuplicatesEntries(ENTRY entry[ ], WORD nEntries);
void PrintEntries(FILE *out, const ENTRY entry[ ], WORD nEntries, WORD n);
void DisposeEntries(ENTRY entry[ ], WORD nEntries);
```

[entry_h.web]

Guard

"entry.h" 6.1.6 ≡

```
#endif    // _AST_ENTRY_H_
```

[entry_h.web]

6.2 Entry vector: implementation

[entry.web]

"entry.h" 6.2 ≡

```
@O entry.c
```

[entry.web]

Includes

"entry.c" 6.2.1 ≡

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

#include "basic_defs.h"
#include "entry.h"
#include "next_hop.h"
#include "prefix.h"
```

[entry.web]

Internal prototypes

"entry.c" 6.2.2 ≡

```
static int CompareEntries(const ENTRY *i, const ENTRY *j);
```

[entry.web]

Functions

Function: *ReadEntries*( )

Description: It reads the routing table entries from a file.

Note: Each entry is represented by three numbers: *string_of_bits*, *length*, and *next-hop* in decimal notation, where *string_of_bits* contains the bits of the prefix in its highest part, *length* is the length of the prefix, and *next-hop* is the corresponding next-hop address.

Parameters:

fileName	←	name of the routing file
entry	↔	array to store the entries of the routing table
maxEntries	←	size of *entry*[ ]: max number of entries that can be read

Return value: Number of read entries or −1 if the number of entries exceeds *maxEntries*.

"entry.c" 6.2.3 ≡

```
long ReadEntries(const char *fileName, ENTRY entry[], WORD maxEntries)
  {
  FILE *inFile;    // pointer to the file containing the entries
  long nEntries = 0;    // # of read entries
```

```
IP_ADDRESS string;   // the routing entry
int length;   // length of the routing entry
NEXT_HOP nextHop;   // the corresponding next-hop

  // ———————————— Open the routing file
if (¬(inFile = fopen(fileName, "rb")))
  {
  perror(fileName);
  abort( );
  }

  // ———————————— Read the entries
while (fscanf(inFile, "%u%i%u", &string, &length, &nextHop) ≠ EOF)
  {
  if (nEntries ≥ maxEntries)
    return −1;

  entry[nEntries] = (ENTRY) malloc(sizeof(ENTRY_RECORD));
  CHECK_MEMORY_ALLOCATION(entry[nEntries]);

    /* Note: we clear the IP_ADDRESS_SIZE − lenght last bits: this shouldn't be necessary if the
       routing table data was consistent */
  string = string >> (IP_ADDRESS_SIZE − length) << (IP_ADDRESS_SIZE − length);

  entry[nEntries]→string = string;
  entry[nEntries]→length = length;
  entry[nEntries]→nextHop = nextHop;

  nEntries++;
  }

  // ———————————— Return the # of read entries
return nEntries;
}
```

[entry.web]

Function: *SortAndRemoveDuplicatesEntries*( )

Description: It sorts the entries of the routing table and remove duplicates.

Note: This function disposes the entries that are duplicate.

Parameters:

entry	↔	routing table entries stored in the initial simple array
nEntries	←	# of entries of the routing table

Return value: Number of unique entries.

"entry.c" 6.2.4 ≡

```
WORD SortAndRemoveDuplicatesEntries(ENTRY entry[ ], WORD nEntries)
  {
  WORD i;   // index
  WORD nUniqueEntries;   // # of routing entries after duplicate removal

    // ———————————— Sort the entries
  qsort(entry, nEntries, sizeof(ENTRY), (int (*)(const void *, const void *)) CompareEntries);

    // ———————————— Remove duplicates
```

```
nUniqueEntries = nEntries > 0 ? 1 : 0;
for (i = 1; i < nEntries; i++)
  if (CompareEntries(&entry[nUniqueEntries − 1], &entry[i]) ≠ 0)
    {
    if (nUniqueEntries ≠ i)
      {
      if (entry[nUniqueEntries] ≠ NULL)
        free(entry[nUniqueEntries]);

      entry[nUniqueEntries] = entry[i];
      entry[i] = NULL;
      }
    nUniqueEntries++;
    }

for (i = nUniqueEntries; i < nEntries; i++)
  if (entry[i] ≠ NULL)
    free(entry[i]);

  // ———————————— Return the # of unique entries
return nUniqueEntries;
}
```

[entry.web]

Function: *PrintEntries*( )		
Description: It prints the first n lines of the initial array where the routing table is stored.		
Parameters:		
out	↔	file of output
entry	←	routing table entries stored in the initial simple array
nEntries	←	# of entries of the routing table
n	←	# of entries to print
Return value: None.		

"entry.c" 6.2.5 ≡

```
void PrintEntries(FILE *out, const ENTRY entry[ ], WORD nEntries, WORD n)
  {
  WORD i;    // index to scan entries
  int j;   // index to extract the bits of each prefix

  if (nEntries > n)
    nEntries = n;

  fprintf(out, "\nEntries:\n");
  fprintf(out, "␣␣next-hop␣␣entry\n");

  for (i = 0; i < nEntries; i++)
    {
    fprintf(out, "␣%010u␣", entry[i]→nextHop);

    for (j = 0; j < entry[i]→length; j++)
      {
      fprintf(out, "%1d", EXTRACT(entry[i]→string, j, 1));
      if (j % 8 ≡ 7)
        printf("␣");
```

```
    }
  fprintf(out, "\n");
  }
fflush(out);
}
```

[entry.web]

Function: *DisposeEntries*( )

Description: It dispose the entries.

Important: *entry* is allocated statically in *main*( ) (page 16.0.0.5).

Parameters:

entry	↔	routing table entries stored in the initial simple array
nEntries	←	# of entries of the routing table

Return value: None.

"entry.c" 6.2.6 ≡

```
void DisposeEntries(ENTRY entry[ ], WORD nEntries)
  {
  WORD i;

  for (i = 0; i < nEntries; i++)
    free(entry[i]);
  }
```

[entry.web]

Internal functions

Internal function: *CompareEntries*( )

Description: It compares two routing table entries.

Note: It is used by ***qsort***.

Parameters:

i	←	first entry
j	←	second entry

Return value:

-1	≡	$i.string < j.string$
1	≡	$i.string > j.string$
-1	≡	$i.string = j.string \wedge i.length < j.length$
1	≡	$i.string = j.string \wedge i.length > j.length$
0	≡	$i.string = j.string \wedge i.length = j.length$

"entry.c" 6.2.7 ≡

```
static int CompareEntries(const ENTRY *i, const ENTRY *j)
  {
  if ((*i)→string < (*j)→string)
    return −1;
  else if ((*i)→string > (*j)→string)
```

```
    return 1;
  else if ((*i)→length < (*j)→length)
    return −1;
  else if ((*i)→length > (*j)→length)
    return 1;
  else
    return 0;
  }
```

[entry.web]

7 Prefixes

[prefix_h.web]

This section contains the definition of a prefix and some macros to operate on them.

A prefix of an IP address is a string of bits which length is less or equal then *IP_ADDRESS_SIZE*. To store a prefix we need two variables:

- a *string* of bit (of type **IP_ADDRESS**) which contains the bits of the prefix in its highest part,
- and an integer indicating the *length* of the prefix ($0 < length \leq IP_ADDRESS_SIZE$).

"entry.c" 7 ≡
```
@O prefix.h
```

[prefix_h.web]

Guard

"prefix.h" 7.0.1 ≡

```
#ifndef _AST_PREFIX_H_
#define _AST_PREFIX_H_
```

[prefix_h.web]

Includes

"prefix.h" 7.0.2 ≡

```
#include "basic_defs.h"
```

[prefix_h.web]

Types

"prefix.h" 7.0.3 ≡

```
    // ——————————————— Prefix
typedef struct PREFIX
{
IP_ADDRESS string;    // string of bits which contains the prefix in its highest part
WORD length;    // length of the prefix
} PREFIX;
```

[prefix_h.web]

7.1 Segment associated to a prefix

[prefix_h.web]

A prefix has naturally associated a segment, which extremes are:

$$left = string$$

$$right = string + 2^{IP_ADDRESS_SIZE\ -\ length} - 1$$

The following two macros computes the extremes of a segment associated to a prefix.

Macro: *PREFIX_LEFT_EXTREME*( )

Description: It computes the left extreme of the segment associated to a prefix.

Important: This macro works well also when the type of *prefix* is not **PREFIX** but contains the same fields of **PREFIX** (e.g. it works also when the type of *prefix* is **ENTRY_RECORD**).

Parameters:
(**PREFIX**) *prefix* ← prefix

Return value: (**IP_ADDRESS**) The left extreme of the segment associated to a prefix.

"prefix.h" 7.1 ≡

```
#define PREFIX_LEFT_EXTREME(prefix) \
  ((prefix).string)
```

[prefix_h.web]

Macro: *PREFIX_RIGHT_EXTREME*( )

Description: It computes the right extreme of the segment associated to a prefix.

Important: This macro works well also when the type of *prefix* is not **PREFIX** but contains the same fields of **PREFIX** (e.g. it works also when the type of *prefix* is **ENTRY_RECORD**).

Parameters:
(**PREFIX**) *prefix* ← prefix

Return value: (**IP_ADDRESS**) The right extreme of the segment associated to a prefix.

"prefix.h" 7.1.1 ≡

```
#define PREFIX_RIGHT_EXTREME(prefix) \
  ((prefix).string + (1 << (IP_ADDRESS_SIZE - (prefix).length)) - 1)
```

[prefix_h.web]

7.2 Macros to operate on prefixes

[prefix_h.web]

Macro: *EXTRACT*( )

Description: It extracts n bits from a *string* starting at a given *position*.

Note: If n is equal to 0 *EXTRACT*( ) returns 0.

Parameters:

(**IP_ADDRESS**) *string*	←	string
(**int**) *position*	←	position from which the extraction should start
(**int**) n	←	# of bits to extract

Return value: (**IP_ADDRESS**) The extracted bits.

"prefix.h" 7.2 ≡

```
#define EXTRACT(string, position, n) \
  (((string) << (position)) >> (IP_ADDRESS_SIZE - (n)))
```

[prefix_h.web]

Macro: *EXTRACT_LEFT*( )

Description: It extracts n bits from a *string* starting from left.

Note: It is equivalent to *EXTRACT*(*string*, 0, n).

Note: If n is equal to 0 *EXTRACT_LEFT*( ) returns 0.

Parameters:

(**IP_ADDRESS**) *string*	←	string
(**int**) n	←	# of bits to extract

Return value: (**IP_ADDRESS**) The extracted bits.

"prefix.h" 7.2.1 ≡

```
#define EXTRACT_LEFT(string, n) \
  ((string) >> (IP_ADDRESS_SIZE - (n)))
```

[prefix_h.web]

Macro: *REMOVE*( )

Description: It removes the first n bits from a *string*.

Parameters:

(**IP_ADDRESS**) *string*	←	string
(**int**) n	←	# of bits to remove

Return value: (**IP_ADDRESS**) The string without the removed bits.

"prefix.h" 7.2.2 ≡

```
#define REMOVE(string, n) \
  (((string) << (n)) >> (n))
```

[prefix_h.web]

Macro: *IS_PREFIX*( )

Description: It checks if one *prefix* is itself a prefix of another one.

Note: *EXTRACT_LEFT*( ) can not handle 0-bit length prefixes.

Important: This macro works well also when the type of p and q is not **PREFIX** but contains the same fields of **PREFIX** (e.g. it works also when the type of p and q is **ENTRY_RECORD**).

Parameters:
(**PREFIX**) p ← first prefix
(**PREFIX**) q ← second prefix

Return value:
TRUE ≡ p is a prefix q
FALSE ≡ otherwise

"prefix.h" 7.2.3 ≡

```
#define IS_PREFIX(p, q) \
  ( \
  ((p).length ≡ 0) ∨ \
  (((p).length ≤ (q).length) ∧ \
  (EXTRACT_LEFT((p).string, (p).length) ≡ EXTRACT_LEFT((q).string, (p).length))) \
  )
```

[prefix_h.web]

Macro: *PREFIX_MATCH*( )

Description: It checks if a prefix matches an IP address (i.e. if it is a prefix of that IP address).

Important: This macro works well also when the type of p is not **PREFIX** but contains the same fields of **PREFIX** (e.g. it works also when the type of p is **ENTRY_RECORD**).

Parameters:
(**PREFIX**) p ← prefix
(**IP_ADDRESS**) a ← IP address

Return value:
TRUE ≡ the prefix p matches the IP address a
FALSE ≡ otherwise

"prefix.h" 7.2.4 ≡

```
#define PREFIX_MATCH(p, a) \
  (EXTRACT_LEFT((p).string, (p).length) ≡ EXTRACT_LEFT(a, (p).length))
```

[prefix_h.web]

Guard

"prefix.h" 7.2.5 ≡

```
#endif  // _AST_PREFIX_H_
```

[prefix_h.web]

8 Macros to explore buckets

[bucket_h.web]

When we build the AST or when we explore it (e.g. to compute statistics about it), we are given an interval associated to a node and we need to explore the buckets (sub-intervals) in which it is subdivided. The given interval is subdivided into buckets according to an infinite grid: in this way all the buckets have the same lenght except the first and the last that can be smaller. The grid is aligned on an anchor point.

We know the extremes of that interval (*left* and *right*), the *anchor* point and the *step* (length of a regular sub-interval)[1]. When we iterate over each sub-interval we want to know its extremes (*subLeft* and *subRight*). The buckets can be visited in two different ways: from the anchor point to the extremes, or from *left* to *right*. The iterators should be used in the following way:

*WHILE_EXPLORE_BUCKETS_FROM_*... (..., *subLeft*, *subRight*)
{

... do something using *subLeft* and *subRight* ...

} *END_EXPLORE_BUCKETS_FROM_*... (..., *subLeft*, *subRight*) ;

During the construction of the AST we know the branching factor and we need to compute the step, so before the iterators we define the macros to compute it.

"prefix.h" 8 ≡
@O bucket.h

[bucket_h.web]

Guard

"bucket.h" 8.0.1 ≡

#ifndef *_AST_BUCKET_H_*
#define *_AST_BUCKET_H_*

[bucket_h.web]

Includes

"bucket.h" 8.0.2 ≡

#include "basic_defs.h"

[bucket_h.web]

[1] $step \equiv 0$ means a step of $2^{IP_ADDRESS_SIZE}$

8.1 Compute the step

[bucket_h.web]

Macro: *GREATER_EQUAL_POWER2*( )

Description: It computes the smallest power of 2 greater or equal to the given number.

Parameters:

(**IP_ADDRESS**) *number*	←	given number
(**IP_ADDRESS**) *result*	→	smallest power of 2 greater or equal to *number*
(**int**) *logResult*	→	$\log_2(result)$

Return value: None.

"bucket.h" 8.1 ≡

```
#define GREATER_EQUAL_POWER2(number, result, logResult) \
  { \
  IP_ADDRESS __number__, __originalNumber__; \
  \
  __number__ = __originalNumber__ = (number); \
  \
  (result) = 1; \
  (logResult) = 0; \
  \
  while (__number__ > 1)
    { \
    __number__ >>= 1; \
    (result) <<= 1; \
    (logResult)++; \
    } \
  \
  if ((result) ≠ __originalNumber__)
    { \
    (result) <<= 1; \
    (logResult)++; \
    } \
  }
```

[bucket_h.web]

Macro: *COMPUTE_STEP*( )

Description: It computes the *step* (length of a regular sub-interval).

Note: *step* ≡ 0 means a step of $2^{IP_ADDRESS_SIZE}$

Parameters:

(**IP_ADDRESS**) *left*	←	left extreme of the interval
(**IP_ADDRESS**) *right*	←	right extreme of the interval
(**int**) *logBranch*	←	$\log_2$(branching factor)
(**IP_ADDRESS**) *step*	→	step
(**int**) *logStep*	→	$\log_2(step)$

Return value: None.

"bucket.h" 8.1.1 ≡

```
#define COMPUTE_STEP(left, right, logBranch, step, logStep) \
  { \
  if (((left) ≡ 0) ∧ ((right) ≡ MAX_IP_ADDRESS))
    { \
    (logStep) = IP_ADDRESS_SIZE − (logBranch); \
    (step) = ((logStep) ≡ IP_ADDRESS_SIZE) ? 0 : (1 << (logStep)); \
    } \
  else \
    GREATER_EQUAL_POWER2(((right) − (left) + 1) >> (logBranch), (step), (logStep)); \
  }
```

[bucket_h.web]

8.2 Explore buckets from anchor to extremes

[bucket_h.web]

The extremes of the buckets (sub-intervals) computed from the anchor point to the extremes are:

$subLeft$:	$subRight$:
$left$	$anchor - x \cdot step - 1$
$\ldots$	$\ldots$
$anchor - 2 \cdot step$	$anchor - step - 1$
$anchor - step$	$anchor - 1$
$anchor$	$anchor + step - 1$
$anchor + step$	$anchor + 2 \cdot step - 1$
$anchor + 2 \cdot step$	$anchor + 3 \cdot step - 1$
$\ldots$	$\ldots$
$anchor + y \cdot step$	$right$

Macro: *WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT*( )

Description: Iterator to explore the buckets (sub-intervals) from the *anchor* point to the *right* extreme.

Important: We assume $anchor \leq right$.

Parameters:

(**IP_ADDRESS**) *anchor*	←	anchor point
(**IP_ADDRESS**) *right*	←	right extreme of the interval
(**IP_ADDRESS**) *step*	←	step (= length of a regular sub-interval)
(**IP_ADDRESS**) *subLeft*	→	left extreme of the current sub-interval
(**IP_ADDRESS**) *subRight*	→	right extreme of the current sub-interval

Return value: None.

"bucket.h" 8.2 ≡

```
#define WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT(anchor, right, step,\
  subLeft, subRight) \
```

```
{ \
(subLeft) = (anchor); \
\
while ((subLeft) ≤ (right)) \
  { \
  (subRight) = (((right) − (subLeft) ≥ (step)) ∧ ((step) ≠ 0)) ? (subLeft) + (step) − 1 : (right);
```

[bucket_h.web]

Macro: *END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT*( )

Description: It closes the iterator to explore the buckets (sub-intervals) from the *anchor* point to the *right* extreme.

Parameters:

(**IP_ADDRESS**) *anchor*	←	anchor point
(**IP_ADDRESS**) *right*	←	right extreme of the interval
(**IP_ADDRESS**) *step*	←	step (= length of a regular sub-interval)
(**IP_ADDRESS**) *subLeft*	→	left extreme of the current sub-interval
(**IP_ADDRESS**) *subRight*	→	right extreme of the current sub-interval

Return value: None.

"bucket.h" 8.2.1 ≡

```
#define END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT(anchor, right, step, \
  subLeft, subRight) \
    if (((right) − (subLeft) ≥ (step)) ∧ ((step) ≠ 0)) \
      (subLeft) += (step); \
    else \
      break; \
    } \
  }
```

[bucket_h.web]

Macro: *WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT*( )

Description: Iterator to explore the buckets (sub-intervals) from the *anchor* point to the *left* extreme.

Important: We assume $anchor \geq left$.

Parameters:

(**IP_ADDRESS**) *anchor*	←	anchor point
(**IP_ADDRESS**) *left*	←	left extreme of the interval
(**IP_ADDRESS**) *step*	←	step (= length of a regular sub-interval)
(**IP_ADDRESS**) *subLeft*	→	left extreme of the current sub-interval
(**IP_ADDRESS**) *subRight*	→	right extreme of the current sub-interval

Return value: None.

"bucket.h" 8.2.2 ≡

```
#define WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT(anchor, left, step, \
  subLeft, subRight) \
```

```
{ \
(subRight) = (anchor) − 1; \
\
while (((anchor) > (left)) ∧ ((subRight) ≥ (left))) \
  { \
  (subLeft) = (((subRight) − (left) ≥ (step)) ∧ ((step) ≠ 0)) ? (subRight) − (step) + 1 : (left);
```

[bucket_h.web]

Macro: *END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT*( )

Description: It closes the iterator to explore the buckets (sub-intervals) from the *anchor* point to the *left* extreme.

Parameters:

(**IP_ADDRESS**) *anchor*	←	anchor point
(**IP_ADDRESS**) *left*	←	left extreme of the interval
(**IP_ADDRESS**) *step*	←	step (= length of a regular sub-interval)
(**IP_ADDRESS**) *subLeft*	→	left extreme of the current sub-interval
(**IP_ADDRESS**) *subRight*	→	right extreme of the current sub-interval

Return value: None.

"bucket.h" 8.2.3 ≡

```
#define END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT(anchor, left, step, \
  subLeft, subRight) \
    if (((subRight) − (left) ≥ (step)) ∧ ((step) ≠ 0)) \
      (subRight) −= (step); \
    else \
      break; \
    } \
  }
```

[bucket_h.web]

8.3 Explore buckets from left to right

[bucket_h.web]

If we consider that when we choose an anchor the grid is shifted by

$$shift = (anchorLeft - left) \,\%\, step$$

the extremes of the buckets (sub-intervals) from *left* to *right* are:

$subLeft$:	$subRight$:
$left$	$left + shift - 1$
$left + shift$	$left + shift + step - 1$
$left + shift + step$	$left + shift + 2 \cdot step - 1$
$left + shift + 2 \cdot step$	$left + shift + 3 \cdot step - 1$
...	...
$anchor$	$anchor + step - 1$
$anchor + step$	$anchor + 2 \cdot step - 1$
$anchor + 2 \cdot step$	$anchor + 3 \cdot step - 1$
...	...
$anchor + y \cdot step$	$right$

Macro: *WHILE_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT*( )

Description: Iterator to explore the buckets (sub-intervals) from the *left* to the *right* extremes.

Important: We assume *anchor* ≥ *left*.

Parameters:

(**IP_ADDRESS**) *left*	←	left extreme of the interval
(**IP_ADDRESS**) *right*	←	right extreme of the interval
(**IP_ADDRESS**) *anchor*	←	left extreme of the sub-interval covered by anchor-leaf
(**IP_ADDRESS**) *step*	←	step (= length of a regular sub-interval)
(**IP_ADDRESS**) *subLeft*	→	left extreme of the current sub-interval
(**IP_ADDRESS**) *subRight*	→	right extreme of the current sub-interval

Return value: None.

"bucket.h" 8.3 ≡

```
#define WHILE_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, anchor, step,\
  subLeft, subRight) \
  {\
  (subLeft) = (left);\
  if (step > 0)\
    {\
    IP_ADDRESS __shift__ = ((anchor) - (left)) % (step);\

    (subRight) = (__shift__ > 0) ? (left) + __shift__ - 1 :\
    (((right) - (left) >= (step)) ? (left) + (step) - 1 : (right));\
    }\
  else\
    (subRight) = ((anchor) > (left)) ? (anchor) - 1 : (right);\
  \
  while ((subLeft) <= (right)) \
    {\
```

[bucket_h.web]

Macro: *END_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT* ( )

Description: It closes the iterator to explore the buckets (sub-intervals) from the *left* to the *right* extremes.

Parameters:

(**IP_ADDRESS**) *left*	←	left extreme of the interval
(**IP_ADDRESS**) *right*	←	right extreme of the interval
(**IP_ADDRESS**) *anchor*	←	left extreme of the sub-interval covered by anchor-leaf
(**IP_ADDRESS**) *step*	←	step (= length of a regular sub-interval)
(**IP_ADDRESS**) *subLeft*	→	left extreme of the current sub-interval
(**IP_ADDRESS**) *subRight*	→	right extreme of the current sub-interval

Return value: None.

"bucket.h" 8.3.1 ≡

```
#define END_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, anchor, step,\
    subLeft, subRight) \
      if ((subRight) < (right)) \
        (subLeft) = (subRight) + 1; \
      else \
        break; \
      \
      if (((right) − (subRight) ≥ (step)) ∧ ((step) ≠ 0)) \
        (subRight) += (step); \
      else \
        (subRight) = (right); \
      } \
    }
```

[bucket_h.web]

Guard

"bucket.h" 8.3.2 ≡

```
#endif   // _AST_BUCKET_H_
```

[bucket_h.web]

9 Priority queue

[p_queue_h.web]

The priority queues are needed for the construction of the AST (for more details see section 11 on page 65).

9.1 Priority queue: header

[p_queue_h.web]

"bucket.h" 9.1 ≡
@O p_queue.h

[p_queue_h.web]

Guard

"p_queue.h" 9.1.1 ≡

```
#ifndef _AST_P_QUEUE_H_
#define _AST_P_QUEUE_H_
```

[p_queue_h.web]

Includes

"p_queue.h" 9.1.2 ≡

```
#include "basic_defs.h"
```

[p_queue_h.web]

Types

Type: POINT
Description: The index of a point in the base vector and its priority in the priority queue.

"p_queue.h" 9.1.3 ≡

```
typedef struct POINT
{
IP_ADDRESS key;    // key of a point
WORD baseIndex;    // index of a point in the base vector
} POINT;
```

[p_queue_h.web]

Type: **POINTS_PQ**
Description: Min-priority queue of points.

"p_queue.h" 9.1.4 ≡
```
typedef struct POINTS_PQ
{
  /*@null@*/ POINT *point;   // binary heap of points (stored in an array)
WORD nPoints;   // # of points in the priority queue
WORD maxPoints;   // maximum # of points that can be stored in the queue
} POINTS_PQ;
```

[p_queue_h.web]

Type: **INTERVAL**
Description: The index of an interval and its priority in the priority queue.

"p_queue.h" 9.1.5 ≡
```
typedef struct INTERVAL
{
WORD key;   // key of an interval
WORD index;   // index of an interval
} INTERVAL;
```

[p_queue_h.web]

Type: **INTERVAL_PQ**
Description: Max-priority queue of intervals.
Important: We assume that the indices of the interval go from 0 to *nIntervals* − 1 and are one different to the others.

"p_queue.h" 9.1.6 ≡
```
typedef struct INTERVALS_PQ
{
  /*@null@*/ INTERVAL *interval;   // binary heap of intervals (stored in an array)
WORD nIntervals;   // # of intervals in the max-priority queue
WORD maxIntervals;   // maximum # of intervals that can be stored in the queue
WORD *index;   // array directly access the intervals in interval
} INTERVALS_PQ;
```

[p_queue_h.web]

Prototypes

"p_queue.h" 9.1.7 ≡
```
    // ———————————————— Min-priority queue of points
void InitializePointsPQ(POINTS_PQ *pointsPQ, WORD maxPoints);
BOOL IsEmptyPointsPQ(const POINTS_PQ *pointsPQ);
void InsertPointsPQ(POINTS_PQ *pointsPQ, IP_ADDRESS key, WORD baseIndex);
WORD ExtractPointsPQ(POINTS_PQ *pointsPQ);
```

```
WORD GetMinKeyPointsPQ(const POINTS_PQ *pointsPQ);
void DisposePointsPQ(POINTS_PQ *pointsPQ);

  // ———————————————— Max-priority queue of intervals
void InitializeIntervalsPQ(INTERVALS_PQ *intervalsPQ, WORD maxIntervals);
void InsertIntervalsPQ(INTERVALS_PQ *intervalsPQ, WORD key, WORD index);
WORD IncreaseKeyIntervalsPQ(INTERVALS_PQ *intervalsPQ, WORD index);
WORD DecreaseKeyIntervalsPQ(INTERVALS_PQ *intervalsPQ, WORD index);
WORD GetMaxKeyIntervalsPQ(const INTERVALS_PQ *intervalsPQ);
void DisposeIntervalsPQ(INTERVALS_PQ *intervalsPQ);
```

[p_queue_h.web]

Guard

"p_queue.h" 9.1.8 ≡

```
#endif    // _AST_P_QUEUE_H_
```

[p_queue_h.web]

9.2 Priority queue: implementation

[p_queue.web]

"p_queue.h" 9.2 ≡
```
@O p_queue.c
```

[p_queue.web]

Includes

"p_queue.c" 9.2.1 ≡

```
#include <stdio.h>
#include <stdlib.h>

#include "basic_defs.h"
#include "p_queue.h"
```

[p_queue.web]

Functions

Function: *InitializePointsPQ*( )

Description: It initializes an empty min-priority queue of points.

Parameters:

pointsPQ	→	min-priority queue of points
maxPoints	←	maximum # of points it can contain

Return value: None.

"p_queue.c" 9.2.2 ≡
```
void InitializePointsPQ(POINTS_PQ *pointsPQ, WORD maxPoints)
  {
  pointsPQ->point = (POINT *) calloc(maxPoints, sizeof(POINT));
  CHECK_MEMORY_ALLOCATION(pointsPQ->point);
  pointsPQ->nPoints = 0;
  pointsPQ->maxPoints = maxPoints;
  }
```

[p_queue.web]

Function: *IsEmptyPointsPQ*( )

Description: It checks if a min-priority queue of points is empty.

Parameters:

pointsPQ	←	min-priority queue of points

Return value:

TRUE	≡	*pointsPQ* is empty
FALSE	≡	otherwise

"p_queue.c" 9.2.3 ≡

```
BOOL IsEmptyPointsPQ(const POINTS_PQ *pointsPQ)
  {
  return pointsPQ→nPoints ≡ 0;
  }
```

[p_queue.web]

Function:	*InsertPointsPQ*( )
Description:	It inserts a point in a min-priority queue of points.
Important:	We assume that the priority queue is right initialized.
Parameters:	
pointsPQ ↔	min-priority queue of points
key ←	priority of the point
baseIndex ←	index of the point in the base vector
Return value:	None.

"p_queue.c" 9.2.4 ≡

```
void InsertPointsPQ(POINTS_PQ *pointsPQ, IP_ADDRESS key, WORD baseIndex)
  {
  WORD i;    // index of the point currently examined
  POINT tempPoint;    // temporary point used for swapping

    // ———————————————— Check heap space
  if (pointsPQ→point ≡ NULL)
    {
    fprintf(stderr, "InsertPointsPQ():␣Uninitialized␣heap.\n");
    abort( );
    }

  if (pointsPQ→nPoints ≥ pointsPQ→maxPoints)
    {
    fprintf(stderr, "InsertPointsPQ():␣Maximum␣number␣of␣points␣exceeded.\n");
    abort( );
    }

    // ———————————————— Add new point to the heap
  pointsPQ→point[pointsPQ→nPoints].key = key;
  pointsPQ→point[pointsPQ→nPoints].baseIndex = baseIndex;
  pointsPQ→nPoints++;

    // ———————————————— Heapify
  i = pointsPQ→nPoints;
  while ((i > 1) ∧ (pointsPQ→point[i / 2 − 1].key > pointsPQ→point[i − 1].key))
    {
    tempPoint = pointsPQ→point[i / 2 − 1];
    pointsPQ→point[i / 2 − 1] = pointsPQ→point[i − 1];
    pointsPQ→point[i − 1] = tempPoint;
    i /= 2;
    }
  }
```

[p_queue.web]

Function: *ExtractPointsPQ*( )

Description: It extracts the point with minimum priority from the priority queue of points.

Important: We assume that the priority queue is right initialized.

Important: We assume that the priority queue is not empty.

Parameters:
pointsPQ ↔ min-priority queue of points

Return value: Base index of the point with minimum priority.

"p_queue.c" 9.2.5 ≡

```
WORD ExtractPointsPQ(POINTS_PQ *pointsPQ)
  {
  WORD baseIndex;   // base index of the point with minimum priority
  WORD i, j;   // indices of points currently examined
  POINT tempPoint;   // temporary point used for swapping

    // ———————————— Check emptiness
  if (pointsPQ→nPoints ≡ 0)
    {
    fprintf(stderr, "ExtractPointsPQ():␣Priority␣queue␣is␣empty.\n");
    abort( );
    }
  if (pointsPQ→point ≡ NULL)
    {
    fprintf(stderr, "ExtractPointsPQ():␣Uninitialized␣heap.\n");
    abort( );
    }

    // ———————————— Get the point with minimum priority
  baseIndex = pointsPQ→point[0].baseIndex;

    // ———————————— Move last item to first position
  pointsPQ→nPoints−−;
  pointsPQ→point[0] = pointsPQ→point[pointsPQ→nPoints];

    // ———————————— Heapify
  i = 1;
  while (i ≤ pointsPQ→nPoints / 2)
    {
    j = 2 * i;
    if ((j < pointsPQ→nPoints) ∧ (pointsPQ→point[j − 1].key > pointsPQ→point[j].key))
      j++;

    if (pointsPQ→point[i − 1].key ≤ pointsPQ→point[j − 1].key)
      break;

    tempPoint = pointsPQ→point[i − 1];
    pointsPQ→point[i − 1] = pointsPQ→point[j − 1];
    pointsPQ→point[j − 1] = tempPoint;
    i = j;
    }

    // ———————————— Return the point with minimum priority
  return baseIndex;
```

```
}
```

[p_queue.web]

Function:	*GetMinKeyPointsPQ*( )
Description:	It returns the minimum key over all points stored in a min-priority queue of points.
Important:	We assume that the min-priority queue is not empty.
Parameters:	
pointsPQ ←	min-priority queue of points
Return value:	The minimum key.

"p_queue.c" 9.2.6 ≡

```
WORD GetMinKeyPointsPQ(const POINTS_PQ *pointsPQ)
  {
    // ———————————— Check heap
  if (pointsPQ→point ≡ NULL)
    {
    fprintf(stderr, "GetMinKeyPointsPQ():␣Uninitialized␣heap.\n");
    abort( );
    }   // ———————————— Check emptiness
  if (pointsPQ→nPoints ≡ 0)
    {
    fprintf(stderr, "GetMinKeyPointsPQ():␣Priority␣queue␣is␣empty.\n");
    abort( );
    }
    // ———————————— Return minimum key
  return pointsPQ→point[0].key;
  }
```

[p_queue.web]

Function:	*DisposePointsPQ*( )
Description:	It disposes a min-priority queue of points.
Parameters:	
pointsPQ ↔	min-priority queue of points
Return value:	None.

"p_queue.c" 9.2.7 ≡

```
void DisposePointsPQ(POINTS_PQ *pointsPQ)
  {
  free(pointsPQ→point);
  pointsPQ→point = (POINT *) NULL;
  }
```

[p_queue.web]

Function: *InitializeIntervalsPQ*( )

Description: It initializes an empty max-priority queue of intervals.

Parameters:
intervalsPQ → max-priority queue of intervals
maxIntervals ← maximum # of intervals it can contain

Return value: None.

"p_queue.c" 9.2.8 ≡

```
void InitializeIntervalsPQ(INTERVALS_PQ *intervalsPQ, WORD maxIntervals)
  {
  intervalsPQ->interval = (INTERVAL *) calloc(maxIntervals, sizeof(INTERVAL));
  CHECK_MEMORY_ALLOCATION(intervalsPQ->interval);
  intervalsPQ->nIntervals = 0;
  intervalsPQ->maxIntervals = maxIntervals;
  intervalsPQ->index = (WORD *) calloc(maxIntervals, sizeof(WORD));
  CHECK_MEMORY_ALLOCATION(intervalsPQ->index);
  }
```

[p_queue.web]

Function: *InsertIntervalsPQ*( )

Description: It inserts an interval in a max-priority queue of intervals.

Important: We assume that the priority queue is right initialized.

Parameters:
intervalsPQ ↔ max-priority queue of intervals
key ← priority of the interval
index ← index of the interval

Return value: None.

"p_queue.c" 9.2.9 ≡

```
void InsertIntervalsPQ(INTERVALS_PQ *intervalsPQ, WORD key, WORD index)
  {
  WORD i;    // index of the interval currently in exam
  INTERVAL tempInterval;    // temporary interval used for swapping

    // ------------------------ Check heap space
  if (intervalsPQ->interval ≡ NULL)
    {
    fprintf(stderr, "InsertPointsPQ():␣Uninitialized␣heap.\n");
    abort( );
    }
  if (intervalsPQ->nIntervals ≥ intervalsPQ->maxIntervals)
    {
    fprintf(stderr, "InsertIntervalsPQ():␣Maximum␣number␣of␣intervals␣exceeded.\n");
    abort( );
    }

    // ------------------------ Add new interval to the heap
  intervalsPQ->interval[intervalsPQ->nIntervals].key = key;
  intervalsPQ->interval[intervalsPQ->nIntervals].index = index;
  intervalsPQ->index[index] = intervalsPQ->nIntervals;
```

```
intervalsPQ→nIntervals++;

  // ———————————— Heapify
i = intervalsPQ→nIntervals;
while ((i > 1) ∧ (intervalsPQ→interval[i / 2 − 1].key < intervalsPQ→interval[i − 1].key))
  {
  tempInterval = intervalsPQ→interval[i / 2 − 1];
  intervalsPQ→interval[i / 2 − 1] = intervalsPQ→interval[i − 1];
  intervalsPQ→index[intervalsPQ→interval[i / 2 − 1].index] = i / 2 − 1;
  intervalsPQ→interval[i − 1] = tempInterval;
  intervalsPQ→index[intervalsPQ→interval[i − 1].index] = i − 1;
  i /= 2;
  }
}
```

[p_queue.web]

Function: *IncreaseKeyIntervalsPQ*( )

Description: It increases the key of an interval in a max-priority queue of intervals.

Important: We assume that the queue contains the given index.

Parameters:

intervalsPQ	↔	max-priority queue of intervals
index	←	index of the interval

Return value: The value of the new key.

"p_queue.c" 9.2.10 ≡

```
WORD IncreaseKeyIntervalsPQ(INTERVALS_PQ *intervalsPQ, WORD index)
  {
  WORD i;    // index of the interval currently in exam
  INTERVAL tempInterval;    // temporary interval used for swapping

    // ———————————— Check heap
  if (intervalsPQ→interval ≡ NULL)
    {
    fprintf(stderr, "IncreaseKeyIntervalsPQ():␣Uninitialized␣heap.\n");
    abort( );
    }

    // ———————————— Increase the key
  intervalsPQ→interval[intervalsPQ→index[index]].key++;

    // ———————————— Heapify
  i = intervalsPQ→index[index] + 1;
  while ((i > 1) ∧ (intervalsPQ→interval[i / 2 − 1].key < intervalsPQ→interval[i − 1].key))
    {
    tempInterval = intervalsPQ→interval[i / 2 − 1];
    intervalsPQ→interval[i / 2 − 1] = intervalsPQ→interval[i − 1];
    intervalsPQ→index[intervalsPQ→interval[i / 2 − 1].index] = i / 2 − 1;
    intervalsPQ→interval[i − 1] = tempInterval;
    intervalsPQ→index[intervalsPQ→interval[i − 1].index] = i − 1;
    i /= 2;
    }
```

```
    // ---------------------------- Return the value of new key
    return intervalsPQ->interval[intervalsPQ->index[index]].key;
    }
```

[p_queue.web]

Function: *DecreaseKeyIntervalsPQ*( )

Description: It decreases the key of an interval in a max-priority queue of intervals.

Important: We assume that the queue contains the given index.

Parameters:

intervalsPQ	↔	max-priority queue of intervals
index	←	index of the interval

Return value: The value of the new key.

"p_queue.c" 9.2.11 ≡

```
WORD DecreaseKeyIntervalsPQ(INTERVALS_PQ *intervalsPQ, WORD index)
  {
  WORD i, j;    // indices of points currently examined
  INTERVAL tempInterval;    // temporary interval used for swapping

    // ---------------------------- Check heap
  if (intervalsPQ->interval == NULL)
    {
    fprintf(stderr, "DecreaseKeyIntervalsPQ():␣Uninitialized␣heap.\n");
    abort();
    }

    // ---------------------------- Decrease the key
  intervalsPQ->interval[intervalsPQ->index[index]].key--;

    // ---------------------------- Heapify
  i = intervalsPQ->index[index] + 1;
  while (i <= intervalsPQ->nIntervals / 2)
    {
    j = 2 * i;
    if ((j < intervalsPQ->nIntervals) && (intervalsPQ->interval[j - 1].key <
          intervalsPQ->interval[j].key))
      j++;

    if (intervalsPQ->interval[i - 1].key >= intervalsPQ->interval[j - 1].key)
      break;

    tempInterval = intervalsPQ->interval[i - 1];
    intervalsPQ->interval[i - 1] = intervalsPQ->interval[j - 1];
    intervalsPQ->index[intervalsPQ->interval[i - 1].index] = i - 1;
    intervalsPQ->interval[j - 1] = tempInterval;
    intervalsPQ->index[intervalsPQ->interval[j - 1].index] = j - 1;
    i = j;
    }

    // ---------------------------- Return the value of new key
  return intervalsPQ->interval[intervalsPQ->index[index]].key;
  }
```

[p_queue.web]

Function: *GetMaxKeyIntervalsPQ*( )

Description: It returns the maximum key over all intervals stored in a max-priority queue of intervals.

Important: We assume that the max-priority queue is not empty.

Parameters:
intervalsPQ ← max-priority queue of intervals

Return value: The maximum key.

"p_queue.c" 9.2.12 ≡

```
WORD GetMaxKeyIntervalsPQ(const INTERVALS_PQ *intervalsPQ)
{
  // ------------------------------- Check heap
  if (intervalsPQ→interval ≡ NULL)
  {
    fprintf(stderr, "GetMaxKeyIntervalsPQ():␣Uninitialized␣heap.\n");
    abort( );
  }  // ------------------------------- Check emptiness
  if (intervalsPQ→nIntervals ≡ 0)
  {
    fprintf(stderr, "GetMaxKeyIntervalsPQ():␣Priority␣queue␣is␣empty.\n");
    abort( );
  }
  // ------------------------------- Return maximum key
  return intervalsPQ→interval[0].key;
}
```

[p_queue.web]

Function: *DisposeIntervalsPQ*( )

Description: It disposes a max-priority queue of intervals.

Parameters:
intervalsPQ ↔ max-priority queue of intervals

Return value: None.

"p_queue.c" 9.2.13 ≡

```
void DisposeIntervalsPQ(INTERVALS_PQ *intervalsPQ)
{
  free(intervalsPQ→interval);
  intervalsPQ→interval = (INTERVAL *) NULL;
}
```

[p_queue.web]

10 Parameters

[parameters_h.web]

This section contains the definition of the structure containing all the parameters accepted by this program.

"p_queue.c" 10 ≡
@O parameters.h

[parameters_h.web]

Guard

"parameters.h" 10.1 ≡

```
#ifndef _AST_PARAMETERS_H_
#define _AST_PARAMETERS_H_
```

[parameters_h.web]

Includes

"parameters.h" 10.2 ≡

```
#include "ast_node.h"
#include "basic_defs.h"
```

[parameters_h.web]

Constants

"parameters.h" 10.3 ≡

```
    // ———————————————— Default parameters of AST construction
#define DEFAULT_LOG_ROOT_BRANCH 16
#define DEFAULT_C 1.000
#define DEFAULT_MAX_SPECIAL_LEAF 7
#define DEFAULT_DISABLE_WINDOWING FALSE

    // ———————————————— Default parameters of test program
#define DEFAULT_PRINT_ENTRIES FALSE
#define DEFAULT_PRINT_ROUTING_TABLE FALSE
#define DEFAULT_VERIFY_CORRECTNESS FALSE
#define DEFAULT_INLINE_SEARCH FALSE
#define DEFAULT_N_REPEAT_SEARCH 1

    // ———————————————— Default routing and traffic files
#define DEFAULT_ROUTING_FILE (char *) NULL
#define DEFAULT_TRAFFIC_FILE (char *) NULL

    // ———————————————— Minimum parameters of AST construction
#define MIN_LOG_ROOT_BRANCH 0
```

```
#define MIN_C 0.0
#define MIN_MAX_SPECIAL_LEAF 1

    // ———————————— Minimum parameters of test program
#define MIN_N_REPEAT_SEARCH 1

    // ———————————— Maximum parameters of AST construction
#define MAX_MAX_SPECIAL_LEAF 31
```

[parameters_h.web]

Types

"parameters.h" 10.4 ≡

```
typedef struct PARAMETERS
{
  // ———————————— Parameters of AST construction
  int logRootBranch;   // log2(branching factor at root)
  float c[MAX_AST_LEVELS];   // c at each level
  int maxSpecialLeaf;   // maximum # of nodes in a special leaf
  BOOL disableWindowing;   /* disable the windowing? (for more details about the windowing see the
       fields of a internal node on page 20) */

  // ———————————— Parameters of test program
  BOOL printEntries;   // print routing table entries?
  BOOL printRoutingTable;   // print built routing table?
  BOOL verifyCorrectness;   // verify correctness of Find( ) function?
  BOOL inlineSearch;   // perform also inline search?
  long nRepeatSearch;   /* # of times to repeat the search of the IP addresses in a test experiment */

  // ———————————— Routing and traffic files
  /*@null@*/ char *routing_file;   // routing file
  /*@null@*/ char *traffic_file;   // traffic file
} PARAMETERS;
```

[parameters_h.web]

Guard

"parameters.h" 10.5 ≡

```
#endif   // _AST_PARAMETERS_H_
```

[parameters_h.web]

11 Building the AST

[build_ast_h.web]

To build an AST we call *BuildAST*( ) function (page 68) which itself calls *BuildSubAST*( ) function (page 69) that recursively builds a sub-AST containing the set of points (extremes of segments associated to prefixes) passed as a parameter. These points are contained in the *base vector*: each call of *BuildSubAST*( ) receives in a parameter the position of the *first* prefix in the base vector to put in the new sub-AST and the number n of prefixes to consider. In other words each call of *BuildSubAST*( ), builds a sub-AST that covers the *base vector* from position *first* to $first + n - 1$. The extremes of the local universe considered in each call of *BuildSubAST*( ) are *left* and *right*.

The algorithm to build a sub-AST is:

1. **if** $n \equiv 0$ **then** the sub-AST is composed by an *empty leaf* (*covered or uncovered*);
2. **else if** $n \equiv 1$ **then** the sub-AST is composed by a *full leaf*;
3. **else if** $1 \le n \le parameters \rightarrow maxSpecialLeaf$ **then** the sub-AST is composed by a *special leaf*;
4. **else** the sub-AST is rooted on an *internal node* and its children are recursively built in the following way:
   (a) compute (using *ComputeAnchorStepWindowWidth*( ) function, page 72) the *anchor*, the *step*, *leftWindowWidth* and *rightWindowWidth* (*leftWindowWidth* is the number of nodes, on the left of the anchor, after which there are only uncovered empty leaves, which are not stored and should not be inspected during the search, *rightWindowWidth* is the same, but on the right side);
   (b) create a new *anchor* point in the anchor table;
   (c) place the internal node in the AST;
   (d) consider each sub-interval (without exceeding *rightWindowWidth*), get the prefixes inside it, and recursively call *BuildSubAST*( ).

ComputeAnchorStepWindowWidth( ) function (page 72) computes *anchor*, *step*, *leftWindowWidth* and *rightWindowWidth* in the following way:

1. start with the largest possible *step* (equal to the length of the interval);
2. using the current *step*, compute the minimum R (= # of empty buckets / # of full buckets) trying all the possible *anchor* points (this can be done efficiently in $O(n \log n)$ using two priority queues, see *ComputeMinRAndMinM*( ) function, page 74);
3. **if** the minimum R is not below the constant C **then** split into two the *step* and **goto** 2, **else goto** 4;
4. perform a "shift" (choose an *anchor* within all the possible ones) to obtain the minimum M (= max number of points in a bucket) for that *step*.

11.1 Building the AST: header

[build_ast_h.web]

"parameters.h" 11.1 ≡

```
@O build_ast.h
```

[build_ast_h.web]

Guard

"build_ast.h" 11.1.1 ≡

```
#ifndef _AST_BUILD_AST_H_
#define _AST_BUILD_AST_H_
```

[build_ast_h.web]

Includes

"build_ast.h" 11.1.2 ≡

```
#include <stdio.h>

#include "anchor.h"
#include "ast_node.h"
#include "base.h"
#include "basic_defs.h"
#include "parameters.h"
```

[build_ast_h.web]

Prototypes

"build_ast.h" 11.1.3 ≡

```
WORD BuildAST(const BASE base[ ], WORD nBases, AST_NODE ast[ ],
    ANCHOR anchorTable[ ], WORD *nAnchors, const PARAMETERS *parameters);
```

[build_ast_h.web]

Guard

"build_ast.h" 11.1.4 ≡

```
#endif    // _AST_BUILD_AST_H_
```

[build_ast_h.web]

11.2 Building the AST: implementation

[build_ast.web]

"build_ast.h" 11.2 ≡
```
@O build_ast.c
```

[build_ast.web]

Includes

"build_ast.c" 11.2.1 ≡

```
#include <stdio.h>
#include <stdlib.h>

#include "anchor.h"
#include "ast_node.h"
#include "base.h"
#include "basic_defs.h"
#include "build_ast.h"
#include "bucket.h"
#include "next_hop.h"
#include "p_queue.h"
#include "parameters.h"
```

[build_ast.web]

Internal prototypes

"build_ast.c" 11.2.2 ≡
```
static void BuildSubAST(const BASE base[ ], WORD first, WORD n, IP_ADDRESS left,
    IP_ADDRESS right, AST_NODE ast[ ], WORD rootIndex, WORD *nextFree, int astLevel,
    ANCHOR anchorTable[ ], WORD *nAnchors, const PARAMETERS *parameters);
static void ComputeAnchorStepWindowWidth(const BASE base[ ], WORD first, WORD n,
    IP_ADDRESS left, IP_ADDRESS right, int astLevel, const PARAMETERS *parameters,
    IP_ADDRESS *anchor, IP_ADDRESS *step, int *logStep, WORD *leftWindowWidth,
    WORD *nNodesLeft, WORD *rightWindowWidth, WORD *nNodesRight);
static void ComputeMinRAndMinM(const BASE base[ ], WORD first, WORD n,
    IP_ADDRESS left, IP_ADDRESS right, IP_ADDRESS step,
    const PARAMETERS *parameters, double *minR, WORD *minM,
    IP_ADDRESS *anchor, WORD *leftWindowWidth, WORD *nNodesLeft,
    WORD *rightWindowWidth, WORD *nNodesRight);
```

[build_ast.web]

Functions

Function: *BuildAST*( )

Description: It builds an AST.

Important: The points in the *base*[ ] vector should be sorted.

Important: In this function we assume that the size of allocated memory for *ast*[ ] array is at least *MAX_AST_NODES* and the size of allocated memory for *anchorTable*[ ] is at least *MAX_ANCHORS*.

Parameters:

base	←	*base vector*
nBases	←	# of bases
ast	↔	AST represented as an array
anchorTable	↔	anchor table (for details see sub-section 5.3 on page 16)
nAnchors	↔	# of anchors in the anchor table
parameters	←	parameters

Return value: Size of the AST (i.e. # of nodes in the AST).

"build_ast.c" 11.2.3 ≡

```
WORD BuildAST(const BASE base[], WORD nBases, AST_NODE ast[],
    ANCHOR anchorTable[], WORD *nAnchors, const PARAMETERS *parameters)
{
WORD nASTNodes;    // # of AST nodes
  // ———————————— Recursively build the AST
nASTNodes = 1;
*nAnchors = 0;
BuildSubAST(base, 1, nBases - 1, 0, MAX_IP_ADDRESS, ast, 0, &nASTNodes, 0, anchorTable,
    nAnchors, parameters);
  // ———————————— Return the # of nodes in the AST
return nASTNodes;
}
```

[build_ast.web]

Internal functions

Internal function: *BuildSubAST*( )

Description: It recursively builds a sub-AST.

Important: In this function we assume that the size of allocated memory for *ast*[ ] array is at least *MAX_AST_NODES* and the size of allocated memory for *anchorTable*[ ] is at least *MAX_ANCHORS*.

Note: Each call of this function builds a sub-AST that covers the *base vector* from position *first* to $first + n - 1$.

Note: The extremes of the geometric interval, which contains the extremes of segments from *base*[*first*].*point* to *base*[$first + n - 1$].*point*, are *left* and *right*.

Note: When $n \equiv 0$, *first* is the index of the point after the current geometric interval and $first - 1$ (if exists) is the index of the point before it.

Note: The leftmost child of an internal node will be placed in *ast*[*nextFree*].

Important: The first call of this function should be something like

BuildSubAST(*base*, 1, *nBases* − 1, 0, *MAX_IP_ADDRESS*, *ast*, 0, &*nextFree*, 0, *anchorTable*, &*nAnchors*, *parameters*);

where before this call the value of *nextFree* should be 1 and after it will be equal to the # of allocated positions in the AST array; instead the value of nAnchors before first call should be 0.

Parameters:

base	←	*base vector*
first	←	first position of *base*[ ] vector to cover
n	←	# of points in *base*[ ] to consider
left	←	left extreme of the interval
right	←	right extreme of the interval
ast	↔	AST represented as an array (for details see sub-section 5.4 on page 19)
rootIndex	←	index in the AST array for storing the root of the sub-AST
nextFree	↔	first position in the AST array that has not yet been reserved
astLevel	←	level in the AST of the internal node
anchorTable	↔	anchor table (for details see sub-section 5.3 on page 16)
nAnchors	↔	# of anchors in the anchor table
parameters	←	parameters

Return value: None.

"build_ast.c" 11.2.4 ≡

```
static void BuildSubAST(const BASE base[], WORD first, WORD n, IP_ADDRESS left,
        IP_ADDRESS right, AST_NODE ast[], WORD rootIndex, WORD *nextFree, int astLevel,
        ANCHOR anchorTable[], WORD *nAnchors, const PARAMETERS *parameters)
  {
  WORD firstChildIndex;   // index in the AST array where storing the 1st child
  int i;   // index of scan the children
  IP_ADDRESS anchor;   // anchor point
  IP_ADDRESS step;   // length of a regular sub-interval (bucket)
  int logStep;   // log2(step)
  WORD leftWindowWidth;   /* # of nodes, on the left of the anchor, after which there are only
        uncovered empty leaves */
  WORD nNodesLeft;   /* # of nodes on the left of the anchor to be stored in the AST */
  WORD rightWindowWidth;   /* # of nodes, on the right of the anchor, after which there are only
        uncovered empty leaves */
  WORD nNodesRight;   /* # of nodes on the right of the anchor to be stored in the AST */
  WORD childFirst;   // first position of base[ ] to be covered by a child
```

```
WORD childN;    // # of points a child should consider
IP_ADDRESS subLeft;    // left extreme of a sub-interval (bucket)
IP_ADDRESS subRight;    // right extreme of a sub-interval (bucket)

  /* ———————— If the interval contains no points create an empty leaf */
if (n ≡ 0)
  {
  if ((first ≠ 0) ∧ (base[first − 1].nextHopIndex ≠ DEFAULT_NEXT_HOP_INDEX))
    ast[rootIndex] = SET_COVERED_EMPTY_LEAF |
        SET_NEXT_HOP_INDEX(base[first − 1].nextHopIndex);
  else
    ast[rootIndex] = SET_UNCOVERED_EMPTY_LEAF;
  }

  /* ———————— Else if the current node is a full leaf ... */
else if (n ≡ 1)
  ast[rootIndex] = SET_FULL_LEAF | SET_BASE_INDEX(first);

  // ———————— Else if it is a special leaf ...
else if (n ≤ parameters→maxSpecialLeaf)    /* ∧ (n ≥ 2) */
  ast[rootIndex] = SET_SPECIAL_LEAF |
      SET_N_LEFT_BASES(n / 2) | SET_N_RIGHT_BASES(n − 1 − (n / 2)) |
      SET_BASE_INDEX(first + n / 2);

  // ———————— Else it is an internal node ...
else    /* (n > parameters→maxSpecialLeaf) */
  {
    // ———————— Compute anchor step and window width
  ComputeAnchorStepWindowWidth(base, first, n, left, right, astLevel, parameters, &anchor,
      &step, &logStep, &leftWindowWidth, &nNodesLeft, &rightWindowWidth, &nNodesRight);

    // ———————— Allocate memory in the AST array
  firstChildIndex = *nextFree;
  *nextFree += nNodesLeft + nNodesRight;
  if (*nextFree > MAX_AST_NODES)
    {
    fprintf(stderr, "BuildSubAST():␣maximum␣allowed␣AST␣size␣exceeded.\n");
    abort();
    }

    // ———————— Store the anchor
  if (*nAnchors ≥ MAX_ANCHORS)
    {
    fprintf(stderr, "BuildSubAST():␣maximum␣allowed␣number␣of␣anchors␣exceeded.\n");
    abort();
    }
  anchorTable[*nAnchors].point = anchor;
  anchorTable[*nAnchors].fields = SET_LOG_STEP(logStep) |
      SET_AST_INDEX(firstChildIndex + nNodesLeft);
  (*nAnchors)++;

    // ———————— Fill in the internal node
  ast[rootIndex] = SET_INTERNAL_NODE | SET_LEFT_WINDOW_WIDTH(leftWindowWidth) |
      SET_RIGHT_WINDOW_WIDTH(rightWindowWidth) | SET_ANCHOR_INDEX(*nAnchors − 1);

    // ———————— Adjust extremes
  if (¬parameters→disableWindowing)
```

```
      {
      if ((leftWindowWidth ≠ 0) ∧ (leftWindowWidth * step ≠ 0) ∧
            (leftWindowWidth * step < anchor − left))
        left = anchor − leftWindowWidth * step;
      if ((rightWindowWidth ≠ 0) ∧ (rightWindowWidth * step ≠ 0) ∧
            (rightWindowWidth * step < right − anchor + 1))
        right = anchor − 1 + rightWindowWidth * step;
      }
      // ——————————————— Work recursively on the children
    i = 0;
    childFirst = first;
    WHILE_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, anchor, step,
          subLeft, subRight)
      {
      childN = 0;
      while ((childFirst + childN < first + n) ∧ (base[childFirst + childN].point ≤ subRight))
        childN++;
      BuildSubAST(base, childFirst, childN, subLeft, subRight, ast, firstChildIndex + i, nextFree,
          astLevel + 1, anchorTable, nAnchors, parameters);
      i++;
      childFirst += childN;
      } END_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, anchor, step,
          subLeft, subRight);
    }
  }
```

[build_ast.web]

Internal function: *ComputeAnchorStepWindowWidth*( )

Description: It computes the *step* for the root of the sub-AST rooted on the internal node which called this function. It computes also *leftWindowWidth*, *nNodesLeft*, *rightWindowWidth* and *nNodesRight* (see description below).

Important: We assume that $n > parameters \rightarrow maxSpecialLeaf \wedge n > 1$ (this implies also that $right - left + 1 > parameters \rightarrow maxSpecialLeaf \wedge right - left + 1 > 1$. Where n is the # of points to place in the current interval.

Note: *leftWindowWidth* is the # of nodes on the left of the anchor after which there are only uncovered empty leaves (which are not stored and should not been inspected during the search). If the effective value of *leftWindowWidth* exceeds *MAX_WINDOW_WIDTH*, we set *leftWindowWidth* to 0 (this is a way to disable it). *nNodesLeft* is the final # of nodes on the left of the anchor-leaf that should be stored in the AST. So if $leftWindowWidth \leq MAX_WINDOW_WIDTH$ then $nNodesLeft = leftWindowWidth$, else *nNodesLeft* is the total # of sub-intervals before the one associated to anchor-leaf.
rightWindowWidth and *nNodesRight* are respectively the same of *leftWindowWidth* and *nNodesLeft*, but on the right side of the anchor.

Parameters:

base	←	*base vector*
first	←	first position of *base*[ ] vector to cover
n	←	# of points in *base*[ ] to consider
left	←	left extreme of the interval
right	←	right extreme of the interval
astLevel	←	level in the AST of the internal node
parameters	←	parameters
anchor	→	point chosen as anchor
step	→	length of a regular sub-interval
logStep	→	$\log_2(step)$
leftWindowWidth	→	# of nodes, on the left of the anchor-leaf, after which there are only uncovered empty leaves
nNodesLeft	→	# of nodes on the left of the anchor-leaf to be stored in the AST
rightWindowWidth	→	# of nodes, on the right of the anchor-leaf, after which there are only uncovered empty leaves
nNodesRight	→	# of nodes on the right of the anchor-leaf to be stored in the AST

Return value: None.

"build_ast.c" 11.2.5 ≡

```
static void ComputeAnchorStepWindowWidth(const BASE base[], WORD first, WORD n,
        IP_ADDRESS left, IP_ADDRESS right, int astLevel, const PARAMETERS *parameters,
        IP_ADDRESS *anchor, IP_ADDRESS *step, int *logStep, WORD *leftWindowWidth,
        WORD *nNodesLeft, WORD *rightWindowWidth, WORD *nNodesRight)
    {
    BOOL useFixedBranch;    /* TRUE if it is the root and parameters->logRootBranch > 0 */
    int b;    // log2(branching factor)
    IP_ADDRESS oldAnchor;    // value of *anchor at the prev. iteration
    IP_ADDRESS oldStep;    // value of *step at the prev. iteration
    int oldLogStep;    // value of *logStep at the prev. iteration
    WORD oldLeftWindowWidth;    // value of *leftWindowWidth at prev. iteration
    WORD oldNNodesLeft;    // value of *nNodesLeft at the prev. iteration
    WORD oldRightWindowWidth;    // value of *rightWindowWidth at prev. iteration
    WORD oldNNodesRight;    // value of *nNodesRight at the prev. iteration
    double minR;    // minimum R (= nEmpties/nFulls)
```

```
WORD minM;    // minimum M (= max number of points in a bucket)
  // ———————————— Shall we use a fixed branch for root?
  /* Note: we use a large branching factor at the root. */
if ((left ≡ 0) ∧ (right ≡ MAX_IP_ADDRESS) ∧ (parameters→logRootBranch > 0))
  useFixedBranch = TRUE;
else
  useFixedBranch = FALSE;
  // ———————————— Initialize parameters
b = −1;
*anchor = 0;
*step = *logStep = 0;
*leftWindowWidth = *nNodesLeft = 0;
*rightWindowWidth = *nNodesRight = 0;
do
  {
    // ———————————— Save old parameters
  b++;
  oldAnchor = *anchor;
  oldStep = *step;
  oldLogStep = *logStep;
  oldLeftWindowWidth = *leftWindowWidth;
  oldNNodesLeft = *nNodesLeft;
  oldRightWindowWidth = *rightWindowWidth;
  oldNNodesRight = *nNodesRight;
    // ———————————— Compute the step
  if (useFixedBranch)
    {
    COMPUTE_STEP(left, right, parameters→logRootBranch, *step, *logStep);
    }
  else
    {
    COMPUTE_STEP(left, right, b, *step, *logStep);
    }
    // ———————————— Compute minR and minM
  ComputeMinRAndMinM(base, first, n, left, right, *step, parameters, &minR, &minM,
      anchor, leftWindowWidth, nNodesLeft, rightWindowWidth, nNodesRight);
  }
while ((*logStep ≡ IP_ADDRESS_SIZE) ∨
    (¬useFixedBranch ∧ (minR ≤ parameters→c[astLevel]) ∧ (minM > 1)));
  /* Note: in the condition above here we can put again
    "∧ (mimM > parameters→maxSpecialLeaf)" */
  // ———————————— Step back to values at prev. iteration
  /* Note: we step back if more than one iteration of the do-while was performed (b > 0) and we do
    not exit because minM ≡ 1. */
if ((oldLogStep < IP_ADDRESS_SIZE) ∧ (b > 0) ∧ (¬useFixedBranch) ∧
      (minR > parameters→c[astLevel]))
  {
  *anchor = oldAnchor;
  *step = oldStep;
```

```
    *logStep = oldLogStep;
    *leftWindowWidth = oldLeftWindowWidth;
    *nNodesLeft = oldNNodesLeft;
    *rightWindowWidth = oldRightWindowWidth;
    *nNodesRight = oldNNodesRight;
    }
}
```

[build_ast.web]

Internal function: *ComputeMinRAndMinM*( )

Description: Given an interval and a step it computes the minimum R (= $nEmpties/nFulls$) that can be obtained, and the minimum M (= maximum number of points in a bucket) that can be obtained.

Important: We assume that $n > parameters \rightarrow maxSpecialLeaf \wedge n > 1$ (this implies also that $right - left + 1 > parameters \rightarrow maxSpecialLeaf \wedge right - left + 1 > 1$. Where n is the # of points to place in the current internal.

Note: Initially we set the anchor equal to *left* and the length of the first bucket is equal to *step*. When we perform a shift a new bucket is created before that one, which length is less than *step*. To simplify the treatment, at the beginning we act as if the bucket that will be created at second step just exists, but it is of length 0 and contains no points.

Note: $step \equiv 0$ means a step of $2^{IP_ADDRESS_SIZE}$.

Parameters:

base	←	*base vector*
first	←	first position of *base*[ ] vector to cover
n	←	# of points in *base*[ ] to consider
left	←	left extreme of the interval
right	←	right extreme of the interval
step	←	length of a regular sub-interval
parameters	←	parameters
minR	→	minimum R (= $nEmpties/nFulls$)
minM	→	minimum M (= maximum number of points in a bucket)
anchor	→	anchor when M has the minimum value
leftWindowWidth	→	when M is minimum, # of nodes, on the left of the anchor-leaf, after which there are only uncovered empty leaves
nNodesLeft	→	when M is minimum, # of nodes, on the left of the anchor-leaf, to be stored in the AST
rightWindowWidth	→	same as *leftWindowWidth* but on the right
nNodesRight	→	same as *nNodesRight* but on the right

Return value: None.

"build_ast.c" 11.2.6 ≡

```
  static void ComputeMinRAndMinM(const BASE base[ ], WORD first, WORD n,
        IP_ADDRESS left, IP_ADDRESS right, IP_ADDRESS step,
        const PARAMETERS *parameters, double *minR, WORD *minM,
        IP_ADDRESS *anchor, WORD *leftWindowWidth, WORD *nNodesLeft,
        WORD *rightWindowWidth, WORD *nNodesRight)
    {
    POINTS_PQ pointsPQ;    // priority queue of points inside current interval
    WORD i;    // index to scan points and intervals
```

```
IP_ADDRESS priority;   // priority of point being examined
WORD nBuckets;   // # of sub-intervals (buckets)
INTERVALS_PQ intervalsPQ;   // priority queue of sub-intervals
BOOL leftCovered;   // if the first bucket is empty, is it covered empty?
BOOL rightCovered;   // if the last bucket is empty, is it covered empty?
WORD nFulls;   // # of full buckets
WORD firstFull;   // index of first full bucket
WORD lastFull;   // index of last full bucket
WORD childFirst;   // first position of base[ ] to be covered by a child
WORD childN;   // # of points a child should consider
IP_ADDRESS subLeft;   // left extreme of a sub-interval (bucket)
IP_ADDRESS subRight;   // right extreme of a sub-interval (bucket)
IP_ADDRESS currAnchor;   // current position of the anchor
WORD anchorInterval;   // index of interval containing currAnchor
BOOL left0length;   // is first interval on the left of length 0?
BOOL right0length;   // is last interval on the right of length 0?
WORD nEmpties;   // # of empty buckets
WORD nEmptiesLeft;   // # of empty buckets saved by left windowing
WORD nEmptiesRight;   // # of empty buckets saved by right windowing
double r;   // R obtained with currAnchor
WORD m;   // M obtained with currAnchor
WORD newKey;   // new key of examined interval

  // ———————————— Build the priority queue of points
InitializePointsPQ(&pointsPQ, n);

for (i = 0; i < n; i++)
  {
  priority = (step ≡ 0) ? base[first + i].point − left : (base[first + i].point − left) % step;
  InsertPointsPQ(&pointsPQ, priority, first + i);
  }
  // ———————————— Compute the # of sub-intervals
  /* Note: we count also the bucket on the left that initially is 0-length. */
if (step ≡ 0)
  nBuckets = 2;
else if ((left ≡ 0) ∧ (right ≡ MAX_IP_ADDRESS))
  nBuckets = ((1 << (IP_ADDRESS_SIZE − 1)) / (step >> 1)) + 1;
else
  nBuckets = 1 + ((right − left + 1) / step) + (((right − left + 1) % step) > 0 ? 1 : 0);
  // ———————————— Initialize priority queue of buckets
  /* Note: Initially the length of the first sub-interval (which index is 0) is 0 (because anchor ≡ left),
     and thus it contains no points. */
InitializeIntervalsPQ(&intervalsPQ, nBuckets);
InsertIntervalsPQ(&intervalsPQ, 0, 0);

  // ———————————— Set leftCovered and rightCovered
  /* Note: if leftCovered ≡ TRUE it means that there are no uncovered empty buckets on the left
     and thus leftWindowWidth ≡ nNodesLeft (or leftWindowWidth ≡ 0). The same thing holds for
     rightCovered but on the right side. */
leftCovered = (first > 0) ∧ (base[first − 1].nextHopIndex ≠ DEFAULT_NEXT_HOP_INDEX);
rightCovered = (base[first + n − 1].nextHopIndex ≠ DEFAULT_NEXT_HOP_INDEX);

  // ———————————— Initialize "full" counters
nFulls = 0;
```

```
firstFull = lastFull = 0;    // just to prevent a compiler warning
  // ———————————— Build the priority queue of intervals
childFirst = first;
i = 1;
WHILE_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, left, step, subLeft, subRight)
  {
  childN = 0;
  while ((childFirst + childN < first + n) ∧ (base[childFirst + childN].point ≤ subRight))
    childN++;
  if (childN > 0)
    {
    if (nFulls ≡ 0)
      firstFull = i;
    nFulls++;
    lastFull = i;
    }
  InsertIntervalsPQ(&intervalsPQ, childN, i);
  i++;
  childFirst += childN;
  } END_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, left, step, subLeft, subRight);
  // ———————————— Set initial anchor position
  /* Note: Initially the anchor is taken equal to left. Instead of setting it exactly to left, it is taken
     in an equivalent point, but in a more central bucket: using this optimization we get more
     advantage of the windowing. */
currAnchor = (step ≡ 0) ? left : left + (nBuckets / 2 − 1) ∗ step;
anchorInterval = nBuckets / 2;
  // ———————————— Set initial 0-length extremes
  /* Note: The first bucket on the left is 0-length only at the beginning when no shift is performed.
     The last bucket on the right will initially is surely not 0-length. */
left0length = TRUE;
right0length = FALSE;
  // ———————————— Initialize minR and minM
  /* Note: The following assignments make minR and minM certainly higher than their final
     values, and thus also ∗anchor, ∗leftWindowWidth, ∗nNodesLeft, etc. will be updated. */
∗minR = (double) nBuckets + 1.0;
∗minM = GetMaxKeyIntervalsPQ(&intervalsPQ) + 1;
  /* Note: The following initialization are only needed to prevent a compiler warning: the final
     values will be assigned in the following loop. */
∗anchor = currAnchor;
∗leftWindowWidth = ∗nNodesLeft = ∗rightWindowWidth = ∗nNodesRight = 0;
  // ———————————— Examine all possible anchors ...
  /* Note: There is at least 1 point to examine. */
do
  {
    // ———————————— Compute # of empty buckets
  if (parameters→disableWindowing)
```

```
    nEmptiesLeft = nEmptiesRight = 0;
  else
    {
    if ((leftCovered) ∨ (anchorInterval − firstFull > MAX_WINDOW_WIDTH))
      nEmptiesLeft = 0;
    else
      {
      nEmptiesLeft = firstFull − (left0length ? 1 : 0);
      if ((anchorInterval ≤ firstFull) ∧ (currAnchor > left))
        nEmptiesLeft−−;
      }
    if ((rightCovered) ∨ (lastFull − anchorInterval + 1 > MAX_WINDOW_WIDTH))
      nEmptiesRight = 0;
    else
      {
      nEmptiesRight = nBuckets − lastFull − 1 − (right0length ? 1 : 0);
      if (anchorInterval ≥ lastFull)
        nEmptiesRight−−;
      }
    }
  nEmpties = nBuckets − nFulls − nEmptiesLeft − nEmptiesRight −
      (left0length ? 1 : 0) − (right0length ? 1 : 0);
    // ———————————— Check minR
  r = (double) nEmpties / (double) nFulls;
  if (r < ∗minR)
    ∗minR = r;
    // ———————————— Check minM
  m = GetMaxKeyIntervalsPQ(&intervalsPQ);
  if (m < ∗minM)
    {
    ∗minM = m;
    ∗anchor = currAnchor;
    if ((parameters→disableWindowing) ∨ (leftCovered) ∨
          (anchorInterval − firstFull > MAX_WINDOW_WIDTH))
      {
      ∗leftWindowWidth = 0;
      ∗nNodesLeft = anchorInterval − (left0length ? 1 : 0);
      }
    else
      {
      ∗leftWindowWidth = anchorInterval − firstFull;
      if ((∗leftWindowWidth ≡ 0) ∧ (currAnchor > left))
        ∗leftWindowWidth = 1;
      ∗nNodesLeft = ∗leftWindowWidth;
      }
    if ((parameters→disableWindowing) ∨ (rightCovered) ∨
          (lastFull − anchorInterval + 1 > MAX_WINDOW_WIDTH))
      {
      ∗rightWindowWidth = 0;
      ∗nNodesRight = nBuckets − anchorInterval − (right0length ? 1 : 0);
      }
```

```
    else
      {
      *rightWindowWidth = lastFull − anchorInterval + 1;
      if (*rightWindowWidth ≡ 0)
        *rightWindowWidth = 1;
      *nNodesRight = *rightWindowWidth;
      }
    }
    // ———————————— Shift the anchor ...
  priority = GetMinKeyPointsPQ(&pointsPQ);
  while (¬IsEmptyPointsPQ(&pointsPQ) ∧ (GetMinKeyPointsPQ(&pointsPQ) ≡ priority))
    {
      // ———————————— Get new anchor
    currAnchor = base[ExtractPointsPQ(&pointsPQ)].point;
    anchorInterval = (step ≡ 0) ? 1 : (currAnchor − left) / step + 1;

      // ———————————— Update # of points per bucket
    newKey = IncreaseKeyIntervalsPQ(&intervalsPQ, anchorInterval − 1);
    if (newKey ≡ 1)
      {
      nFulls++;
      if (firstFull ≡ anchorInterval)
        firstFull = anchorInterval − 1;
      if ((anchorInterval > 1) ∧ (lastFull ≡ anchorInterval − 2))
        lastFull = anchorInterval − 1;
      }

    newKey = DecreaseKeyIntervalsPQ(&intervalsPQ, anchorInterval);
    if (newKey ≡ 0)
      {
      nFulls−−;
      if (firstFull ≡ anchorInterval)
        firstFull−−;
      if (lastFull ≡ anchorInterval)
        lastFull++;
      }
    }
  if (¬IsEmptyPointsPQ(&pointsPQ))
    {
      // ———————————— Get next anchor
      /* Note: The following optimization can be used only because we store the points chosen as
        anchor in the anchor table (defined in sub-section 5.3 on page 16). Using this optimization
        we get more advantage of the windowing. */
    currAnchor = (step ≡ 0) ? GetMinKeyPointsPQ(&pointsPQ) :
        left + (nBuckets / 2 − 1) * step + GetMinKeyPointsPQ(&pointsPQ);
    anchorInterval = nBuckets / 2;

      // ———————————— Update 0-length extremes
    left0length = FALSE;

      /* Note: When right0length becomes TRUE it remains TRUE up to the end. */
    if (¬right0length ∧ (step ≠ 0) ∧ ¬((left ≡ 0) ∧ (right ≡ MAX_IP_ADDRESS)) ∧
          ((right − left + 1) % step > 0) ∧ (step < right − left + 1) ∧
```

```
            (((right − currAnchor + 1) % step ≡ 0) ∨
            (nBuckets − anchorInterval > (right − currAnchor + 1) / step + 1)))
        right0length = TRUE;
      }
    }
  while (¬IsEmptyPointsPQ(&pointsPQ));
    // ——————————————— Dispose priority queues
  DisposePointsPQ(&pointsPQ);
  DisposeIntervalsPQ(&intervalsPQ);
  }
```

[build_ast.web]

12 Building the routing table

[build_rt_h.web]

This section contains the routines to build, print and dispose a routing table based on ASTs.

The entries (prefixes) of the routing table are first stored in an initial simple array (of type **ENTRY**) allocated statically (in function *main*( ), page 123). We build the routing table (*BuildRoutingTable*( ) function, on page 83) in the following way:

1. we sort the input set of prefixes and remove duplicates using *SortAndRemoveDuplicatesEntries*( ) function (page 37);
2. we build the next-hop table using *BuildNextHopTable*( ) function (page 87);
3. we build the base vector using *BuildBaseVector*( ) function (page 88);
4. we allocate a temporary AST and a temporary anchor table which sizes are larger than the ones that will be required at the end of the construction of the routing table;
5. we build the AST using *BuildAST*( ) function (page 68): the algorithm to build the AST is described at the beginning of section 11 on page 65;
6. at this point the proper sizes of the AST and of the anchor table are known, so we reallocate them;
7. finally, we fill in the routing table structure.

12.1 Building the routing table: header

[build_rt_h.web]

"build_ast.c" 12.1 ≡
    @O build_rt.h

[build_rt_h.web]

Guard

"build_rt.h" 12.1.1 ≡

```
#ifndef _AST_BUILD_RT_H_
#define _AST_BUILD_RT_H_
```

[build_rt_h.web]

Includes

"build_rt.h" 12.1.2 ≡

```
#include <stdio.h>

#include "basic_defs.h"
#include "entry.h"
#include "parameters.h"
#include "routing_table.h"
```

[build_rt_h.web]

Prototypes

"build_rt.h" 12.1.3 ≡

```
ROUTING_TABLE BuildRoutingTable(FILE *out, ENTRY entry[], WORD *nEntries,
    const PARAMETERS *parameters, BOOL verbose);
void PrintRoutingTable(FILE *out, const ROUTING_TABLE t);
void DisposeRoutingTable(ROUTING_TABLE t);
```

[build_rt_h.web]

Guard

"build_rt.h" 12.1.4 ≡

```
#endif    // _AST_BUILD_RT_H_
```

[build_rt_h.web]

12.2 Building the routing table: implementation

[build_rt.web]

"build_rt.h" 12.2 ≡
@O build_rt.c

[build_rt.web]

Includes

"build_rt.c" 12.2.1 ≡

```
#include <stdio.h>
#include <stdlib.h>

#include "anchor.h"
#include "ast_node.h"
#include "base.h"
#include "basic_defs.h"
#include "build_ast.h"
#include "build_rt.h"
#include "clock.h"
#include "entry.h"
#include "next_hop.h"
#include "parameters.h"
#include "prefix.h"
#include "routing_table.h"
```

[build_rt.web]

Internal prototypes

"build_rt.c" 12.2.2 ≡

```
static WORD BuildNextHopTable(const ENTRY entry[ ], WORD nEntries,
    NEXT_HOP *nextHop[ ]);
static int CompareNextHops(const NEXT_HOP *i, const NEXT_HOP *j);
static WORD BuildBaseVector(const ENTRY entry[ ], WORD nEntries,
    const NEXT_HOP nextHop[ ], WORD nNextHops, BASE *base[ ]);
static void RecursivelyBuildBaseVector(const ENTRY entry[ ], WORD *nExaminedEntries,
    WORD nEntries, BOOL isIncluded, WORD includerIndex,
    const NEXT_HOP nextHop[ ], WORD nNextHops, BASE base[ ], WORD *nBases);
static int BinarySearch(NEXT_HOP x, const NEXT_HOP v[ ], WORD n);
```

[build_rt.web]

Functions

Function: *BuildRoutingTable*( )

Description: It builds the routing table.

Note: We first allocate a temporary AST and a temporary routing table which sizes are larger than the ones that will be required at the end of the construction of the routing table. Only when their proper size are known we reallocate them and we store them in the routing table.

Note: After the call of this function, *nEntries* will contain the # of unique entries. This is the # of entries to dispose.

Parameters:

out	↔	file of output where to print statistic messages
entry	↔	routing table entries stored in an initial simple array
nEntries	↔	# of entries of the routing table
parameters	←	parameters
verbose	←	if *TRUE* it prints statistic messages on *out*

Return value: The built routing table.

"build_rt.c" 12.2.3 ≡

```
ROUTING_TABLE BuildRoutingTable(FILE *out, ENTRY entry[], WORD *nEntries,
      const PARAMETERS *parameters, BOOL verbose)
  {
  WORD nUniqueEntries;    // # of routing entries after duplicate removal
  NEXT_HOP *nextHop;    // next-hop table
  WORD nNextHops;    // # of next-hops
  BASE *base;    // base vector
  WORD nBases;    // # of base records
  AST_NODE *tempAST;    // temporary AST (bigger than the final AST)
  WORD nASTNodes;    // # of nodes in the AST
  ANCHOR *tempAnchorTable;    // temporary anchor table (bigger than the final one)
  WORD nAnchors;    // # of anchors in the anchor table
  AST_NODE *ast;    // final AST
  ANCHOR *anchorTable;    // final anchor table
  ROUTING_TABLE table;    // the complete data structure

    // ———————————————— Sort the entries and remove duplicates
  ClockOn( );
  nUniqueEntries = SortAndRemoveDuplicatesEntries(entry, *nEntries);
  ClockOff( );
  if (verbose)
    {
    fprintf(out, "\nSorting:␣%.2f␣sec", GetTime( ));
    if (nUniqueEntries ≠ *nEntries)
      {
      fprintf(out, "␣␣(");
      PRINT_WORD(out, nUniqueEntries, 0);
      fprintf(out, "␣unique␣entries)");
      }
    fprintf(out, "\n");
    }
  *nEntries = nUniqueEntries;

    // ———————————————— Build the next-hop table
```

```
ClockOn( );
nNextHops = BuildNextHopTable(entry, nUniqueEntries, &nextHop);
ClockOff( );
if (verbose)
  fprintf(out, "Building␣next-hop␣table:␣%.2f␣sec\n", GetTime( ));
  // ———————————— Build the base vector
ClockOn( );
nBases = BuildBaseVector(entry, *nEntries, nextHop, nNextHops, &base);
ClockOff( );
if (verbose)
  {
  fprintf(out, "Building␣base␣vector:␣%.2f␣sec␣␣(", GetTime( ));
  PRINT_WORD(out, nBases, 0);
  fprintf(out, "␣different␣points)\n");
  }
  /* ———————————— Allocate memory for temporary AST and temporary anchor table */
ClockOn( );
tempAST = (AST_NODE *) calloc(MAX_AST_NODES, sizeof(AST_NODE));
CHECK_MEMORY_ALLOCATION(tempAST);
tempAnchorTable = (ANCHOR *) calloc(MAX_ANCHORS, sizeof(ANCHOR));
CHECK_MEMORY_ALLOCATION(tempAnchorTable);
  // ———————————— Build AST structure
nASTNodes = BuildAST(base, nBases, tempAST, tempAnchorTable, &nAnchors, parameters);
  /* ———————————— Reallocate AST and anchor table on a proper size arrays */
  /* Note: at this point we know how much memory is needed. */
ast = (AST_NODE *) realloc(tempAST, nASTNodes * sizeof(AST_NODE));
CHECK_MEMORY_ALLOCATION(ast);
anchorTable = (ANCHOR *) realloc(tempAnchorTable, nAnchors * sizeof(ANCHOR));
if (nAnchors > 0)
  CHECK_MEMORY_ALLOCATION(anchorTable);
  // ———————————— Allocate routing table structure
table = (ROUTING_TABLE) malloc(sizeof(ROUTING_TABLE_STRUCTURE));
CHECK_MEMORY_ALLOCATION(table);
table→ast = ast;
table→astSize = nASTNodes;
table→anchor = anchorTable;
table→anchorSize = nAnchors;
table→base = base;
table→baseSize = nBases;
table→nextHop = nextHop;
table→nextHopSize = nNextHops;
  // ———————————— Count time
ClockOff( );
if (verbose)
  fprintf(out, "Building␣AST␣and␣anchor␣table␣table:␣%.2f␣sec\n", GetTime( ));
  // ———————————— Return the routing table
return table;
}
```

[build_rt.web]

Function: *PrintRoutingTable*( )

Description: It prints a routing table.

Parameters:

out	↔	file of output
t	←	routing table

Return value: None.

"build_rt.c" 12.2.4 ≡

```
void PrintRoutingTable(FILE *out, const ROUTING_TABLE t)
  {
  WORD i;    // index
  AST_NODE k;    // value of a node in the trie

  fprintf(out, "\n");

    // ———————————————— Print the AST
  fprintf(out, "AST:\n");
  fprintf(out, "␣␣␣␣␣␣␣kind␣LWW␣RWW␣AnchorInd␣(␣␣␣␣␣␣␣internal␣nodes␣)\n");
  fprintf(out, "␣␣␣␣␣␣␣kind␣␣␣␣␣␣␣␣␣␣␣BaseInd␣(␣␣␣␣␣␣␣␣␣␣␣full␣leaves)\n");
  fprintf(out, "␣␣␣␣␣␣␣kind␣nLP␣nRP␣␣␣BaseInd␣(␣␣␣␣␣␣␣␣special␣leaves)\n");
  fprintf(out, "␣␣␣␣␣␣␣kind␣␣␣␣␣␣␣␣nextHopInd␣(␣␣covered␣empty␣leaves)\n");
  fprintf(out, "␣␣␣␣␣␣␣kind␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(uncovered␣empty␣leaves)\n");
  fprintf(out, "␣------------------------------------------------------\n");

  for (i = 0; i < t→astSize; i++)
    {
    k = t→ast[i];

    if (IS_INTERNAL_NODE(k))
      fprintf(out, "%6u␣␣1--␣%3d␣%3d␣␣␣%7d␣(␣␣␣␣␣␣␣internal␣node)\n", i,
          GET_LEFT_WINDOW_WIDTH(k), GET_RIGHT_WINDOW_WIDTH(k),
          GET_ANCHOR_INDEX(k));
    else if (IS_FULL_LEAF(k))
      fprintf(out, "%6u␣␣011␣␣␣␣␣␣␣␣␣␣␣%7d␣(␣␣␣␣␣␣␣␣␣␣␣full␣leaf)\n", i,
          GET_BASE_INDEX(k));
    else if (IS_SPECIAL_LEAF(k))
      fprintf(out, "%6u␣␣010␣%3d␣%3d␣␣␣%7d␣(␣␣␣␣␣␣␣␣special␣leaf)\n", i,
          GET_N_LEFT_BASES(k), GET_N_RIGHT_BASES(k), GET_BASE_INDEX(k));
    else if (IS_COVERED_EMPTY_LEAF(k))
      fprintf(out, "%6u␣␣001␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%3d␣(␣␣covered␣empty␣leaf)\n", i,
          GET_NEXT_HOP_INDEX(k));
    else if (IS_UNCOVERED_EMPTY_LEAF(k))
      fprintf(out, "%6u␣␣000␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(uncovered␣empty␣leaf)\n", i);
    else
      fprintf(out, "%6u␣Error!␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(␣␣␣␣␣␣␣unknown␣␣␣␣␣␣␣)\n", i);
    }

    // ———————————————— Print the anchor table
  fprintf(out, "Anchor␣table:\n");
  fprintf(out, "␣␣␣␣␣␣␣␣␣␣point␣␣␣logStep␣AST_Ind\n");
  for (i = 0; i < t→anchorSize; i++)
    {
    fprintf(out, "%6u␣", i);
```

```
      fprintf(out, "%010u␣␣␣", t→anchor[i].point);
      fprintf(out, "%3d␣␣␣", GET_LOG_STEP(t→anchor[i].fields));
      fprintf(out, "%7d\n", GET_AST_INDEX(t→anchor[i].fields));
      }
    // ——————————————— Print the base vector
    fprintf(out, "Base␣vector:\n");
    fprintf(out, "␣␣␣␣␣␣␣␣␣␣point␣␣␣nextHop\n");
    for (i = 0; i < t→baseSize; i++)
      {
      fprintf(out, "%6u␣", i);
      fprintf(out, "%010u␣␣␣", t→base[i].point);
      fprintf(out, "%3d\n", t→base[i].nextHopIndex);
      }
    // ——————————————— Print the next-hop table
    fprintf(out, "Next-hop␣table:\n");
    fprintf(out, "␣␣␣␣␣␣␣␣next-hop\n");
    for (i = 0; i < t→nextHopSize; i++)
      fprintf(out, "%6u␣%010u\n", i, t→nextHop[i]);
    // ——————————————— Flush the file of output
    fflush(out);
    }
```

[build_rt.web]

Function: *DisposeRoutingTable*( )
Description: It disposes a routing table.
Parameters: t ↔ routing table
Return value: None.

"build_rt.c" 12.2.5 ≡

```
  void DisposeRoutingTable(ROUTING_TABLE t)
    {
    free(t→ast);
    free(t→anchor);
    free(t→base);
    free(t→nextHop);
    free(t);
    }
```

[build_rt.web]

Internal functions

Internal function: *BuildNextHopTable*( )

Description: It builds the *next-hop table.*

Note: The *next-hop table* contains all the next-hops of the routing repeated only once and sorted.

Important: We store in position 0 of the *nextHop*[ ] array the *DEFAULT_NEXT_HOP*: this is useful during the search. In fact when we arrive in a full leaf and the prefix pointed by it is not a prefix of the searched IP address, we return the next-hop pointed by that full leaf. That next-hop could be either a next-hop which belongs to another prefix or the default next-hop.

Parameters:

entry	←	routing table entries stored in an initial simple array
nEntries	←	# of entries of the routing table
nextHop	→	pointer to the built *next-hop table*

Return value: Size of the next-hop table.

"build_rt.c" 12.2.6 ≡

```
static WORD BuildNextHopTable(const ENTRY entry[ ], WORD nEntries,
      NEXT_HOP *nextHop[ ])
  {
  NEXT_HOP *tempNextHop;    // temporary next-hop table (bigger than the final one)
  WORD i;   // index
  WORD nNextHops;    // # of entries of the (final) next-hop table

    /* ——————————— Extract the next-hop addresses from the entry array */
  tempNextHop = (NEXT_HOP *) calloc(nEntries, sizeof(NEXT_HOP));
  CHECK_MEMORY_ALLOCATION(tempNextHop);
  for (i = 0; i < nEntries; i++)
    tempNextHop[i] = entry[i]→nextHop;

    // ——————————— Sort elements
  qsort(tempNextHop, nEntries, sizeof(NEXT_HOP),
      (int (*)(const void *, const void *)) CompareNextHops);

    // ——————————— Remove duplicates
  nNextHops = nEntries > 0 ? 1 : 0;
  for (i = 1; i < nEntries; i++)
    if (CompareNextHops(&tempNextHop[i − 1], &tempNextHop[i]) ≠ 0)
      tempNextHop[nNextHops++] = tempNextHop[i];

      // ——————————— Check # of next-hops
  nNextHops++;
  if (nNextHops > MAX_NEXT_HOPS)
    {
    fprintf(stderr,
        "BuildNextHopTable():␣maximum␣allowed␣number␣of␣next-hops␣exceeded.\n");
    abort( );
    }

    /* ——————————— Move the elements to an array of proper size */
  *nextHop = (NEXT_HOP *) calloc(nNextHops, sizeof(NEXT_HOP));
  CHECK_MEMORY_ALLOCATION(*nextHop);
  (*nextHop)[0] = DEFAULT_NEXT_HOP;
  for (i = 1; i < nNextHops; i++)
```

```
    (*nextHop)[i] = tempNextHop[i − 1];
  free(tempNextHop);

    // ——————————— Return the next-hop table
  return nNextHops;
  }
```

[build_rt.web]

Internal function: *CompareNextHops*( )

Description: It compares two next-hop addresses.

Note: It is used by ***qsort***.

Parameters:

i	←	first next-hop
j	←	second next-hop

Return value:

-1	$\equiv$	$i < j$
1	$\equiv$	$i > j$
0	$\equiv$	$i = j$

"build_rt.c" 12.2.7 ≡

```
static int CompareNextHops(const NEXT_HOP *i, const NEXT_HOP *j)
  {
  if (*i < *j)
    return −1;
  else if (*i > *j)
    return 1;
  else
    return 0;
  }
```

[build_rt.web]

Internal function: *BuildBaseVector*( )

Description: It builds the *base vector*.

Parameters:

entry	←	routing table entries stored in an initial simple array
nEntries	←	# of entries of the routing table
nextHop	←	*next-hop table*
nNextHops	←	# of next-hops
base	→	pointer to the built *base vector*

Return value: Number of bases.

"build_rt.c" 12.2.8 ≡

```
static WORD BuildBaseVector(const ENTRY entry[ ], WORD nEntries,
      const NEXT_HOP nextHop[ ], WORD nNextHops, BASE *base[ ])
  {
  BASE *tempBase;   // temporary base vector (bigger than the final one)
  WORD nBases;   // # of bases
  WORD nExaminedEntries;   // # of examined entries of the entry vector
```

```
  // ——————————————— Allocate a bigger base vector
tempBase = (BASE *) calloc(MAX_BASES, sizeof(BASE));
CHECK_MEMORY_ALLOCATION(tempBase);

  // ——————————————— Insert the first point
  /* Note: the first point is always 0. */
tempBase[0].point = 0;
tempBase[0].nextHopIndex = DEFAULT_NEXT_HOP_INDEX;

  // ——————————————— Build base vector
nBases = 1;
nExaminedEntries = 0;
RecursivelyBuildBaseVector(entry, &nExaminedEntries, nEntries, FALSE, 0,
    nextHop, nNextHops, tempBase, &nBases);

  // ——————————————— Reallocate extreme vector
*base = realloc(tempBase, nBases * sizeof(BASE));
CHECK_MEMORY_ALLOCATION(*base);

  // ——————————————— Return the # of extremes
return nBases;
}
```

[build_rt.web]

Internal function: *RecursivelyBuildBaseVector*( )

Description: It recursively creates the base vector.

Important: The first call of this function should be something like
RecursivelyBuildBaseVector(*entry*, &*nExaminedEntries*, *nEntries*, *FALSE*, 0,
nextHop, *nNextHops*, *base*, &*nBases*);
where before this call the value of *nBases* should be 1 and *nExaminedEntries* should be 0.

Note: The set of segments associated to prefixes is an inclusion complex: two segments are either disjoint or one is included in the other.

Important: In this function we assume that the size of allocated memory for *base*[ ] array is at least *MAX_BASES*.

Parameters:

entry	←	routing table entries stored in an initial simple array
nExaminedEntries	↔	# of just examined entries
nEntries	←	# of entries in the routing table
isIncluded	←	is the entry to examine included in a prefix's segment encountered before?
includerIndex	←	if *isIncluded* ≡ *TRUE* this is the index of includer in entry vector
nextHop	←	*next-hop table*
nNextHops	←	# of next-hops
base	↔	*base vector*
nBases	↔	# of bases inserted in *base*[ ]

Return value: None.

"build_rt.c" 12.2.9 ≡

```
static void RecursivelyBuildBaseVector(const ENTRY entry[], WORD *nExaminedEntries,
      WORD nEntries, BOOL isIncluded, WORD includerIndex,
      const NEXT_HOP nextHop[], WORD nNextHops, BASE base[], WORD *nBases)
```

```
{
WORD currIndex;    // index of the current entry to examine
while ( (*nExaminedEntries < nEntries) ∧
      ( (¬isIncluded) ∨ (PREFIX_RIGHT_EXTREME(*(entry[includerIndex])) + 1 ≡ 0) ∨
      (PREFIX_LEFT_EXTREME(*(entry[*nExaminedEntries])) <
      PREFIX_RIGHT_EXTREME(*(entry[includerIndex]))) ) )
  {
    // ———————————— Increment # of examined entries
  currIndex = *nExaminedEntries;
  (*nExaminedEntries)++;
    // ———————————— Examine left extreme
  if (base[*nBases − 1].point ≡ PREFIX_LEFT_EXTREME(*(entry[currIndex])))
    base[*nBases − 1].nextHopIndex = BinarySearch(entry[currIndex]→nextHop, nextHop,
        nNextHops);
  else
    {
    if (*nBases ≥ MAX_BASES)
      {
      fprintf(stderr, "RecursivelyBuildBaseVector():␣%s\n",
          "maximum␣allowed␣number␣of␣bases␣exceeded.");
      abort( );
      }
    base[*nBases].point = PREFIX_LEFT_EXTREME(*(entry[currIndex]));
    base[*nBases].nextHopIndex = BinarySearch(entry[currIndex]→nextHop, nextHop, nNextHops);
    (*nBases)++;
    }
    // ———————————— Examine included prefixes
  if ( (currIndex < nEntries − 1) ∧ ((PREFIX_RIGHT_EXTREME(*(entry[currIndex])) + 1 ≡ 0) ∨
        (PREFIX_RIGHT_EXTREME(*(entry[currIndex])) ≥
        PREFIX_RIGHT_EXTREME(*(entry[currIndex + 1])))) )
    RecursivelyBuildBaseVector(entry, nExaminedEntries, nEntries, TRUE, currIndex,
        nextHop, nNextHops, base, nBases);
    // ———————————— Examine right point
  if (PREFIX_RIGHT_EXTREME(*(entry[currIndex])) + 1 ≠ 0)
    {
    if (base[*nBases − 1].point ≡ PREFIX_RIGHT_EXTREME(*(entry[currIndex])) + 1)
      base[*nBases − 1].nextHopIndex = ¬isIncluded ? DEFAULT_NEXT_HOP_INDEX :
          BinarySearch(entry[includerIndex]→nextHop, nextHop, nNextHops);
    else
      {
      if (*nBases ≥ MAX_BASES)
        {
        fprintf(stderr, "RecursivelyBuildBaseVector():␣%s\n",
            "maximum␣allowed␣number␣of␣bases␣exceeded.");
        abort( );
        }
      base[*nBases].point = PREFIX_RIGHT_EXTREME(*(entry[currIndex])) + 1;
      base[*nBases].nextHopIndex = ¬isIncluded ? DEFAULT_NEXT_HOP_INDEX :
          BinarySearch(entry[includerIndex]→nextHop, nextHop, nNextHops);
      (*nBases)++;
```

```
            }
          }
        }
      }
```

[build_rt.web]

Internal function: *BinarySearch*( )

Description: It searches for the prefix of a string in a vector of strings.

Parameters:

x	←	searched string
v	←	vector of strings
n	←	length of the vector of strings

Return value: Index of the searched value, or -1 if x is not in v.

"build_rt.c" 12.2.10 ≡

```
static int BinarySearch(NEXT_HOP x, const NEXT_HOP v[ ], WORD n)
  {
  WORD low, high, mid;

  low = 0;
  high = n − 1;
  while (low ≤ high)
    {
    mid = (low + high) / 2;
    if (x < v[mid])
      high = mid − 1;
    else if (x > v[mid])
      low = mid + 1;
    else
      return mid;
    }
  return −1;
  }
```

[build_rt.web]

13 Searching the next-hop of an address

[search_h.web]

This section contains the implementation of the algorithm to search the next-hop of an IP address.

We use the following names:

- s is the searched IP address;
- *anchor* is the point chosen as anchor;
- *step* is the length of a regular bucket;
- *anchorIndex* index in the AST array of the bucket containing the anchor;
- *medianPoint* is the median of the point pointed by a special leaf.

When we reach an internal node of an AST the formula to determine the next node to visit is:

$$next_AST_node = \begin{cases} (anchorIndex - 1) - \lfloor \frac{(anchor - 1) - s}{step} \rfloor & \textbf{if } s < anchor \\ anchorIndex + \lfloor \frac{s - anchor}{step} \rfloor & \textbf{if } s \geq anchor \end{cases} \tag{1}$$

The algorithm to search the next-hop of an IP address is:

- Start from the root and ...

1. **if** the current node is an *internal node* using formula (1) determine either the next AST node to examine (in this case continue the search (**goto** 1)), or **if** the next node is outside the *window* (and the windowing is not disabled) **then return** the *DEFAULT_NEXT_HOP*;
2. **else if** the current node is a *full or special leaf* **then**:
   (a) **if** the current node is a *full leaf* **then**:
      i. **if** the prefix pointed by this full leaf is a prefix of s **then return** it;
      ii. **else return** the next-hop pointed by this full leaf;
   (b) **else** (the current node is a *special leaf*):
      i. **if** $s \geq medianPoint$ **then** scan each point on the right of the *medianPoint* **until** the last point $\leq s$ **then** return the next-hop *after* (or on) that point;
      ii. **else** ($s < medianPoint$) scan each point on the left of the *medianPoint* **until** the last point $> s$ **then** return the next-hop *before* that point;
3. **else** (the current node is an *empty leaf*):
   (a) **if** it is a *covered empty leaf* **then return** the next-hop pointed by it;
   (b) **else return** *DEFAULT_NEXT_HOP*.

The same searching code is inserted in a function (*Find*( ), page 98) and in a macro (*FIND*( ), page 93). Using the macro the search can be performed inline, saving the time of a function call.

The AST can be build also without the use of *left_window_width* and *right_window_width* (for more details about the windowing see the fields of internal nodes on page 20): to disable the windowing, build the AST using the parameter *parameters→disableWindowing* ≡ *TRUE*. *Find*( ) function and *FIND*( ) macro work also in this case, but we provided also other two ones: *Find_NoWindowing*( ) and *FIND_NO_WINDOWING*( ). These are a bit faster, but do not work when windowing is used.

We provide also a special version of the *Find*( ) function called *StatisticFind*( ) (page 102). This function returns some additional information needed for statistics (it works also when the windowing is disabled).

13.1 Searching the next-hop of an address: header

[search_h.web]

"build_rt.c" 13.1 ≡
    @O search.h

[search_h.web]

Guard

"search.h" 13.1.1 ≡

```
#ifndef _AST_SEARCH_H_
#define _AST_SEARCH_H_
```

[search_h.web]

Includes

"search.h" 13.1.2 ≡

```
#include "ast_node.h"
#include "basic_defs.h"
#include "prefix.h"
#include "routing_table.h"
```

[search_h.web]

Macros

Macro: *FIND*( )

Description: It searches for the next-hop of an address.

Important: This is the same code of *Find*( ) function. It is transformed into a macro to perform inline search (and saving procedure call time). In this macro the comments are cut away, so for more details please refer to the code of *Find*( ).

Important: If for some reason you modify *Find*( ) function, make sure to actuate the same changes also here.

Note: This macro works also if windowing is disabled, but in that case *FIND_NO_WINDOWING*( ) is a bit faster.

Parameters:

(**IP_ADDRESS**) *s*	←	searched IP address
(**const ROUTING_TABLE**) *t*	←	routing table
(**NEXT_HOP**) *_nextHop_*	→	next-hop of *s*, or 0 if it is not found

Return value: None.

"search.h" 13.1.3 ≡

```
#define FIND(s, t, _nextHop_) \
  { \
  WORD __nodeIndex__ = 0; \
  AST_NODE __node__; \
  ANCHOR *__anchorPtr__; \
  int __windowWidth__, __dist__, __nBases__, __i__; \
  BASE *__basePtr__; \
  while (TRUE) \
    { \
    __node__ = (t)→ast[__nodeIndex__]; \
    if (FAST_IS_INTERNAL_NODE(__node__)) \
      { \
      __anchorPtr__ = &((t)→anchor[GET_ANCHOR_INDEX(__node__)]); \
      if ((s) ≥ __anchorPtr__→point) \
        { \
        __windowWidth__ = GET_RIGHT_WINDOW_WIDTH(__node__); \
        __dist__ = ((s) − __anchorPtr__→point) >> GET_LOG_STEP(__anchorPtr__→fields); \
        if ((__windowWidth__ ≠ 0) ∧ (__dist__ ≥ __windowWidth__)) \
          { (_nextHop_) = DEFAULT_NEXT_HOP; break; } \
        else \
          __nodeIndex__ = GET_AST_INDEX(__anchorPtr__→fields) + __dist__; \
        } \
      else \
        { \
        __windowWidth__ = GET_LEFT_WINDOW_WIDTH(__node__); \
        __dist__ = (__anchorPtr__→point − 1 − (s)) >> GET_LOG_STEP(__anchorPtr__→fields); \
        if ((__windowWidth__ ≠ 0) ∧ (__dist__ ≥ __windowWidth__)) \
          { (_nextHop_) = DEFAULT_NEXT_HOP; break; } \
        else \
          __nodeIndex__ = GET_AST_INDEX(__anchorPtr__→fields) − 1 − __dist__; \
        } \
      } \
    else if (IS_FULL_OR_SPECIAL_LEAF(__node__)) \
      { \
      if (FAST_IS_FULL_LEAF(__node__)) \
        { \
        __basePtr__ = &((t)→base[GET_BASE_INDEX(__node__)]); \
        if ((s) ≥ __basePtr__→point) \
          { (_nextHop_) = (t)→nextHop[__basePtr__→nextHopIndex]; break; } \
        else \
          { (_nextHop_) = (t)→nextHop[(__basePtr__ − 1)→nextHopIndex]; break; } \
        } \
      else \
        { \
        __basePtr__ = &((t)→base[GET_BASE_INDEX(__node__)]); \
        if ((s) ≥ __basePtr__→point) \
          { \
          __nBases__ = GET_N_RIGHT_BASES(__node__); \
          __i__ = 0; \
          while ((__i__ < __nBases__) ∧ ((s) ≥ (__basePtr__ + 1)→point)) \
            { __basePtr__++; __i__++; } \
          (_nextHop_) = (t)→nextHop[__basePtr__→nextHopIndex]; break; \
```

```
          } \
        else \
          { \
          __nBases__ = GET_N_LEFT_BASES(__node__); \
          __i__ = 0; \
          while ((__i__ < __nBases__) ∧ ((s) < (__basePtr__ − 1)→point)) \
            { __basePtr__−−; __i__++; } \
          (_nextHop_) = (t)→nextHop[(__basePtr__ − 1)→nextHopIndex]; break; \
          } \
        } \
      } \
    else if (FAST_IS_COVERED_EMPTY_LEAF(__node__)) \
      { (_nextHop_) = (t)→nextHop[GET_NEXT_HOP_INDEX(__node__)]; break; } \
    else \
      { (_nextHop_) = DEFAULT_NEXT_HOP; break; } \
    } \
  }
```

[search_h.web]

Macro: *FIND_NO_WINDOWING*( )

Description: It searches for the next-hop of an address.

Important: This version of *FIND*( ) is a bit faster, but works only if the AST is built without the use of *left_window_width* and *right_window_width* (for more details about the windowing see the fields of internal nodes on page 20). To disable the windowing, build the AST using the parameter *parameters→disableWindowing* ≡ *TRUE*.

Important: This is the same code of *Find_NoWindowing*( ) function. It is transformed into a macro to perform inline search (and saving procedure call time). In this macro the comments are cut away, so for more details please refer to the code of *Find_NoWindowing*( ).

Important: If for some reason you modify *Find_NoWindowing*( ) function, make sure to actuate the same changes also here.

Parameters:

(**IP_ADDRESS**) *s*	←	searched IP address
(**const ROUTING_TABLE**) *t*	←	routing table
(**NEXT_HOP**) *_nextHop_*	→	next-hop of *s*, or 0 if it is not found

Return value: None.

"search.h" 13.1.4 ≡

```
#define FIND_NO_WINDOWING(s, t, _nextHop_) \
  { \
  WORD __nodeIndex__ = 0; \
  AST_NODE __node__; \
  ANCHOR *__anchorPtr__; \
  int __nBases__, __i__; \
  BASE *__basePtr__; \

  while (TRUE) \
    { \
    __node__ = (t)→ast[__nodeIndex__]; \
    if (FAST_IS_INTERNAL_NODE(__node__)) \
```

```
      { \
      __anchorPtr__ = &((t)→anchor[GET_ANCHOR_INDEX(__node__)]); \
      __nodeIndex__ = GET_AST_INDEX(__anchorPtr__→fields) + (((s) ≥ __anchorPtr__→point) ? \
          (((s) − __anchorPtr__→point) >> GET_LOG_STEP(__anchorPtr__→fields)) : \
          −1 − ((__anchorPtr__→point − 1 − (s)) >> GET_LOG_STEP(__anchorPtr__→fields))); \
      } \
    else if (IS_FULL_OR_SPECIAL_LEAF(__node__)) \
      { \
      if (FAST_IS_FULL_LEAF(__node__)) \
        { \
        __basePtr__ = &((t)→base[GET_BASE_INDEX(__node__)]); \
        if ((s) ≥ __basePtr__→point) \
          { (_nextHop_) = (t)→nextHop[__basePtr__→nextHopIndex]; break; } \
        else \
          { (_nextHop_) = (t)→nextHop[(__basePtr__ − 1)→nextHopIndex]; break; } \
        } \
      else \
        { \
        __basePtr__ = &((t)→base[GET_BASE_INDEX(__node__)]); \
        if ((s) ≥ __basePtr__→point) \
          { \
          __nBases__ = GET_N_RIGHT_BASES(__node__); \
          __i__ = 0; \
          while ((__i__ < __nBases__) ∧ ((s) ≥ (__basePtr__ + 1)→point)) \
            { __basePtr__++; __i__++; } \
          (_nextHop_) = (t)→nextHop[__basePtr__→nextHopIndex]; break; \
          } \
        else \
          { \
          __nBases__ = GET_N_LEFT_BASES(__node__); \
          __i__ = 0; \
          while ((__i__ < __nBases__) ∧ ((s) < (__basePtr__ − 1)→point)) \
            { __basePtr__−−; __i__++; } \
          (_nextHop_) = (t)→nextHop[(__basePtr__ − 1)→nextHopIndex]; break; \
          } \
        } \
      } \
    else if (FAST_IS_COVERED_EMPTY_LEAF(__node__)) \
      { (_nextHop_) = (t)→nextHop[GET_NEXT_HOP_INDEX(__node__)]; break; } \
    else \
      { (_nextHop_) = DEFAULT_NEXT_HOP; break; } \
    } \
  }
```

[search_h.web]

Prototypes

"search.h" 13.1.5 ≡

```
NEXT_HOP Find(IP_ADDRESS s, const ROUTING_TABLE t);
NEXT_HOP Find_NoWindowing(IP_ADDRESS s, const ROUTING_TABLE t);
NEXT_HOP StatisticFind(IP_ADDRESS s, const ROUTING_TABLE t, int *nASTAccesses,
    int *nAnchorAccesses, int *nBaseAccesses, int *nNextHopAccesses);
```

[search_h.web]

Guard

"search.h" 13.1.6 ≡

```
#endif    // _AST_SEARCH_H_
```

[search_h.web]

13.2 Searching the next-hop of an address: implementation

[search.web]

"search.h" 13.2 ≡
  @O search.c

[search.web]

Includes

"search.c" 13.2.1 ≡

```
#include "anchor.h"
#include "ast_node.h"
#include "base.h"
#include "basic_defs.h"
#include "next_hop.h"
#include "routing_table.h"
#include "search.h"
```

[search.web]

Functions

Function: *Find*( )

Description: It searches for the next-hop of an address.

Important: If for some reason you modify this function make sure to actuate the same changes also to *FIND*( ) (macro version of this function), to *Find_NoWindowing*( ) (function to search when the windowing is disabled), to *FIND_NO_WINDOWING*( ) (macro version of *Find_NoWindowing*( )) and to *StatisticFind*( ) (version of the *Find*( ) that returns some information needed for statistics).

Note: This function works also if windowing disabled, but in that case *Find_NoWindowing*( ) is a bit faster.

Parameters:
s ← searched IP address
t ← routing table

Return value: Next-hop of s, or *DEFAULT_NEXT_HOP* if it is not found.

"search.c" 13.2.2 ≡

```
NEXT_HOP Find(IP_ADDRESS s, const ROUTING_TABLE t)
  {
  WORD nodeIndex = 0;   // index of the current examined node
  AST_NODE node;   // current examined node of an AST
  ANCHOR *anchorPtr;   // pointer to an anchor record
  int windowWidth;   // # of nodes after which there are only empty leaves
  int dist;   // distance of the next node from the anchor
  BASE *basePtr;   // pointer to the base record pointed by a leaf
  int nBases;   // # of bases pointed by a special leaf
  int i;   // index to scan bases
```

```
while (TRUE)
  {
  node = t→ast[nodeIndex];

    // ———————————— If the node is internal ...
  if (FAST_IS_INTERNAL_NODE(node))
    {
    anchorPtr = &(t→anchor[GET_ANCHOR_INDEX(node)]);

    if (s ≥ anchorPtr→point)
      {
      windowWidth = GET_RIGHT_WINDOW_WIDTH(node);

      dist = (s − anchorPtr→point) >> GET_LOG_STEP(anchorPtr→fields);

      if ((windowWidth ≠ 0) ∧ (dist ≥ windowWidth))
        return DEFAULT_NEXT_HOP;
      else
        nodeIndex = GET_AST_INDEX(anchorPtr→fields) + dist;
      }
    else    /* (s < anchorPtr→point) */
      {
      windowWidth = GET_LEFT_WINDOW_WIDTH(node);

      dist = (anchorPtr→point − 1 − s) >> GET_LOG_STEP(anchorPtr→fields);

      if ((windowWidth ≠ 0) ∧ (dist ≥ windowWidth))
        return DEFAULT_NEXT_HOP;
      else
        nodeIndex = GET_AST_INDEX(anchorPtr→fields) − 1 − dist;
      }
    }

    // ———————————— If the node is a leaf ...
  else if (IS_FULL_OR_SPECIAL_LEAF(node))
    {
      // ———————————— If the node is a full leaf ...
    if (FAST_IS_FULL_LEAF(node))
      {
      basePtr = &(t→base[GET_BASE_INDEX(node)]);

      if (s ≥ basePtr→point)
        return t→nextHop[basePtr→nextHopIndex];
      else
        return t→nextHop[(basePtr − 1)→nextHopIndex];
      }

      // ———————————— If the node is a special leaf...
    else    /* (IS_SPECIAL_LEAF(node)) */
      {
      basePtr = &(t→base[GET_BASE_INDEX(node)]);

      if (s ≥ basePtr→point)
        {
        nBases = GET_N_RIGHT_BASES(node);

        i = 0;
        while ((i < nBases) ∧ (s ≥ (basePtr + 1)→point))
```

```
                {
                basePtr++;
                i++;
                }
            return t→nextHop[basePtr→nextHopIndex];
            }
          else    /* (s < basePtr→point) */
            {
            nBases = GET_N_LEFT_BASES(node);

            i = 0;
            while ((i < nBases) ∧ (s < (basePtr − 1)→point))
                {
                basePtr−−;
                i++;
                }
            return t→nextHop[(basePtr − 1)→nextHopIndex];
            }
          }
        }
        /* ———————————— If the node is a covered empty leaf ... */
      else if (FAST_IS_COVERED_EMPTY_LEAF(node))
        return t→nextHop[GET_NEXT_HOP_INDEX(node)];

        /* ———————————— If the node is an uncovered empty leaf ... */
      else    /* (IS_UNCOVERED_EMPTY_LEAF(node)) */
        return DEFAULT_NEXT_HOP;
      }
    }
```

[search.web]

Function: *Find_NoWindowing*( )

Description: It searches for the next-hop of an address.

Important: This version of *Find*( ) is a bit faster, but works only if the AST is built without the use of the windowing (for more details about the windowing see the fields of internal nodes on page 20). To disable the windowing, build the AST using the parameter *parameters*→*disableWindowing* ≡ *TRUE*.

Important: The code in this function is almost the same of *Find*( ) except that the conditions concerning the windowing are deleted: if for some reason you modify *Find*( ) function, make sure to actuate the same changes also here. *FIND_NO_WINDOWING*( ) is the macro version of this function: if for some reason you modify this function make sure to actuate the same changes also to it.

Parameters:
- s ← searched IP address
- t ← routing table

Return value: Next-hop of s, or *DEFAULT_NEXT_HOP* if it is not found.

"search.c" 13.2.3 ≡

```
NEXT_HOP Find_NoWindowing(IP_ADDRESS s, const ROUTING_TABLE t)
  {
```

```
WORD nodeIndex = 0;    // index of the current examined node
AST_NODE node;    // current examined node of an AST
ANCHOR *anchorPtr;    // pointer to an anchor record
BASE *basePtr;    // pointer to the base record pointed by a leaf
int nBases;    // # of bases pointed by a special leaf
int i;    // index to scan bases

while (TRUE)
  {
  node = t->ast[nodeIndex];

    // ———————————— If the node is internal ...
  if (FAST_IS_INTERNAL_NODE(node))
    {
    anchorPtr = &(t->anchor[GET_ANCHOR_INDEX(node)]);

    nodeIndex = GET_AST_INDEX(anchorPtr->fields) + ((s ≥ anchorPtr->point) ?
        ((s − anchorPtr->point) >> GET_LOG_STEP(anchorPtr->fields)) :
        −1 − ((anchorPtr->point − 1 − s) >> GET_LOG_STEP(anchorPtr->fields)));
    }

    // ———————————— If the node is a leaf ...
  else if (IS_FULL_OR_SPECIAL_LEAF(node))
    {
      // ———————————— If the node is a full leaf ...
    if (FAST_IS_FULL_LEAF(node))
      {
      basePtr = &(t->base[GET_BASE_INDEX(node)]);

      if (s ≥ basePtr->point)
        return t->nextHop[basePtr->nextHopIndex];
      else
        return t->nextHop[(basePtr − 1)->nextHopIndex];
      }

      // ———————————— If the node is a special leaf...
    else    /* (IS_SPECIAL_LEAF(node)) */
      {
      basePtr = &(t->base[GET_BASE_INDEX(node)]);

      if (s ≥ basePtr->point)
        {
        nBases = GET_N_RIGHT_BASES(node);

        i = 0;
        while ((i < nBases) ∧ (s ≥ (basePtr + 1)->point))
          {
          basePtr++;
          i++;
          }

        return t->nextHop[basePtr->nextHopIndex];
        }
      else    /* (s < basePtr->point) */
        {
        nBases = GET_N_LEFT_BASES(node);

        i = 0;
```

```
            while ((i < nBases) ∧ (s < (basePtr − 1)→point))
              {
              basePtr−−;
              i++;
              }
            return t→nextHop[(basePtr − 1)→nextHopIndex];
            }
          }
        }
        /* ———————————— If the node is a covered empty leaf ... */
      else if (FAST_IS_COVERED_EMPTY_LEAF(node))
        return t→nextHop[GET_NEXT_HOP_INDEX(node)];
        /* ———————————— If the node is an uncovered empty leaf ... */
      else    /* (IS_UNCOVERED_EMPTY_LEAF(node)) */
        return DEFAULT_NEXT_HOP;
      }
    }
```

[search.web]

Function: *StatisticFind*( )

Description: It searches for the next-hop of an address.

Important: This function is the same as *Find*( ) except that it returns some information needed for statistics. If for some reason you modify *Find*( ) function, make sure to actuate the same changes also here.

Note: This function works also if the windowing is disabled.

Parameters:

s	←	searched IP address
t	←	routing table
nASTAccesses	→	# of accesses to the AST
nAnchorAccesses	→	# of accesses to the anchor table
nBaseAccesses	→	# of accesses to the base vector
nNextHopAccesses	→	# of accesses to the next-hop table

Return value: Next-hop of s, or *DEFAULT_NEXT_HOP* if it is not found.

"search.c" 13.2.4 ≡

```
NEXT_HOP StatisticFind(IP_ADDRESS s, const ROUTING_TABLE t, int *nASTAccesses,
      int *nAnchorAccesses, int *nBaseAccesses, int *nNextHopAccesses)
  {
  WORD nodeIndex = 0;   // index of the current examined node
  AST_NODE node;   // current examined node of an AST
  ANCHOR *anchorPtr;   // pointer to an anchor record
  int windowWidth;   // # of nodes after which there are only empty leaves
  int dist;   // distance of the next node from the anchor
  BASE *basePtr;   // pointer to the base record pointed by a leaf
  int nBases;   // # of bases pointed by a special leaf
  int i;   // index to scan bases
    // ———————————— Initialize indices
```

```
*nASTAccesses = 0;
*nAnchorAccesses = 0;
*nBaseAccesses = 0;
*nNextHopAccesses = 0;

while (TRUE)
  {
  (*nASTAccesses)++;
  node = t→ast[nodeIndex];

    // ———————————— If the node is internal ...
  if (FAST_IS_INTERNAL_NODE(node))
    {
    (*nAnchorAccesses)++;
    anchorPtr = &(t→anchor[GET_ANCHOR_INDEX(node)]);

    if (s ≥ anchorPtr→point)
      {
      windowWidth = GET_RIGHT_WINDOW_WIDTH(node);

      dist = (s − anchorPtr→point) >> GET_LOG_STEP(anchorPtr→fields);

      if ((windowWidth ≠ 0) ∧ (dist ≥ windowWidth))
        return DEFAULT_NEXT_HOP;
      else
        nodeIndex = GET_AST_INDEX(anchorPtr→fields) + dist;
      }
    else    /* (s < anchorPtr→point) */
      {
      windowWidth = GET_LEFT_WINDOW_WIDTH(node);

      dist = (anchorPtr→point − 1 − s) >> GET_LOG_STEP(anchorPtr→fields);

      if ((windowWidth ≠ 0) ∧ (dist ≥ windowWidth))
        return DEFAULT_NEXT_HOP;
      else
        nodeIndex = GET_AST_INDEX(anchorPtr→fields) − 1 − dist;
      }
    }
    // ———————————— If the node is a leaf ...
  else if (IS_FULL_OR_SPECIAL_LEAF(node))
    {
      // ———————————— If the node is a full leaf ...
    if (FAST_IS_FULL_LEAF(node))
      {
      basePtr = &(t→base[GET_BASE_INDEX(node)]);

      (*nBaseAccesses)++;
      (*nNextHopAccesses)++;
      if (s ≥ basePtr→point)
        return t→nextHop[basePtr→nextHopIndex];
      else
        {
        (*nBaseAccesses)++;
        return t→nextHop[(basePtr − 1)→nextHopIndex];
        }
```

```
        }
        // ————————————————— If the node is a special leaf...
      else    /* (IS_SPECIAL_LEAF(node)) */
        {
        basePtr = &(t→base[GET_BASE_INDEX(node)]);

        (*nBaseAccesses)++;
        if (s ≥ basePtr→point)
          {
          nBases = GET_N_RIGHT_BASES(node);

          i = 0;
          while ((i < nBases) ∧ ((*nBaseAccesses)++, (s ≥ (basePtr + 1)→point)))
            {
            basePtr++;
            i++;
            }

          (*nNextHopAccesses)++;
          return t→nextHop[basePtr→nextHopIndex];
          }
        else    /* (s < basePtr→point) */
          {
          nBases = GET_N_LEFT_BASES(node);

          i = 0;
          while ((i < nBases) ∧ ((*nBaseAccesses)++, (s < (basePtr − 1)→point)))
            {
            basePtr−−;
            i++;
            }

          (*nNextHopAccesses)++;
          return t→nextHop[(basePtr − 1)→nextHopIndex];
          }
        }
      }
      /* ————————————————— If the node is a covered empty leaf ... */
    else if (FAST_IS_COVERED_EMPTY_LEAF(node))
      {
      (*nNextHopAccesses)++;
      return t→nextHop[GET_NEXT_HOP_INDEX(node)];
      }
      /* ————————————————— If the node is an uncovered empty leaf ... */
    else    /* (IS_UNCOVERED_EMPTY_LEAF(node)) */
      return DEFAULT_NEXT_HOP;
    }
  }
```

[search.web]

14 CPU time measurement

[clock_h.web]

The following routines allow to measure the CPU time taken by a set of instructions.

Use the following scheme:

- before the set of instructions: call *ClockOn*( );
- ... instructions ...
- after the set of instructions: call *ClockOff* ( );
- to know the seconds taken by the instruction: call *GetTime*( ).

14.1 CPU time measurement: header

[clock_h.web]

"search.c" 14.1 ≡

```
@O clock.h
```

[clock_h.web]

Guard

"clock.h" 14.1.1 ≡

```
#ifndef _AST_CLOCK_H_
#define _AST_CLOCK_H_
```

[clock_h.web]

Prototypes

"clock.h" 14.1.2 ≡

```
void ClockOn(void);
void ClockOff(void);
double GetTime(void);
```

[clock_h.web]

Guard

"clock.h" 14.1.3 ≡

```
#endif    // _AST_CLOCK_H_
```

[clock_h.web]

14.2 CPU time measurement: implementation

[clock.web]

"clock.h" 14.2 ≡
```
@O clock.c
```

[clock.web]

Includes

"clock.c" 14.2.1 ≡

```
#include <time.h>
#include "clock.h"
```

[clock.web]

Internal global variables

"clock.c" 14.2.2 ≡
```
static clock_t startClock, stopClock;
```

[clock.web]

Functions

Function: *ClockOn* ( )

Description: It starts the measurement of the CPU time.

Parameters: None.

Return value: None.

"clock.c" 14.2.3 ≡
```
void ClockOn(void)
  {
  startClock = clock( );
  }
```

[clock.web]

Function: *ClockOff* ( )

Description: It stops the measurement of the CPU time.

Parameters: None.

Return value: None.

"clock.c" 14.2.4 ≡

```
void ClockOff (void)
  {
  stopClock = clock( );
  }
```

[clock.web]

Function: *GetTime*( )
Description: It gives the seconds elapsed between the precedent calls of *ClockOn*( ) and *ClockOff*( ).
Parameters: None.
Return value: The elapsed seconds.

"clock.c" 14.2.5 ≡

```
double GetTime(void)
  {
  return (stopClock − startClock) / (double) CLOCKS_PER_SEC;
  }
```

[clock.web]

15 Statistics

[statistics_h.web]

Here there are some routines to print statistics about the routing table.

15.1 Statistics: header

[statistics_h.web]

"clock.c" 15.1 ≡

```
@O statistics.h
```

[statistics_h.web]

Guard

"statistics.h" 15.1.1 ≡

```
#ifndef _AST_STATISTICS_H_
#define _AST_STATISTICS_H_
```

[statistics_h.web]

Includes

"statistics.h" 15.1.2 ≡

```
#include <stdio.h>

#include "basic_defs.h"
#include "parameters.h"
#include "routing_table.h"
```

[statistics_h.web]

Prototypes

"statistics.h" 15.1.3 ≡

```
void PrintRoutingTableStatistics(FILE *out, const ROUTING_TABLE t,
    const PARAMETERS *parameters);
void RunSearchTest(FILE *out, const IP_ADDRESS testData[ ], WORD nAddresses, const
    ROUTING_TABLE table, const PARAMETERS *parameters, BOOL useInline,
    int nExperiments, BOOL getMemoryAccesses, BOOL verbose);
```

[statistics_h.web]

Guard

"statistics.h" 15.1.4 ≡

```
#endif    // _AST_STATISTICS_H_
```

[statistics_h.web]

15.2 Statistics: implementation

[statistics.web]

"statistics.h" 15.2 ≡
```
@O statistics.c
```

[statistics.web]

Includes

"statistics.c" 15.2.1 ≡

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "anchor.h"
#include "ast_node.h"
#include "base.h"
#include "basic_defs.h"
#include "bucket.h"
#include "clock.h"
#include "next_hop.h"
#include "parameters.h"
#include "routing_table.h"
#include "search.h"
#include "statistics.h"
```

[statistics.web]

Internal constants

Internal constant: *MAX_EXPERIMENTS*

Description: Maximum # of experiments that the *RunSearchTest*( ) function can run (to understand what an experiment is read the description of that function).

"statistics.c" 15.2.2 ≡

```
#define MAX_EXPERIMENTS 100
```

[statistics.web]

Internal prototypes

"statistics.c" 15.2.3 ≡
```
static void TraverseAST(const AST_NODE ast[], const ANCHOR anchorTable[],
        AST_NODE root, IP_ADDRESS left, IP_ADDRESS right, int currDepth,
        WORD *sumLeafDepths, int *maxDepth, WORD nLeavesPerLevel[],
```

```
double *sumDepthTimesWeight, double *totalWeight, WORD *nFullL, WORD *nSpecialL,
WORD *nCoveredEL, WORD *nUncoveredEL);
```

[statistics.web]

Functions

Function: *PrintRoutingTableStatistics*( )

Description: It prints some statistics about the routing table.

Parameters:

out	↔	file of output
t	←	routing table
parameters	←	parameters

Return value: None.

"statistics.c" 15.2.4 ≡

```
void PrintRoutingTableStatistics(FILE *out, const ROUTING_TABLE t,
      const PARAMETERS *parameters)
  {
  WORD totalMemory;    // total used memory
  int i, j;    // indexes
  int maxDepth;    // maximum depth of the AST
  WORD sumLeafDepths;    // sum of the depths of all the leaves in the AST
  WORD nLeavesPerLevel[MAX_AST_LEVELS];    // # of leaves at each level of the AST
  double sumDepthTimesWeight;    /* ∑ of the depth of the leaves multiplied by the weight */
  double totalWeight;    // ∑ of weights of leaves
  WORD nFullL;    // # of full leaves
  WORD nSpecialL;    // # of special leaves
  WORD nCoveredEL;    // # of covered empty leaves
  WORD nUncoveredEL;    // # of uncovered empty leaves
  WORD nLeaves;    // # of leaves

  fprintf(out, "\n");

    // ———————————————— Compute memory occupation
  totalMemory = t→astSize * sizeof(AST_NODE) + t→anchorSize * sizeof(ANCHOR) +
      t→baseSize * sizeof(BASE) + t→nextHopSize * sizeof(NEXT_HOP);

    // ———————————————— Print statistics about the AST
  fprintf(out, "AST\n");

  fprintf(out, "␣␣␣␣␣Parameters:␣branch␣at␣root␣=␣");
  if ((parameters→logRootBranch ≡ 0) ∧ (IS_INTERNAL_NODE(t→ast[0])))
    fprintf(out, "2^%d\n",
        IP_ADDRESS_SIZE − GET_LOG_STEP(t→anchor[GET_ANCHOR_INDEX(t→ast[0])].fields));
  else
    fprintf(out, "2^%d␣(fixed)\n", parameters→logRootBranch);
  fprintf(out, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣c␣=");
  for (i = 0; i < 9; i++)
    fprintf(out, "␣%05.3f", parameters→c[i]);
  fprintf(out, "\n");
  fprintf(out, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣max␣special␣leaf␣=␣%d\n", parameters→maxSpecialLeaf);
```

```
fprintf(out, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣windowing␣%s\n",
     parameters→disableWindowing ? "disabled" : "enabled");
fprintf(out, "␣␣␣␣␣Memory:␣␣␣␣␣");
PRINT_WORD(out, t→astSize, 0);
fprintf(out, "*%d␣bytes␣=␣", sizeof(AST_NODE));
PRINT_WORD(out, t→astSize * sizeof(AST_NODE), 0);
fprintf(out, "␣bytes␣=␣%.2f%%␣of␣total␣memory\n",
     (double) t→astSize * sizeof(AST_NODE) / totalMemory * 100.0);

nLeaves = sumLeafDepths = maxDepth = 0;
for (i = 0; i < MAX_AST_LEVELS; i++)
  nLeavesPerLevel[i] = 0;
sumDepthTimesWeight = totalWeight = 0.0;
nFullL = nSpecialL = nCoveredEL = nUncoveredEL = 0;

TraverseAST(t→ast, t→anchor, t→ast[0], 0, MAX_IP_ADDRESS, 0,
     &sumLeafDepths, &maxDepth, nLeavesPerLevel, &sumDepthTimesWeight, &totalWeight,
     &nFullL, &nSpecialL, &nCoveredEL, &nUncoveredEL);

nLeaves = nFullL + nSpecialL + nCoveredEL + nUncoveredEL;

fprintf(out, "␣␣␣␣␣Statistics:␣internal␣nodes:␣");
PRINT_WORD(out, t→astSize − nLeaves, 7);
fprintf(out, "\n␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣leaves:␣␣␣␣␣␣␣␣␣");
PRINT_WORD(out, nLeaves, 7);
fprintf(out, "␣␣␣␣␣␣full␣leaves:␣␣␣␣␣␣␣␣␣␣␣␣");
PRINT_WORD(out, nFullL, 7);
fprintf(out, "\n␣%44s␣special␣leaves:␣␣␣␣␣␣␣␣␣", "");
PRINT_WORD(out, nSpecialL, 7);
fprintf(out, "\n␣%44s␣covered␣␣␣empty␣leaves:␣", "");
PRINT_WORD(out, nCoveredEL, 7);
fprintf(out, "\n␣%44s␣uncovered␣empty␣leaves:␣", "");
PRINT_WORD(out, nUncoveredEL, 7);
fprintf(out, "\n");

for (i = MAX_AST_LEVELS − 1; i ≥ 0 ∧ nLeavesPerLevel[i] ≡ 0; i−−)
  ;

for (j = 1; j ≤ i; j++)
  {
  if (j ≡ 1)
    fprintf(out, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣leaves␣per␣level:␣");
  else
    fprintf(out, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣");
  fprintf(out, "%2d␣", j);
  PRINT_WORD(out, nLeavesPerLevel[j], 7);
  fprintf(out, "\n");
  }
fprintf(out, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣maximum␣path:␣␣␣%d\n", maxDepth);
fprintf(out, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣average␣path:␣␣␣%f\n", (double) sumLeafDepths / nLeaves);
fprintf(out, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣weighted␣depth:␣%f\n",
     (totalWeight < 0.5) ? 0.0 : (sumDepthTimesWeight / totalWeight));

  // ——————————————— Print statistics about anchor[ ]
fprintf(out, "Anchor␣table\n");
fprintf(out, "␣␣␣␣␣Memory:␣␣␣␣␣");
```

```
PRINT_WORD(out, t→anchorSize, 0);
fprintf(out, "*%d bytes = ", sizeof(ANCHOR));
PRINT_WORD(out, t→anchorSize * sizeof(ANCHOR), 0);
fprintf(out, " bytes = %.2f%% of total memory\n",
    (double) t→anchorSize * sizeof(ANCHOR) / totalMemory * 100.0);

  // ———————————— Print statistics about base[ ]
fprintf(out, "Base vector\n");
fprintf(out, "     Memory:     ");
PRINT_WORD(out, t→baseSize, 0);
fprintf(out, "*%d bytes = ", sizeof(BASE));
PRINT_WORD(out, t→baseSize * sizeof(BASE), 0);
fprintf(out, " bytes = %.2f%% of total memory\n",
    (double) t→baseSize * sizeof(BASE) / totalMemory * 100.0);

  // ———————————— Print statistics about the nextHop[ ]
fprintf(out, "Next-hop table\n");
fprintf(out, "     Memory:     ");
PRINT_WORD(out, t→nextHopSize, 0);
fprintf(out, "*%d bytes = ", sizeof(NEXT_HOP));
PRINT_WORD(out, t→nextHopSize * sizeof(NEXT_HOP), 0);
fprintf(out, " bytes = %.2f%% of total memory\n",
    (double) t→nextHopSize * sizeof(NEXT_HOP) / totalMemory * 100.0);

  // ———————————— Print the total used memory
fprintf(out, "\nMemory used for AST and anchor table:           ");
PRINT_WORD(out, t→astSize * sizeof(AST_NODE) + t→anchorSize * sizeof(ANCHOR), 9);
fprintf(out, " bytes\n");
fprintf(out, "Memory used for base vector and next-hop table: ");
PRINT_WORD(out, t→baseSize * sizeof(BASE) + t→nextHopSize * sizeof(NEXT_HOP), 9);
fprintf(out, " bytes\n");
fprintf(out, "Total used memory:                              ");
PRINT_WORD(out, totalMemory, 9);
fprintf(out, " bytes\n");

  // ———————————— Flush the file of output
fflush(out);
}
```

[statistics.web]

Internal function: *RunSearchTest*( )

Description: It searches, in the routing table, for the given IP addresses. It prints the statistics about the search on *out* file.

Note: One experiment consists in searching *parameters*→*nRepeatSearch* times, in the routing table, for the *nAddresses* IP addresses stored in *testData*[ ]. This function executes *nExperiments* times this experiment and computes the statistics about the execution time.

Parameters:

out	↔	file of output
testData	←	array containing the IP addresses to search
nAddresses	←	# of addresses in *testData*[ ]
table	←	routing table
parameters	←	parameters
useInline	←	if *TRUE* the search is performed inline
nExperiments	←	# of experiments to run (*nExperiments* ≤ *MAX_EXPERIMENTS*)
getMemoryAccesses	←	if *TRUE* it computes statistics about memory accesses
verbose	←	if *TRUE* it prints more details

Return value: None.

"statistics.c" 15.2.5 ≡

```
void RunSearchTest(FILE *out, const IP_ADDRESS testData[ ], WORD nAddresses, const
        ROUTING_TABLE table, const PARAMETERS *parameters, BOOL useInline,
        int nExperiments, BOOL getMemoryAccesses, BOOL verbose)
  {
  const int nTimes= parameters→nRepeatSearch;    /* # of times to search the nAddresses addresses
        in each experiment */
  double time[MAX_EXPERIMENTS];    // execution time of each experiment
  int i, j, k;    // indexes
  volatile NEXT_HOP nextHop;    // result of Find( ) or FIND( ): volatile to avoid optimization
  IP_ADDRESS s;    // searched IP address
  double minTime;    // minimum time
  double timeSum;    // sum of the times
  double time2Sum;    // sum of the squares of the times
  double averTime;    // average of the times
  double stdevTime;    // standard deviation of the times

    /* ———————————— Run all experiments measuring execution times ... */
  if (nExperiments > MAX_EXPERIMENTS)
    nExperiments = MAX_EXPERIMENTS;

  if (¬parameters→disableWindowing)
    {
    if (¬useInline)
      {
      for (i = 0; i < nExperiments; i++)
        {
        ClockOn( );

        for (j = 0; j < nTimes; j++)
          for (k = 0; k < nAddresses; k++)
            nextHop = Find(testData[k], table);

        ClockOff( );
        time[i] = GetTime( );
        }
```

```
        }
      else    /* (useInline) */
        {
        for (i = 0; i < nExperiments; i++)
          {
          ClockOn( );

          for (j = 0; j < nTimes; j++)
            for (k = 0; k < nAddresses; k++)
              {
              s = testData[k];
              FIND(s, table, nextHop);
              }

          ClockOff( );
          time[i] = GetTime( );
          }
        }
      }
    else    /* (parameters→disableWindowing) */
      {
      if (¬useInline)
        {
        for (i = 0; i < nExperiments; i++)
          {
          ClockOn( );

          for (j = 0; j < nTimes; j++)
            for (k = 0; k < nAddresses; k++)
              nextHop = Find_NoWindowing(testData[k], table);

          ClockOff( );
          time[i] = GetTime( );
          }
        }
      else    /* (useInline) */
        {
        for (i = 0; i < nExperiments; i++)
          {
          ClockOn( );

          for (j = 0; j < nTimes; j++)
            for (k = 0; k < nAddresses; k++)
              {
              s = testData[k];
              FIND_NO_WINDOWING(s, table, nextHop);
              }

          ClockOff( );
          time[i] = GetTime( );
          }
        }
      }
      /* ———————————— Compute and print statistics about execution times */
    if (verbose)
```

```
    {
    fprintf(out, "     Time per experiment: ");
    for (i = 0; i < nExperiments - 1; i++)
      fprintf(out, "%.2f, ", time[i]);
    fprintf(out, "%.2f", time[i]);
    fprintf(out, "\n");
    }
  minTime = time[0];
  timeSum = time2Sum = 0;
  for (i = 0; i < nExperiments; i++)
    {
    minTime = time[i] < minTime ? time[i] : minTime;
    timeSum += time[i];
    time2Sum += time[i] * time[i];
    }
  if (nExperiments > 1)
    {
    averTime = timeSum / (double) nExperiments;
    stdevTime = sqrt(fabs(time2Sum - nExperiments * averTime * averTime) /
        (double)(nExperiments - 1));
    fprintf(out, "     Minimum: %.2f sec\n", minTime);
    fprintf(out, "     Average: %.2f sec\n", averTime);
    fprintf(out, "     St.Dev.: %.2f sec\n", stdevTime);
    }
  fprintf(out, "     ");
  PRINT_WORD(out, (WORD) nTimes * nAddresses / minTime, 0);
  fprintf(out, " lookups/sec");
  if (verbose)
    {
    fprintf(out, "   (= ");
    PRINT_WORD(out, nTimes * nAddresses, 0);
    fprintf(out, " lookups / %.2f sec)", minTime);
    }
  fprintf(out, "\n");
    // ———————————————— Compute memory accesses
  if (getMemoryAccesses)
    {
    double sumNASTAccesses;   // ∑ # of accesses to the AST
    double sumNAnchorAccesses;   // ∑ # of accesses to the anchor table
    double sumNBaseAccesses;   // ∑ # of accesses to the base vector
    double sumNNextHopAccesses;   // ∑ # of accesses to the next-hop table
  #define MAX_N_TIMES 7
    int nASTTimes[MAX_N_TIMES];   // # of times the AST is accessed
    int nAnchorTimes[MAX_N_TIMES];   // # of times the anchor table is accessed
    int nBaseTimes[MAX_N_TIMES];   // # of times the base vector is accessed
    int nNextHopTimes[MAX_N_TIMES];   // # of times the nexthop table is accessed
    int nDefaultNextHops;   // # of times the default next-hop is returned
    int nASTAccesses;   // # of accesses to the AST in a search
    int nAnchorAccesses;   // # of accesses to the anchor table in a search
    int nBaseAccesses;   // # of accesses to the base vector in a search
    int nNextHopAccesses;   // # of accesses to the next-hop table in a search
```

```
  // ——————————————— Initialize counters
sumNASTAccesses = 0.0;
sumNAnchorAccesses = 0.0;
sumNBaseAccesses = 0.0;
sumNNextHopAccesses = 0.0;
for (i = 0; i < MAX_N_TIMES; i++)
  {
  nASTTimes[i] = 0;
  nAnchorTimes[i] = 0;
  nBaseTimes[i] = 0;
  nNextHopTimes[i] = 0;
  }
nDefaultNextHops = 0;
  // ——————————————— Count accesses
for (k = 0; k < nAddresses; k++)
  {
  nextHop = StatisticFind(testData[k], table, &nASTAccesses, &nAnchorAccesses,
      &nBaseAccesses, &nNextHopAccesses);
  sumNASTAccesses += nASTAccesses;
  sumNAnchorAccesses += nAnchorAccesses;
  sumNBaseAccesses += nBaseAccesses;
  sumNNextHopAccesses += nNextHopAccesses;
  nASTTimes[MIN(nASTAccesses, MAX_N_TIMES − 1)]++;
  nAnchorTimes[MIN(nAnchorAccesses, MAX_N_TIMES − 1)]++;
  nBaseTimes[MIN(nBaseAccesses, MAX_N_TIMES − 1)]++;
  nNextHopTimes[MIN(nNextHopAccesses, MAX_N_TIMES − 1)]++;
  if (nextHop ≡ DEFAULT_NEXT_HOP)
    nDefaultNextHops++;
  }
  // ——————————————— Output values
fprintf(out, "␣␣␣␣␣Search␣statistics:␣␣␣␣␣␣␣␣␣␣␣␣average");
for (i = 0; i < MAX_N_TIMES − 1; i++)
  fprintf(out, "␣␣%dtime%s", i, (i ≠ 1) ? "s" : "␣");
fprintf(out, "␣␣>%dtimes", MAX_N_TIMES − 2);
fprintf(out, "\n␣␣␣␣␣␣␣␣␣␣accesses␣to␣AST␣␣␣␣␣␣␣␣␣␣␣␣␣%4.2f␣␣␣␣␣␣␣␣",
    sumNASTAccesses / (double) nAddresses);
for (i = 1; i < MAX_N_TIMES; i++)
  fprintf(out, "␣%6.2f%%", nASTTimes[i] / (double) nAddresses ∗ 100.0);
fprintf(out, "\n␣␣␣␣␣␣␣␣␣␣accesses␣to␣Anchor␣table␣␣␣␣%4.2f",
    sumNAnchorAccesses / (double) nAddresses);
for (i = 0; i < MAX_N_TIMES; i++)
  fprintf(out, "␣%6.2f%%", nAnchorTimes[i] / (double) nAddresses ∗ 100.0);
fprintf(out, "\n␣␣␣␣␣␣␣␣␣␣accesses␣to␣Base␣vector␣␣␣␣␣%4.2f",
    sumNBaseAccesses / (double) nAddresses);
for (i = 0; i < MAX_N_TIMES; i++)
  fprintf(out, "␣%6.2f%%", nBaseTimes[i] / (double) nAddresses ∗ 100.0);
fprintf(out, "\n␣␣␣␣␣␣␣␣␣␣accesses␣to␣Next-hop␣table␣␣%4.2f",
    sumNNextHopAccesses / (double) nAddresses);
for (i = 0; i < 2; i++)
  fprintf(out, "␣%6.2f%%", nNextHopTimes[i] / (double) nAddresses ∗ 100.0);
fprintf(out, "\n␣␣␣␣␣␣␣␣␣␣number␣of␣default␣next-hops␣%4.2f␣%6.2f%%␣%6.2f%%\n",
```

```
            (double) nDefaultNextHops / (double) nAddresses,
            (double)(nAddresses − nDefaultNextHops) / (double) nAddresses ∗ 100.0,
            (double) nDefaultNextHops / (double) nAddresses ∗ 100.0);
    }
}
```

[statistics.web]

Internal functions

Internal function: *TraverseAST*( )

Description: It recursively traverse an AST and get information.

Important: The first call of this function should be something like

TraverseAST(*ast*, *anchorTable*, *ast*[0], 0, *MAX_IP_ADDRESS*, 0, &*sumLeafDepths*, &*maxDepth*, *nLeavesPerLevel*, &*sumDepthTimesWeight*, &*totalWeight*, &*nFullL*, &*nSpecialL*, &*nCoveredEL*, &*nUncoveredEL*);

where before this call *sumLeafDepths*, *maxDepth*, *nLeavesPerLevel*[*i*] $\forall i$, *sumDepthTimesWeight*, *totalWeight*, *nFullL*, *nSpecialL*, *nCoveredEL* and *nUncoveredEL* should be 0.

Parameters:

t	←	routing table
root	←	root of the current examined sub-AST
left	←	left extreme of the interval
right	←	right extreme of the interval
currDepth	←	depth of the current examined level of the AST
sumLeafDepths	↔	sum of the depths of all the leaves in the AST
maxDepth	↔	maximum depth of the AST
nLeavesPerLevel	↔	number of leaves at each level
sumDepthTimesWeight	↔	$\sum$ of the depth of the leaves multiplied by the weight
totalWeight	↔	$\sum$ of weights of leaves
nFullL	↔	# of full leaves
nSpecialL	↔	# of special leaves
nCoveredEL	↔	# of covered empty leaves
nUncoveredEL	↔	# of uncovered empty leaves

Return value: None.

"statistics.c" 15.2.6 ≡

```
static void TraverseAST(const AST_NODE ast[ ], const ANCHOR anchorTable[ ],
        AST_NODE root, IP_ADDRESS left, IP_ADDRESS right, int currDepth,
        WORD ∗sumLeafDepths, int ∗maxDepth, WORD nLeavesPerLevel[ ],
        double ∗sumDepthTimesWeight, double ∗totalWeight, WORD ∗nFullL,
        WORD ∗nSpecialL, WORD ∗nCoveredEL, WORD ∗nUncoveredEL)
    {
    double weight;    // weight of this leaf
    ANCHOR ∗anchorPtr;    // pointer to the prefix in base[ ] chosen as anchor
    IP_ADDRESS anchor;    // point chosen as anchor
    WORD anchorIndex;    // AST index of the child containing the anchor
    IP_ADDRESS step;    // length of a regular sub-interval (bucket)
    int leftWindowWidth;    /∗ # of nodes on the left after which there are only empty leaves ∗/
    int rightWindowWidth;    /∗ # of nodes on the right after which there are only empty leaves ∗/
    IP_ADDRESS subLeft;    // left extreme of a sub-interval (bucket)
```

```
IP_ADDRESS subRight;    // right extreme of a sub-interval (bucket)
long i;    // index
  // ———————————— If the root is an internal node ...
if (IS_INTERNAL_NODE(root))
  {
    // ———————————— Get the anchor
  anchorPtr = (ANCHOR *) &(anchorTable[GET_ANCHOR_INDEX(root)]);
  anchor = anchorPtr→point;
  anchorIndex = GET_AST_INDEX(anchorPtr→fields);

    /* ———————————— Get step, leftWindowWidth and rightWindowWidth */
  step = 1 << GET_LOG_STEP(anchorPtr→fields);
  leftWindowWidth = GET_LEFT_WINDOW_WIDTH(root);
  rightWindowWidth = GET_RIGHT_WINDOW_WIDTH(root);

    // ———————————— Sub-intervals after the anchor
  i = 0;

  WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT(anchor, right, step,
        subLeft, subRight)
    {
    TraverseAST(ast, anchorTable, ast[anchorIndex + i], subLeft, subRight, currDepth + 1,
        sumLeafDepths, maxDepth, nLeavesPerLevel, sumDepthTimesWeight, totalWeight,
        nFullL, nSpecialL, nCoveredEL, nUncoveredEL);

    i++;
    if ((rightWindowWidth ≠ 0) ∧ (i ≥ rightWindowWidth))
      break;

    } END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT(anchor, right, step,
        subLeft, subRight);

    // ———————————— Sub-intervals before the anchor
  i = −1;

  WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT(anchor, left, step, subLeft, subRight)
    {
    TraverseAST(ast, anchorTable, ast[anchorIndex + i], subLeft, subRight, currDepth + 1,
        sumLeafDepths, maxDepth, nLeavesPerLevel, sumDepthTimesWeight, totalWeight,
        nFullL, nSpecialL, nCoveredEL, nUncoveredEL);

    i−−;
    if ((leftWindowWidth ≠ 0) ∧ (−i > leftWindowWidth))
      break;

    } END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT(anchor, left, step,
        subLeft, subRight);
  }
  // ———————————— If the root is a leaf ...
else    /* (IS_FULL_OR_SPECIAL_LEAF(root) ∨ IS_EMPTY_LEAF(root)) */
  {
  *sumLeafDepths += currDepth;
  if (currDepth > *maxDepth)
    *maxDepth = currDepth;
  nLeavesPerLevel[currDepth]++;

  weight = (double) right − (double) left + 1.0;
```

```
    *sumDepthTimesWeight += weight * currDepth;
    *totalWeight += weight;
    if (IS_FULL_LEAF(root))
      (*nFullL)++;
    else if (IS_SPECIAL_LEAF(root))
      (*nSpecialL)++;
    else if (IS_COVERED_EMPTY_LEAF(root))
      (*nCoveredEL)++;
    else    /* (IS_UNCOVERED_EMPTY_LEAF(root)) */
      (*nUncoveredEL)++;
    }
  }
```

[statistics.web]

16 Testing the routing table based on the AST

[ast_test.web]

This section contains the implementation of a program to test the routing table based on the AST.

"statistics.c" 16 ≡
```
@O ast_test.c
```

[ast_test.web]

Includes

"ast_test.c" 16.1 ≡

```
#include <argp.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "basic_defs.h"
#include "check.h"
#include "build_rt.h"
#include "entry.h"
#include "parameters.h"
#include "routing_table.h"
#include "statistics.h"
```

[ast_test.web]

Internal constants

"ast_test.c" 16.2 ≡
```
    // ———————————— Max # of entries of the routing table
#define MAX_ENTRIES 100000

    // ———————————— Max # of entries of the traffic file
#define MAX_TRAFFIC 1000000

    // ———————————— Define it as TRUE to prints messages
#define VERBOSE TRUE
```

[ast_test.web]

Internal prototypes

"ast_test.c" 16.3 ≡
```
static error_t Parser(int key, char *arg, struct argp_state *state);
static char *ParametersHelp(int key, const char *text, void *input);
```

```
static void SetDefaultParameters(PARAMETERS *parameters);
static void CheckParameters(const PARAMETERS *parameters);
static WORD ReadTraffic(const char *fileName, IP_ADDRESS traffic[ ], WORD maxAddresses);
```

[ast_test.web]

Internal global variables

"ast_test.c" 16.4 ≡

```
  // ———————————— Program version
const char *argp_program_version = "ast_test␣1.0";

  // ———————————— Bug-reporting address
const char *argp_program_bug_address = "<giordano.fusco@iit.cnr.it>";

  // ———————————— The options we understand
  /*@-nullassign@*/
static const struct argp_option options[ ] = {
{NULL, 0, NULL, 0, "Parameters␣of␣AST␣construction:", 1},
    {"log-root-branch", 'b', "NUM", 0, "Logarithm␣of␣the␣branching␣factor␣at␣root"},
    {"c", 'c', "NUM", 0, "c␣for␣each␣level␣where␣a␣different␣value␣is␣not␣specified"},
    {"c0", 1000, "NUM", 0, "c␣at␣level␣0"},
    {"c1", 1001, "NUM", 0, "c␣at␣level␣1"},
    {"c2", 1002, "NUM", 0, "c␣at␣level␣2"},
    {"c3", 1003, "NUM", 0, "c␣at␣level␣3"},
    {"c4", 1004, "NUM", 0, "c␣at␣level␣4"},
    {"c5", 1005, "NUM", 0, "c␣at␣level␣5"},
    {"c6", 1006, "NUM", 0, "c␣at␣level␣6"},
    {"c7", 1007, "NUM", 0, "c␣at␣level␣7"},
    {"c8", 1008, "NUM", 0, "c␣at␣level␣8"},
    {"c9", 1009, "NUM", 0, "c␣at␣level␣9"},
    {"max-special-leaf", 'm', "N", 0, "Maximum␣number␣of␣points␣in␣a␣special␣leaf"},
    {"disable-windowing", 'w', NULL, 0, "Disable␣the␣windowing"},

 {NULL, 0, NULL, 0, "Parameters␣of␣test␣program:", 2},
    {"print-entries", 'e', NULL, 0, "Print␣routing␣table␣entries"},
    {"print-routing-table", 'r', NULL, 0, "Print␣built␣routing␣table"},
    {"verify-correctness", 'v', NULL, 0,
    "Verify␣correctness␣of␣Find()␣function␣(this␣is␣very␣slowly)"},
    {"inline-search", 'i', NULL, 0, "Perform␣also␣inline␣search"},
    {"n-repeat-search", 'n', "NUM", 0,
    "Number␣of␣times␣to␣repeat␣the␣search␣of␣the␣IP␣addresses␣in␣a␣test␣experiment"},

 {NULL, 0, NULL, 0, "Miscellaneous:", −1},
    {0}
    };    /*@=nullassign@*/

  // ———————————— Non-option arguments
static const char args_doc[ ] = "routing_file␣[traffic_file]";

  // ———————————— Program documentation
static const char doc[ ] =
    "Implementation␣of␣a␣routing␣table␣based␣on␣Adaptive␣Stratified␣Trees";

  // ———————————— argp parser
```

```
static const struct argp argp = {options, Parser, args_doc, doc, NULL, ParametersHelp};
```

[ast_test.web]

main( )

Function: *main*( )

Description: This is the main function of the test program.

Important: Once compiled, type `ast_test --help` or `ast_test --usage` for more information.

Parameters:
argc ← # of arguments
argv ↔ value of the arguments

Return value:
0 ≡ no errors
−1 ≡ an error occurred

"ast_test.c" 16.5 ≡

```
int main(int argc, char *argv[])
  {
  PARAMETERS parameters;    // parameters
  ENTRY entry[MAX_ENTRIES];    // entries of the routing table
  WORD nEntries;    // # of entries of the routing table
  IP_ADDRESS traffic[MAX_TRAFFIC];    // IP addresses of the traffic
  WORD nTraffic;    // # of IP addresses of the traffic
  IP_ADDRESS *testData;    /* test data: it comes from either a traffic file,
      or it is generated from the routing table */
  WORD i, j;    // indexes to scan entries
  IP_ADDRESS tempAddress;    // temporary IP address used for swapping
  ROUTING_TABLE table;    // routing table

    // ———————————— Read parameters
  argp_parse(&argp, argc, argv, 0, NULL, &parameters);

    // ———————————— Read entries from the routing file
  if (parameters.routing_file ≡ NULL)
    {
    fprintf(stderr, "%s:␣Routing␣file␣not␣specified.\n", argv[0]);
    return −1;
    }

  if ((nEntries = ReadEntries(parameters.routing_file, entry, MAX_ENTRIES)) ≡ (WORD) −1)
    {
    fprintf(stderr, "%s:␣Input␣file␣too␣large.\n", argv[0]);
    return −1;
    }

  fprintf(stderr, "Table␣file:␣%s␣␣␣(", parameters.routing_file);
  PRINT_WORD(stderr, nEntries, 0);
  fprintf(stderr, "␣lines)\n");

  if (parameters.printEntries)
    PrintEntries(stdout, entry, nEntries, nEntries);

    // ———————————— Prepare the test data ...
```

```
if (parameters.traffic_file ≠ NULL)
  {
    // ———————————— Read the traffic from a file
  nTraffic = ReadTraffic(parameters.traffic_file, traffic, MAX_TRAFFIC);

  fprintf(stderr, "Traffic␣file:␣%s␣␣␣(", parameters.traffic_file);
  PRINT_WORD(stderr, nTraffic, 0);
  fprintf(stderr, "␣lines)\n");

  testData = traffic;
  }
else    /* (parameters.traffic_file ≡ NULL) */
  {
    /* ———————————— Use data from routing table as traffic */
  testData = (IP_ADDRESS *) calloc(nEntries, sizeof(IP_ADDRESS));
  CHECK_MEMORY_ALLOCATION(testData);

  for (i = 0; i < nEntries; i++)
    testData[i] = entry[i]→string;

    // ———————————— Permute the entries
  srand48(getpid());
  for (i = 0; i < nEntries − 1; i++)
    {
    j = i + (WORD)(drand48() * (nEntries − i − 1));

    tempAddress = testData[i];
    testData[i] = testData[j];
    testData[j] = tempAddress;
    }

  nTraffic = nEntries;
  }

if (parameters.nRepeatSearch > 1)
  fprintf(stderr, "Repeated:␣%ld\n", parameters.nRepeatSearch);

  // ———————————— Build the routing table
table = BuildRoutingTable(stderr, entry, &nEntries, &parameters, VERBOSE);

if (parameters.printRoutingTable)
  PrintRoutingTable(stdout, table);

  /* ———————————— Print statistics about the routing table */
PrintRoutingTableStatistics(stderr, table, &parameters);

  // ———————————— Verify correctness of Find()
if (parameters.verifyCorrectness)
  {
  VerifyFindCorrectness(stdout, entry, nEntries, table, FALSE, &parameters);

  if (parameters.inlineSearch)
    VerifyFindCorrectness(stdout, entry, nEntries, table, TRUE, &parameters);
  }

  // ———————————— Run the search test
if (parameters.traffic_file ≠ NULL)
  fprintf(stderr, "\nTest:␣search␣the␣entries␣from␣the␣traffic␣file\n");
else
```

```
  fprintf(stderr, "\nTest: search a permutation of the entries of the table file\n");

fprintf(stderr, "Function search:\n");
RunSearchTest(stderr, testData, nTraffic, table, &parameters, FALSE, 8, TRUE, VERBOSE);

if (parameters.inlineSearch)
  {
  fprintf(stderr, "Inline search:\n");
  RunSearchTest(stderr, testData, nTraffic, table, &parameters, TRUE, 8, FALSE, VERBOSE);
  }
  // ———————————— Dispose routing table and entries
DisposeRoutingTable(table);
DisposeEntries(entry, nEntries);

  // ———————————— Exit without errors
return 0;
}
```

[ast_test.web]

Internal functions

Internal function: *Parser*( )

Description: It parses a single command line option.

Parameters:

key	←	key of option to parse
arg	↔	value of the option
state	↔	information about the current parsing state

Return value: 0 on success or *ARGP_ERR_UNKNOWN* if the value of *key* is not handled by this parser function.

"ast_test.c" 16.6 ≡

```
static error_t Parser(int key, char *arg, struct argp_state *state)
  {
  PARAMETERS *parameters = (PARAMETERS *) state→input;    // parameters
  static BOOL cDifferent[MAX_AST_LEVELS];    /* ∀ level, is c different specified? */
  int i;    // index to scan level

  switch (key)
    {
      // ———————————— Before parsing any argument
  case ARGP_KEY_INIT:
    SetDefaultParameters(parameters);
    for (i = 0; i < MAX_AST_LEVELS; i++)
      cDifferent[i] = FALSE;
    break;

      // ———————————— Parameters of AST construction:
      // ———————————— log2(brancing factor at root)
  case 'b':
    parameters→logRootBranch = strtol(arg, (char **) NULL, 10);
```

```
  break;
    // ———————————— c where not different specified
case 'c' :
  for (i = 0; i < MAX_AST_LEVELS; i++)
    if (¬cDifferent[i])
      parameters→c[i] = strtod(arg, (char **) NULL);
  break;
    // ———————————— c level by level
case 1000 :
case 1001 :
case 1002 :
case 1003 :
case 1004 :
case 1005 :
case 1006 :
case 1007 :
case 1008 :
case 1009 :
  parameters→c[key − 1000] = strtod(arg, (char **) NULL);
  cDifferent[key − 1000] = TRUE;
  break;
    // ———————————— Max # of points in a special leaf
case 'm' :
  parameters→maxSpecialLeaf = strtol(arg, (char **) NULL, 10);
  break;
    // ———————————— Disable the windowing
case 'w' :
  parameters→disableWindowing = TRUE;
  break;
    // ———————————— Parameters of test program:
    // ———————————— Print routing table entries
case 'e' :
  parameters→printEntries = TRUE;
  break;
    // ———————————— Print built routing table
case 'r' :
  parameters→printRoutingTable = TRUE;
  break;
    // ———————————— Verify correctness of Find( )
case 'v' :
  parameters→verifyCorrectness = TRUE;
  break;
    // ———————————— Perform also inline search
case 'i' :
  parameters→inlineSearch = TRUE;
```

```
    break;
      // ———————————— # of times to repeat the search
  case 'n':
    parameters→nRepeatSearch = strtol(arg, (char **) NULL, 10);
    break;
      // ———————————— Non-option arguments
  case ARGP_KEY_ARG:
      /* Note: state→arg_num is the number of non-optional arguments found up now. */
    if (state→arg_num ≥ 2)    /* Too many arguments */
      argp_usage(state);
    if (parameters→routing_file ≡ NULL)
      parameters→routing_file = arg;
    else
      parameters→traffic_file = arg;
    break;
      // ———————————— End of arguments
  case ARGP_KEY_END:
      /* ———————————— If the routing_file is missing ... */
    if (state→arg_num < 1)
      argp_usage(state);
      // ———————————— Check parameters
    CheckParameters(parameters);
    break;
      // ———————————— Unknown option
  default:
    return ARGP_ERR_UNKNOWN;
    }
  return 0;
  }
```

[ast_test.web]

Internal function: *ParametersHelp*( )

Description: It adds the text `(default = ...)` at the end of the text describing each parameter.

Parameters:

key	←	key of option to modify help
text	←	default help text
input	↔	*input* supplied to *argp_parse*

Return value: The modified help text to be printed.

"ast_test.c" 16.7 ≡

```
  static char *ParametersHelp(int key, const char *text, void *input)
  {
  PARAMETERS *parameters = (PARAMETERS *) input;    // parameters
  char *newText;    // new text to print
```

```
switch (key)
  {
    // ———————————— log2(brancing factor at root)
case 'b' :
  newText = malloc((strlen(text) + 30) * sizeof(char));
  CHECK_MEMORY_ALLOCATION(newText);
  sprintf(newText, "%s␣(default␣=␣%d)", text, parameters→logRootBranch);
  break;
    // ———————————— c where not different specified
case 'c' :
  newText = malloc((strlen(text) + 30) * sizeof(char));
  CHECK_MEMORY_ALLOCATION(newText);
  sprintf(newText, "%s␣(default␣=␣%.3f)", text, parameters→c[MAX_AST_LEVELS − 1]);
  break;
    // ———————————— c level by level
case 1000 :
case 1001 :
case 1002 :
case 1003 :
case 1004 :
case 1005 :
case 1006 :
case 1007 :
case 1008 :
case 1009 :
  newText = malloc((strlen(text) + 30) * sizeof(char));
  CHECK_MEMORY_ALLOCATION(newText);
  sprintf(newText, "%s␣(default␣=␣%.3f)", text, parameters→c[key − 1000]);
  break;
    // ———————————— Maximum number of special leaves
case 'm' :
  newText = malloc((strlen(text) + 30) * sizeof(char));
  CHECK_MEMORY_ALLOCATION(newText);
  sprintf(newText, "%s␣(default␣=␣%d)", text, parameters→maxSpecialLeaf);
  break;
    // ———————————— # of times to repeat the search
case 'n' :
  newText = malloc((strlen(text) + 30) * sizeof(char));
  CHECK_MEMORY_ALLOCATION(newText);
  sprintf(newText, "%s␣(default␣=␣%ld)", text, parameters→nRepeatSearch);
  break;
    // ———————————— All other help strings
default:
  newText = (char *) text;
  }
  // ———————————— Return new text
return newText;
```

```
    }
```

[ast_test.web]

Internal function: *SetDefaultParameters*( )	
Description: It sets the default parameters.	
Parameters:	
parameters ↔	parameters
Return value: None.	

"ast_test.c" 16.8 ≡

```
static void SetDefaultParameters(PARAMETERS *parameters)
  {
  int i;

    // ———————————— Parameters of AST construction
  parameters→logRootBranch = DEFAULT_LOG_ROOT_BRANCH;
  for (i = 0; i < MAX_AST_LEVELS; i++)
    {
    parameters→c[i] = DEFAULT_C;
    }
  parameters→maxSpecialLeaf = DEFAULT_MAX_SPECIAL_LEAF;
  parameters→disableWindowing = DEFAULT_DISABLE_WINDOWING;

    // ———————————— Parameters of test program
  parameters→printEntries = DEFAULT_PRINT_ENTRIES;
  parameters→printRoutingTable = DEFAULT_PRINT_ROUTING_TABLE;
  parameters→verifyCorrectness = DEFAULT_VERIFY_CORRECTNESS;
  parameters→inlineSearch = DEFAULT_INLINE_SEARCH;
  parameters→nRepeatSearch = DEFAULT_N_REPEAT_SEARCH;

    // ———————————— Routing and traffic files
  parameters→routing_file = DEFAULT_ROUTING_FILE;
  parameters→traffic_file = DEFAULT_TRAFFIC_FILE;
  }
```

[ast_test.web]

Internal function: *CheckParameters*( )	
Description: It checks parameters.	
Parameters:	
parameters ←	parameters
Return value: None.	

"ast_test.c" 16.9 ≡

```
static void CheckParameters(const PARAMETERS *parameters)
  {
  int i;

    // ———————————— Parameters of AST construction
  if (parameters→logRootBranch < MIN_LOG_ROOT_BRANCH)
```

```
    {
    fprintf(stderr, "log-root-branch␣should␣be␣greater␣or␣equal␣to␣%i.\n",
        MIN_LOG_ROOT_BRANCH);
    exit(−1);
    }
  for (i = 0; i < MAX_AST_LEVELS; i++)
    {
    if (parameters→c[i] < MIN_C)
      {
      fprintf(stderr, "c␣should␣be␣greater␣or␣equal␣to␣%.3f.\n", MIN_C);
      exit(−1);
      }
    }
  if ((parameters→maxSpecialLeaf < MIN_MAX_SPECIAL_LEAF) ∨ (parameters→maxSpecialLeaf >
        MAX_MAX_SPECIAL_LEAF))
    {
    fprintf(stderr, "max-special-leaf␣should␣be␣a␣number␣form␣%i␣to␣%i.\n",
        MIN_MAX_SPECIAL_LEAF, MAX_MAX_SPECIAL_LEAF);
    exit(−1);
    }
    // ———————————— Parameters of test program
  if (parameters→nRepeatSearch < MIN_N_REPEAT_SEARCH)
    {
    fprintf(stderr, "n-repeat-search␣should␣be␣greater␣or␣equal␣to␣%i.\n",
        MIN_N_REPEAT_SEARCH);
    exit(−1);
    }
  }
```

[ast_test.web]

Internal function: *ReadTraffic*( )		
Description: Read traffic from a file.		
Note: Each address is an unsigned number representing the bit pattern.		
Parameters:		
fileName	←	name of the traffic file
traffic	↔	array to store the addresses of the traffic
maxAddresses	←	size of *traffic*[ ]: max # of addresses that can be read
Return value: Number of read addresses.		

"ast_test.c" 16.10 ≡

```
 static WORD ReadTraffic(const char *fileName, IP_ADDRESS traffic[ ], WORD maxAddresses)
   {
   FILE *inFile;    // pointer to the file containing the strings
   WORD nAddresses = 0;    // # of read IP addresses
   IP_ADDRESS address;    // read address
     // ———————————— Open the routing file
   if (¬(inFile = fopen(fileName, "rb")))
     {
```

```
    perror(fileName);
    exit(−1);
    }
    // ——————————————— Read the strings
  while ((fscanf(inFile, "%u", &address) ≠ EOF) ∧ (nAddresses < maxAddresses))
    {
    traffic[nAddresses] = address;
    nAddresses++;
    }
    // ——————————————— Return the number of strings
  return nAddresses;
  }
```

[ast_test.web]

17 Check correctness

[check_h.web]

Here there are some routines to check the correctness of ASTs.

17.1 Check correctness: header

[check_h.web]

"ast_test.c" 17.1 ≡

```
@O check.h
```

[check_h.web]

Guard

"check.h" 17.1.1 ≡

```
#ifndef _AST_CHECK_H_
#define _AST_CHECK_H_
```

[check_h.web]

Includes

"check.h" 17.1.2 ≡

```
#include <stdio.h>

#include "basic_defs.h"
#include "entry.h"
#include "parameters.h"
#include "routing_table.h"
```

[check_h.web]

Prototypes

"check.h" 17.1.3 ≡

```
void PrintBucketsLeftGivenStep(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
    IP_ADDRESS anchor, int logStep);
void PrintBucketsAnchorGivenStep(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
    IP_ADDRESS anchor, int logStep);
void CompareBucketsGivenStep(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
    IP_ADDRESS anchor, int logStep);
void PrintBucketsLeftGivenBranch(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
    IP_ADDRESS anchor, int logBranch);
```

```
void PrintBucketsAnchorGivenBranch(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
    IP_ADDRESS anchor, int logBranch);
void CompareBucketsGivenBranch(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
    IP_ADDRESS anchor, int logBranch);
void VerifyFindCorrectness(FILE *out, const ENTRY entry[], WORD nEntries,
    const ROUTING_TABLE t, BOOL useInline, const PARAMETERS *parameters);
```

[check_h.web]

Guard

"check.h" 17.1.4 ≡

```
#endif    // _AST_CHECK_H_
```

[check_h.web]

17.2 Check correctness: implementation

[check.web]

"check.h" 17.2 ≡
    @O check.c

[check.web]

Includes

"check.c" 17.2.1 ≡

```
#include <stdio.h>
#include <stdlib.h>

#include "basic_defs.h"
#include "bucket.h"
#include "check.h"
#include "entry.h"
#include "parameters.h"
#include "prefix.h"
#include "routing_table.h"
#include "search.h"
```

[check.web]

Internal types

"check.c" 17.2.2 ≡

```
// ———————————————— Sub-intervals
typedef struct
  {
  IP_ADDRESS subLeft;
  IP_ADDRESS subRight;
  } SUB_INTERVAL;
```

[check.web]

Functions

Function: *PrintBucketsLeftGivenStep*( )

Description: Given the extremes of an interval, the value of the anchor and the step, it prints the extremes of the buckets (sub-intervals) in which it would be subdivided. It prints the buckets from left to right.

Parameters:

out	↔	file of output
left	←	left extreme of the interval
right	←	right extreme of the interval
anchor	←	point chosen as anchor
logStep	←	$\log_2(step)$

Return value: None.

```
"check.c" 17.2.3 ≡
  void PrintBucketsLeftGivenStep(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
        IP_ADDRESS anchor, int logStep)
    {
    IP_ADDRESS step;    // length of a regular sub-interval
    IP_ADDRESS subLeft, subRight;    // extremes of the current sub-interval

      // ———————————— Compute step
    step = 1 << logStep;

      // ———————————— Print extremes of the interval
    fprintf(out, "Left␣=␣%010u,␣Right␣=␣%010u,␣Anchor␣=␣%010u,␣Step␣=␣%010u,␣", left, right,
        anchor, step);

      // ———————————— Print extremes of the sub-intervals
    fprintf(out, "sub-intervals:");

    WHILE_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, anchor, step,
          subLeft, subRight)
      {
      fprintf(out, "␣(%010u,%010u)", subLeft, subRight);

      } END_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, anchor, step,
          subLeft, subRight);

      // ———————————— Close the line
    fprintf(out, "\n");
    fflush(out);
    }
```

[check.web]

Function:	*PrintBucketsAnchorGivenStep*( )

Description: Given the extremes of an interval, the value of the anchor and the step, it prints the extremes of the buckets (sub-intervals) in which it would be subdivided. It prints the buckets starting from the anchor to the extremes.

Parameters:

out	↔	file of output
left	←	left extreme of the interval
right	←	right extreme of the interval
anchor	←	point chosen as anchor
logStep	←	$\log_2(step)$

Return value: None.

```
"check.c" 17.2.4 ≡
  void PrintBucketsAnchorGivenStep(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
        IP_ADDRESS anchor, int logStep)
    {
    IP_ADDRESS step;    // length of a regular sub-interval
    IP_ADDRESS subLeft, subRight;    // extremes of the current sub-interval

      // ———————————— Compute step and shift
    step = 1 << logStep;

      // ———————————— Print extremes of the interval
```

```
fprintf(out, "Left␣=␣%010u,␣Right␣=␣%010u,␣Anchor␣=␣%010u,␣Step␣=␣%010u,␣", left, right,
    anchor, step);
  // ———————————— Print extremes of the sub-intervals
fprintf(out, "sub-intervals:");
  // ———————————— Sub-intervals after the anchor
WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT(anchor, right, step, subLeft, subRight)
  {
  fprintf(out, "␣(%010u,%010u)", subLeft, subRight);
  } END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT(anchor, right, step,
      subLeft, subRight);
  // ———————————— Sub-intervals before the anchor
WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT(anchor, left, step, subLeft, subRight)
  {
  fprintf(out, "␣(%010u,%010u)", subLeft, subRight);
  } END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT(anchor, left, step, subLeft, subRight);
  // ———————————— Close the line
fprintf(out, "\n");
fflush(out);
}
```

[check.web]

Function: *CompareBucketsGivenStep*( )

Description: Given the extremes of an interval, the value of the anchor and the step, it computes the buckets (sub-intervals) form left to right and from the anchor to the extremes. It prints a message of error if the two sets of buckets are different.

Parameters:

out	↔	file of output
left	←	left extreme of the interval
right	←	right extreme of the interval
anchor	←	point chosen as anchor
logStep	←	$\log_2(step)$

Return value: None.

"check.c" 17.2.5 ≡

```
void CompareBucketsGivenStep(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
    IP_ADDRESS anchor, int logStep)
  {
  SUB_INTERVAL *subIntervals;   // list of sub-intervals from left to right
  IP_ADDRESS step;   // length of a regular subinterval
  IP_ADDRESS subLeft, subRight;   // extremes of the current subinterval
  int i;   // index to scan sub-intervals
  int anchor_pos = 0;   // index of the anchor in the list of sub-intervals
    // ———————————— Allocate the list of sub-intervals
  subIntervals = (SUB_INTERVAL *) calloc((((right >> logStep) − (left >> logStep)) + 10) * 2,
      sizeof(SUB_INTERVAL));
  CHECK_MEMORY_ALLOCATION(subIntervals);
```

```
  // ———————————————— Compute step
step = 1 << logStep;
  // ———————————————— Sub-intervals from left to right
i = 0;
WHILE_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, anchor, step,
      subLeft, subRight)
  {
  if (subLeft ≡ anchor)
    anchor_pos = i;
  subIntervals[i].subLeft = subLeft;
  subIntervals[i].subRight = subRight;
  i++;
  } END_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT(left, right, anchor, step,
      subLeft, subRight);
  // ———————————————— Sub-intervals from anchor to extremes:
  // ———————————————— Sub-intervals after the anchor
i = 0;
WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT(anchor, right, step, subLeft, subRight)
  {
  if ((subIntervals[anchor_pos + i].subLeft ≠ subLeft) ∨
        (subIntervals[anchor_pos + i].subRight ≠ subRight))
    {
    fprintf(out, "ERROR!␣(%010u,␣%010u)␣!=␣(%010u,␣%010u)\n", subLeft, subRight,
        subIntervals[anchor_pos + i].subLeft, subIntervals[anchor_pos + i].subRight);
    fflush(out);
    }
  i++;
  } END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT(anchor, right, step,
      subLeft, subRight)
  // ———————————————— Sub-intervals before the anchor
i= −1;
WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT(anchor, left, step, subLeft, subRight)
  {
  if ((subIntervals[anchor_pos + i].subLeft ≠ subLeft) ∨
        (subIntervals[anchor_pos + i].subRight ≠ subRight))
    {
    fprintf(out, "ERROR!␣(%010u,␣%010u)␣(%010u,␣%010u)\n", subLeft, subRight,
        subIntervals[anchor_pos + i].subLeft, subIntervals[anchor_pos + i].subRight);
    fflush(out);
    }
  i−−;
  } END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT(anchor, left, step, subLeft, subRight);
  // ———————————————— Dispose the list of sub-intervals
free(subIntervals);
}
```

[`check.web`]

Function: *PrintBucketsLeftGivenBranch*( )

Description: Given the extremes of an interval, the value of the anchor and the branching factor, it prints the extremes of the buckets (sub-intervals) in which it would be subdivided. It prints the buckets from left to right.

Parameters:

out	↔	file of output
left	←	left extreme of the interval
right	←	right extreme of the interval
anchor	←	point chosen as anchor
logBranch	←	$\log_2$(branching factor)

Return value: None.

"check.c" 17.2.6 ≡

```
void PrintBucketsLeftGivenBranch(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
        IP_ADDRESS anchor, int logBranch)
  {
  IP_ADDRESS step;     // length of a regular subinterval
  int logStep;     // log2(step)
    // ———————————— Compute step
  COMPUTE_STEP(left, right, logBranch, step, logStep);
    // ———————————— Print sub-intervals from left to right
  PrintBucketsLeftGivenStep(out, left, right, anchor, logStep);
  }
```

[`check.web`]

Function: *PrintBucketsAnchorGivenBranch*( )

Description: Given the extremes of an interval, the value of the anchor and the branching factor, it prints the extremes of the buckets (sub-intervals) in which it would be subdivided. It prints the buckets starting from the anchor to the extremes.

Parameters:

out	↔	file of output
left	←	left extreme of the interval
right	←	right extreme of the interval
anchor	←	point chosen as anchor
logBranch	←	$\log_2$(branching factor)

Return value: None.

"check.c" 17.2.7 ≡

```
void PrintBucketsAnchorGivenBranch(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
        IP_ADDRESS anchor, int logBranch)
  {
  IP_ADDRESS step;     // length of a regular sub-interval
  int logStep;     // log2(step)
    // ———————————— Compute step
  COMPUTE_STEP(left, right, logBranch, step, logStep);
    // ———————————— Print sub-intervals from anchor
```

```
    PrintBucketsAnchorGivenStep(out, left, right, anchor, logStep);
    }
```

[check.web]

Function: *CompareBucketsGivenBranch*( )

Description: Given the extremes of an interval, the value of the anchor and the branching factor, it computes the buckets (sub-intervals) form left to right and from the anchor to the extremes. It prints a message of error if the two sets of buckets are different.

Parameters:

out	↔	file of output
left	←	left extreme of the interval
right	←	right extreme of the interval
anchor	←	point chosen as anchor
logBranch	←	$\log_2$(branching factor)

Return value: None.

"check.c" 17.2.8 ≡

```
void CompareBucketsGivenBranch(FILE *out, IP_ADDRESS left, IP_ADDRESS right,
        IP_ADDRESS anchor, int logBranch)
    {
    IP_ADDRESS step;    // length of a regular sub-interval
    int logStep;    // log2(step)

      // ———————————————— Compute step
    COMPUTE_STEP(left, right, logBranch, step, logStep);

      // ———————————————— Print sub-intervals from anchor
    CompareBucketsGivenStep(out, left, right, anchor, logStep);
    }
```

[check.web]

Function: *VerifyFindCorrectness*( )

Description: It verifies the correctness of the *Find*( ) function. It calls *Find*( ) for each possible IP address and it compares the next-hop returned by *Find*( ) with a next-hop computed slowly but safely. It prints a message of error each time the next-hop returned by *Find*( ) is not correct.

Important: We assume that this function is called after *Build*( ): in this way the prefixes stored in *entry*[ ] are sorted and the routing table *t* is built correctly.

Parameters:

out	↔	file of output
entry	←	routing table entries stored in an initial simple array (and sorted)
nEntries	←	# of entries in *entry*[ ] array
t	←	routing table returned by *Build*( ) function
useInline	←	if *TRUE* the search is performed inline
parameters	←	parameters

Return value: None.

"check.c" 17.2.9 ≡

```
void VerifyFindCorrectness(FILE *out, const ENTRY entry[ ], WORD nEntries,
        const ROUTING_TABLE t, BOOL useInline, const PARAMETERS *parameters)
```

```
{
IP_ADDRESS i;    // index to scan all the possible IP addresses
WORD currEntry = 0;    // index of the current examined entry of entry[ ]
NEXT_HOP correctNextHop;    // next-hop extracted from entry[ ]
WORD j;    // auxiliary index to examine entries of entry[ ]
NEXT_HOP nextHop;    // next-hop returned by Find( )

for (i = 0; i ≤ MAX_IP_ADDRESS; i++)
  {
    // ———————————— Search the nextHop in entry[ ]
  if (PREFIX_MATCH(*(entry[currEntry]), i))    /* if it matches ... */
    {
    correctNextHop = entry[currEntry]→nextHop;

    for (j = currEntry + 1; (j < nEntries) ∧ IS_PREFIX(*(entry[currEntry]), *(entry[j])); j++)
      if (PREFIX_MATCH(*(entry[j]), i))
        correctNextHop = entry[j]→nextHop;
    }
  else    /* (¬PREFIX_MATCH(entry[currEntry], i)) */
    {
    if (currEntry ≡ nEntries − 1)    /* if there are no more entries ... */
      correctNextHop = DEFAULT_NEXT_HOP;
    else    /* if it does not match but there are other entries to examine ... */
      {
      for (j = currEntry + 1; (j < nEntries) ∧ (PREFIX_LEFT_EXTREME(*(entry[j])) ≤ i) ∧
            ¬PREFIX_MATCH(*(entry[j]), i); j++)
        ;

      if ((j < nEntries) ∧ PREFIX_MATCH(*(entry[j]), i))
        {
        currEntry = j;

        correctNextHop = entry[currEntry]→nextHop;

        for (j = currEntry + 1; (j < nEntries) ∧ IS_PREFIX(*(entry[currEntry]), *(entry[j]));
              j++)
          if (PREFIX_MATCH(*(entry[j]), i))
            correctNextHop = entry[j]→nextHop;
        }
      else
        correctNextHop = DEFAULT_NEXT_HOP;
      }
    }
    /* ———————————— Search the nextHop using FIND( ) or Find( ) */
  nextHop = 0;
  if (¬parameters→disableWindowing)
    {
    if (useInline)
      {
      FIND(i, t, nextHop);
      }
    else
      nextHop = Find(i, t);
    }
  else    /* (parameters→disableWindowing) */
```

```
        {
        if (useInline)
          {
          FIND_NO_WINDOWING(i, t, nextHop);
          }
        else
          nextHop = Find_NoWindowing(i, t);
        }
        // ———————————————— Check correctness
      if (nextHop ≠ correctNextHop)
        {
        fprintf(out, "ERROR!␣");
        if (useInline)
          fprintf(out, "FIND");
        else
          fprintf(out, "Find");
        fprintf(out, "(%u,␣t)␣returned␣%u␣instead␣of␣%u!\n", i, nextHop, correctNextHop);
        fflush(out);
        }
        // ———————————————— If we checked all the IP addresses
        /* Note: the following command avoids overflow problems in the condition of the for. */
      if (i ≡ MAX_IP_ADDRESS)
        break;
      }
    }
```

[check.web]

18 Extract routing prefixes from routing tables of IPMA

[extract.web]

Here there is the implementation of a program to extract the routing prefixes from the routing tables of IPMA (http://www.merit.edu/ipma/routing_table).

In the *HTML* pages containing the routing tables we can find lines in one of the following formates:

$$< \mathbf{B} > x1.x2.x3.x4/a1.a2.a3.a4 < /\mathbf{B} >$$

$$< \mathbf{B} > x1.x2.x3/a1.a2.a3 < /\mathbf{B} >$$

$$< \mathbf{B} > x1.x2/a1.a2 < /\mathbf{B} >$$

$$< \mathbf{B} > x1/a1 < /\mathbf{B} >$$

where $x1, x2, \ldots$ are octal values, each of which represents 8 bits of a prefix, and $a1, a2, \ldots$ is a bit-mask.

After each routing prefix there is a field, starting with "$N =$", which contains the next-hop. Some prefixes have associated more than one next-hop: we consider only the first one.

For each one of the lines formatted as above, this program outputs three numbers

$$i \quad j \quad k$$

where i is the decimal representation of $x1.x2.x3.x4$, j is the number of bits in the mask and k is the decimal representation of the next-hop.

"check.c" 18 ≡

```
@O extract.c
```

[extract.web]

Includes

"extract.c" 18.1 ≡

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "basic_defs.h"
```

[extract.web]

Internal constants

"extract.c" 18.2 ≡

```
    // ———————————— Max # of prefixes that can be read
#define MAX_PREFIXES 75000

    // ———————————— Max # of chars in an input file line
#define MAX_LINE 255
```

[extract.web]

Internal prototypes

"extract.c" 18.3 ≡

```
static long ReadRoutingPrefixes(const char *fileName, WORD string[ ], int length[ ],
    WORD nextHop[ ], int maxSize);
void Translate(char *s, WORD *decimal, int *length);
void PrintRoutingPrefixes(FILE *out, const WORD string[ ], const int length[ ],
    const WORD nextHop[ ], long nPrefixes);
```

[extract.web]

main( )

Function: *main*( )

Description: This is the main function of the `extract` program.

Important: Usage:
`extract file_name`

Parameters:

argc	←	# of arguments
argv	↔	value of the arguments

Return value:

0	≡	no errors
−1	≡	an error occurred

"extract.c" 18.4 ≡

```
int main(int argc, char *argv[ ])
  {
  long nPrefixes;    // # of routing prefixes
  static WORD string[MAX_PREFIXES];    // strings of bits of the routing prefixes
  static int length[MAX_PREFIXES];    // lengths of the routing prefixes
  static WORD nextHop[MAX_PREFIXES];    // next-hops of the routing prefixes

    // ———————————— Check parameters
  if (argc ≠ 2)
    {
    fprintf(stderr, "Usage:␣%s␣file_name\n", argv[0]);
    return −1;
    }

  fprintf(stderr, "File:␣%s\n", argv[1]);

    // ———————————— Read routing prefixes
  nPrefixes = ReadRoutingPrefixes(argv[1], string, length, nextHop, MAX_PREFIXES);

    // ———————————— Print routing prefixes
  if (nPrefixes ≥ 0)
    {
    fprintf(stderr, "Strings:␣%ld\n", nPrefixes);

    PrintRoutingPrefixes(stdout, string, length, nextHop, nPrefixes);

    return 0;
```

```
      }
    else
      {
      fprintf(stderr, "Input␣file␣too␣large.\n");
      return −1;
      }
    }
```

[extract.web]

Internal functions

Internal function: *ReadRoutingPrefixes*( )

Description: It reads the routing prefixes from a file.

Parameters:

fileName	←	name of the file containing routing prefixes
string	↔	array to store the strings of bits of the routing prefixes
length	↔	array to store the lengths of the routing prefixes
nextHop	↔	array to store the next-hops of the routing prefixes
maxSize	←	size of *string*[ ], *length*[ ] and *nextHop*[ ]: max # of prefixes that can be read

Return value: Number of read prefixes or −1 if the number of prefixes exceeds *maxSize*.

"extract.c" 18.5 ≡

```
  static long ReadRoutingPrefixes(const char *fileName, WORD string[ ], int length[ ],
        WORD nextHop[ ], int maxSize)
    {
    FILE *inFile;    // pointer to the file containing prefixes
    char inputLine[MAX_LINE];    // content of a line of the input file
    long nPrefixes = 0;    // # of read prefixes
    char *posNextHop;    // position of the next-hop in the input line
    WORD decimal;    // x1.x2.x3.x4 translated in decimal notation
    int nBits;    // # of bits in the mask

      // ———————————————— Open file containing routing prefixes
    if (¬(inFile = fopen(fileName, "rb")))
      {
      perror(fileName);
      exit(−1);
      }

      // ———————————————— Read the prefixes
    while (fgets(inputLine, MAX_LINE, inFile) ≠ NULL)
      {
      if (nPrefixes ≥ maxSize)
        return −1;

      if (strncmp(inputLine, "<B>", 3) ≡ 0)
        {
        Translate(inputLine + 3, &decimal, &nBits);

        string[nPrefixes] = decimal;
        length[nPrefixes] = nBits;
```

```
        fgets(inputLine, MAX_LINE, inFile);
        posNextHop = strstr(inputLine, "N=");
        if (posNextHop ≡ NULL)
          fprintf(stderr, "FORMAT␣ERROR␣IN␣INPUT\n");
        else
          {
          Translate(posNextHop + 2, &decimal, &nBits);
          nextHop[nPrefixes] = decimal;

          nPrefixes++;
          }
        }
      }
    // ——————————————— Return the number of read prefixes
  return nPrefixes;
  }
```

[extract.web]

Internal function: *PrintRoutingPrefixes*( )

Description: It gets a string containing $x1.x2.x3.x4/a1.a2.a3.a4$ and output the decimal representation of $x1.x2.x3.x4$ and the length of the bit mask.

Parameters:

s	↔	string containing $x1.x2.x3.x4/a1.a2.a3.a4$
decimal	→	decimal representation of $x1.x2.x3.x4$
length	→	length of the bit mask

Return value: None.

"extract.c" 18.6 ≡

```
  void Translate(char *s, WORD *decimal, int *length)
    {
    // ——————————————— Read x1.x2.x3.x4
  *decimal = 0;
  while (TRUE)
    {
    *decimal = *decimal << 8 | strtol(s, &s, 10);

    if (*s ≡ '.')
      s++;
    else
      break;
    }

  if (*s ≠ '/')
    return;

    // ——————————————— Read '/'
  s++;

    // ——————————————— Get the length of the bit mask
  *length = 0;
  while (TRUE)
    {
```

```
switch (strtol(s, &s, 10))
  {
case '377 : *length += 8; break;
case '376 : *length += 7; break;
case '374 : *length += 6; break;
case '370 : *length += 5; break;
case '360 : *length += 4; break;
case '340 : *length += 3; break;
case '300 : *length += 2; break;
case '200 : *length += 1; break;
  }
if (*s ≡ '.')
  s++;
else
  break;
}
*decimal = *decimal << (32 − *length);
}
```

[extract.web]

Internal function: *PrintRoutingPrefixes*( )

Description: It prints on *out* the translated routing prefixes.

Parameters:

out	↔	file of output
string	←	array containing the strings of bits of the routing prefixes
length	←	array containing the lengths of the routing prefixes
nextHop	←	array containing the next-hops of the routing prefixes
nPrefixes	←	# of routing prefixes

Return value: None.

"extract.c" 18.7 ≡

```
void PrintRoutingPrefixes(FILE *out, const WORD string[ ], const int length[ ],
      const WORD nextHop[ ], long nPrefixes)
  {
  long i;
  for (i = 0; i < nPrefixes; i++)
    fprintf(out, "%u␣%i␣%u\n", string[i], length[i], nextHop[i]);
  }
```

[extract.web]

19 Randomly generate a traffic file

[traffic_gen.web]

Here there is the implementation of a program to randomly generate a traffic file.

A prefix of a routing table has naturally associated a segment. For each prefix we randomly generate a number of IP addresses proportional to the length of the segment associated to it. A given percentage of IP addresses is generated in the space covered by the segments associated to the prefixes, the remainings are randomly generated in all the total universe.

"extract.c" 19 ≡

```
@O traffic_gen.c
```

[traffic_gen.web]

Includes

"traffic_gen.c" 19.1 ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "basic_defs.h"
#include "entry.h"
#include "prefix.h"
```

[traffic_gen.web]

Internal constants

"traffic_gen.c" 19.2 ≡

```
// ———————————— Max # of entries of the routing table
#define MAX_ENTRIES 100000

// ———————————— Max # of entries of the traffic file
#define MAX_TRAFFIC 1000000
```

[traffic_gen.web]

Internal prototypes

"traffic_gen.c" 19.3 ≡

```
static void GenerateTraffic(const ENTRY entry[ ], WORD nEntries, double percInside,
    IP_ADDRESS traffic[ ], WORD nTraffic);
```

[traffic_gen.web]

main( )

Function: *main*( )

Description: This is the main function of the test program.

Important: Usage:
`traffic_gen routing_file n perc_inside`

Parameters:

argc	←	# of arguments
argv	↔	value of the arguments

Return value:

0	≡	no errors
−1	≡	an error occurred

"traffic_gen.c" 19.4 ≡

```
int main(int argc, char *argv[ ])
  {
  WORD nTraffic;    // # of IP addresses of the traffic to generate
  double percInside;    /* % of IP addresses to be generated inside the space covered by segments
      associated to prefixes */
  ENTRY entry[MAX_ENTRIES];    // entries of the routing table
  WORD nEntries;    // # of entries of the routing table
  IP_ADDRESS traffic[MAX_TRAFFIC];    // IP addresses of the traffic
  WORD i;    // index

    // ———————————— Read parameters
  if (argc ≠ 4)
    {
    fprintf(stderr, "Usage: %s routing_file n perc_inside\n", argv[0]);
    return −1;
    }
  nTraffic = strtol(argv[2], (char **) NULL, 10);
  if (nTraffic > MAX_TRAFFIC)
    {
    fprintf(stderr, "At most %u IP addresses can be generated.\n", MAX_TRAFFIC);
    return −1;
    }
  percInside = strtod(argv[3], (char **) NULL);
  if ((percInside < 0.0) ∨ (percInside > 100.0))
    {
    fprintf(stderr, "The percentage of IP addresses to be generated inside\n");
    fprintf(stderr, "the space covered by the segments associated to prefixes\n");
    fprintf(stderr, "should be a number between 0 and 100.\n");
    return −1;
    }
    // ———————————— Read entries from the routing file
  if ((nEntries = ReadEntries(argv[1], entry, MAX_ENTRIES)) ≡ (WORD) −1)
    {
    fprintf(stderr, "Input file too large.\n");
    return −1;
    }
```

```
  // ————————————— Sort the entries and remove duplicates
nEntries = SortAndRemoveDuplicatesEntries(entry, nEntries);

  // ————————————— Random generate IP addresses
GenerateTraffic(entry, nEntries, percInside, traffic, nTraffic);

  // ————————————— Print genenerated IP addresses
for (i = 0; i < nTraffic; i++)
  fprintf(stdout, "%u\n", traffic[i]);

  // ————————————— Dispose entries
DisposeEntries(entry, nEntries);

  // ————————————— Exit without errors
return 0;
}
```

[traffic_gen.web]

Internal functions

Internal function: *GenerateTraffic*( )

Description: It random generates the IP addresses.

Note: A prefix of a routing table has naturally associated a segment. For each prefix we randomly generate a number of IP addresses proportional to the length of the segment associated to it. A given percentage of IP addresses is generated in the space covered by the segments associated to the prefixes, the remainings are randomly generated in all the total universe.

Note: To random generate addresses inside the space covered by the segments, we consider a space formed by the segments placed each one after another, and then within this space we generate uniform random numbers.

Parameters:

entry	←	array containing the entries of the routing table
nEntries	←	# of entries of the routing table
percInside	←	% of addresses to be generated inside the space covered by segments
traffic	→	array to store the random generated IP addresses
nTraffic	←	# of random IP addresses to generate

Return value: None.

"traffic_gen.c" 19.5 ≡

```
static void GenerateTraffic(const ENTRY entry[], WORD nEntries, double percInside,
      IP_ADDRESS traffic[], WORD nTraffic)
  {
  double totalLength;   // ∑ of lengths of segmets associated to prefixes
  WORD nTrafficInside;   /* # of addresses to be generated inside the space covered by segments */
  WORD n;   // # of generated random IP addresses
  double randNumber;   // random number
  double currPos;   /* current position inside the space of segments placed each one after another */
  WORD i, j;   // indices
  double segmentLength;   // length of a segment associated to a prefix
  IP_ADDRESS tempAddress;   // temporary IP address used for swapping

    // ————————————— Compute ∑ of lengths of prefixes
```

```
totalLength = 0.0;
for (i = 0; i < nEntries; i++)
  if (entry[i]→length ≡ 0)
    totalLength += (double) MAX_IP_ADDRESS + 1.0;
  else
    totalLength += (double) PREFIX_RIGHT_EXTREME(*(entry[i])) −
        (double) PREFIX_LEFT_EXTREME(*(entry[i])) + 1.0;

  /* ———————————— Generate IP addresses inside segments associated to prefixes */
srand48(getpid( ));
nTrafficInside = nTraffic * percInside / 100.0;
for (n = 0; n < nTrafficInside; n++)
  {
  randNumber = drand48( ) * totalLength;

  currPos = 0.0;
  for (i = 0; i < nEntries; i++)
    {
    segmentLength = (double) PREFIX_RIGHT_EXTREME(*(entry[i])) −
        (double) PREFIX_LEFT_EXTREME(*(entry[i])) + 1.0;

    if ((currPos ≤ randNumber) ∧ (randNumber < currPos + segmentLength))
      {
      traffic[n] = (double) PREFIX_LEFT_EXTREME(*(entry[i])) + (IP_ADDRESS)(randNumber −
          currPos);
      break;
      }

    currPos += segmentLength;
    }
  }
  /* ———————————— Generate the remaining IP addresses in all the universe */
for ( ; n < nTraffic; n++)
  traffic[n] = (IP_ADDRESS)(drand48( ) * ((double) MAX_IP_ADDRESS + 1.0));

  // ———————————— Permute the generated IP address
for (i = 0; i < nTraffic − 1; i++)
  {
  j = i + (WORD)(drand48( ) * (nTraffic − i − 1));

  tempAddress = traffic[i];
  traffic[i] = traffic[j];
  traffic[j] = tempAddress;
  }
}
```

[traffic_gen.web]

20 References

References

[NK 99] S. Nilsson, G. Karlsson, *IP-Address Lookup Using LC-Tries*, In *IEEE Journal on Selected Areas in Communications*, volume 17, number 6, pages 1083–1092, June 1999. `http://www.nada.kth.se/ snilsson/public/papers/router2`

[PV 01a] M. Pellegrini, G. Vecchiocattivi, *How Deep is your Search Tree?*, Technical Report IMC B4-01-12, Istituto di Matematica Computazionale del CNR, Area della Ricerca, Via Moruzzi 1, 56124 Pisa, Italy December 2001.

[PV 01b] M. Pellegrini, G. Vecchiocattivi, *Empirical study of search trees for IP address lookup*, Technical Report IMC B4-01-13, Istituto di Matematica Computazionale del CNR, Area della Ricerca, Via Moruzzi 1, 56124 Pisa, Italy December 2001.

21 Index

__anchorPtr__: 13.1.3, 13.1.4.
__basePtr__: 13.1.3, 13.1.4.
__dist__: 13.1.3.
__i__: 4.4.3, 13.1.3, 13.1.4.
__length__: 4.4.3.
__n__: 4.4.3.
__nBases__: 13.1.3, 13.1.4.
__node__: 13.1.3, 13.1.4.
__nodeIndex__: 13.1.3, 13.1.4.
__number__: 8.1.
__originalNumber__: 8.1.
__shift__: 8.3.
__windowWidth__: 13.1.3.
_AST_ANCHOR_H_: 5.3.0.1, 5.3.0.9.
_AST_AST_NODE_H_: 5.4.0.1, 5.4.6.2.
_AST_BASE_H_: 5.2.0.1, 5.2.0.5.
_AST_BASIC_DEFS_H_: 4.0.1, 4.4.4.
_AST_BUCKET_H_: 8.0.1, 8.3.2.
_AST_BUILD_AST_H_: 11.1.1, 11.1.4.
_AST_BUILD_RT_H_: 12.1.1, 12.1.4.
_AST_CHECK_H_: 17.1.1, 17.1.4.
_AST_CLOCK_H_: 14.1.1, 14.1.3.
_AST_ENTRY_H_: 6.1.1, 6.1.6.
_AST_NEXT_HOP_H_: 5.1.0.1, 5.1.0.7.
_AST_P_QUEUE_H_: 9.1.1, 9.1.8.
_AST_PARAMETERS_H_: 10.1, 10.5.
_AST_PREFIX_H_: 7.0.1, 7.2.5.
_AST_ROUTING_TABLE_H_: 5.5.0.1, 5.5.0.5.
_AST_SEARCH_H_: 13.1.1, 13.1.6.
_AST_STATISTICS_H_: 15.1.1, 15.1.4.
nextHop: 13.1.3, 13.1.4.

abort: 4.4.2, 6.2.3, 9.2.4, 9.2.5, 9.2.6, 9.2.9, 9.2.10, 9.2.11, 9.2.12, 11.2.4, 12.2.6, 12.2.9.
address: 16.10.
ANCHOR: 5.3.0.4.
anchor: 5.5.0.3, 8, 8.2, 8.2.1, 8.2.2, 8.2.3, 8.3, 8.3.1, 11, 11.2.2, 11.2.4, 11.2.5, 11.2.6, 12.2.3, 12.2.4, 12.2.5, 13, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.4, 15.2.6, 17.1.3, 17.2.3, 17.2.4, 17.2.5, 17.2.6, 17.2.7, 17.2.8.
anchor_index: 5.4.2.8.
anchor_pos: 17.2.5.
anchorIndex: 13, 15.2.6.
anchorInterval: 11.2.6.
anchorPtr: 13.2.2, 13.2.3, 13.2.4, 15.2.6.
anchorSize: 5.5.0.3, 12.2.3, 12.2.4, 15.2.4.
anchorTable: 11.1.3, 11.2.2, 11.2.3, 11.2.4, 12.2.3, 15.2.3, 15.2.6.
arg: 16.3, 16.6.
arg_num: 16.6.
argc: 16.5, 18.4, 19.4.
argp: 16.4, 16.5.
ARGP_ERR_UNKNOWN: 16.6.
ARGP_KEY_ARG: 16.6.

ARGP_KEY_END: 16.6.
ARGP_KEY_INIT: 16.6.
argp_parse: 16.5, 16.7.
argp_program_bug_address: 16.4.
argp_program_version: 16.4.
argp_state: 16.3, 16.6.
argp_usage: 16.6.
args_doc: 16.4.
argv: 16.5, 18.4, 19.4.
ast: 5.5.0.3, 11.1.3, 11.2.2, 11.2.3, 11.2.4, 12.2.3, 12.2.4, 12.2.5, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.3, 15.2.4, 15.2.6.
ast_index: 5.3.0.7.
AST_NODE: 5.3.0.5, 5.3.0.7, 5.3.0.8, 5.4, 5.4.0.5.
astLevel: 11.2.2, 11.2.4, 11.2.5.
astSize: 5.5.0.3, 12.2.3, 12.2.4, 15.2.4.
averTime: 15.2.5.

b: 11.2.5.
BASE: 5.2, 5.2.0.4.
base: 5.5.0.3, 11.1.3, 11.2.2, 11.2.3, 11.2.4, 11.2.5, 11.2.6, 12.2.2, 12.2.3, 12.2.4, 12.2.5, 12.2.8, 12.2.9, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.4, 15.2.6.
base_index: 5.4.3.4.
baseIndex: 9.1.3, 9.1.7, 9.2.4, 9.2.5.
basePtr: 13.2.2, 13.2.3, 13.2.4.
baseSize: 5.5.0.3, 12.2.3, 12.2.4, 15.2.4.
BinarySearch: 12.2.2, 12.2.9, 12.2.10.
BOOL: 4.1.
Build: 17.2.9.
BuildAST: 5.4, 11, 11.1.3, 11.2.3, 12, 12.2.3.
BuildBaseVector: 5.2, 12, 12.2.2, 12.2.3, 12.2.8.
BuildNextHopTable: 5.1, 5.2, 12, 12.2.2, 12.2.3, 12.2.6.
BuildRoutingTable: 5.2, 5.4, 5.5.0.3, 6, 12, 12.1.3, 12.2.3, 16.5.
BuildSubAST: 11, 11.2.2, 11.2.3, 11.2.4.

c: 10.4.
calloc: 4.4.2, 9.2.2, 9.2.8, 12.2.3, 12.2.6, 12.2.8, 16.5, 17.2.5.
cDifferent: 16.6.
CHECK_MEMORY_ALLOCATION: 4.4.2, 6.2.3, 9.2.2, 9.2.8, 12.2.3, 12.2.6, 12.2.8, 16.5, 16.7, 17.2.5.
CheckParameters: 16.3, 16.6, 16.9.
childFirst: 11.2.4, 11.2.6.
childN: 11.2.4, 11.2.6.
clock: 14.2.3, 14.2.4.
ClockOff: 12.2.3, 14, 14.1.2, 14.2.4, 14.2.5, 15.2.5.
ClockOn: 12.2.3, 14, 14.1.2, 14.2.3, 14.2.5, 15.2.5.
CLOCKS_PER_SEC: 14.2.5.
CompareBucketsGivenBranch: 17.1.3, 17.2.8.
CompareBucketsGivenStep: 17.1.3, 17.2.5, 17.2.8.
CompareEntries: 6.2.2, 6.2.4, 6.2.7.
CompareNextHops: 12.2.2, 12.2.6, 12.2.7.
COMPUTE_STEP: 8.1.1, 11.2.5, 17.2.6, 17.2.7, 17.2.8.
ComputeAnchorStepWindowWidth: 11, 11.2.2, 11.2.4, 11.2.5.
ComputeMinRAndMinM: 11, 11.2.2, 11.2.5, 11.2.6.
correctNextHop: 17.2.9.

currAnchor: 11.2.6.
currDepth: 15.2.3, 15.2.6.
currEntry: 17.2.9.
currIndex: 12.2.9.
currPos: 19.5.

decimal: 18.3, 18.5, 18.6.
DecreaseKeyIntervalsPQ: 9.1.7, 9.2.11, 11.2.6.
DEFAULT_C: 10.3, 16.8.
DEFAULT_DISABLE_WINDOWING: 10.3, 16.8.
DEFAULT_INLINE_SEARCH: 10.3, 16.8.
DEFAULT_LOG_ROOT_BRANCH: 10.3, 16.8.
DEFAULT_MAX_SPECIAL_LEAF: 10.3, 16.8.
DEFAULT_N_REPEAT_SEARCH: 10.3, 16.8.
DEFAULT_NEXT_HOP: 5.1.0.3, 5.4.1, 12.2.6, 13, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.5, 17.2.9.
DEFAULT_NEXT_HOP_INDEX: 5.1.0.4, 5.4.1, 11.2.4, 11.2.6, 12.2.8, 12.2.9.
DEFAULT_PRINT_ENTRIES: 10.3, 16.8.
DEFAULT_PRINT_ROUTING_TABLE: 10.3, 16.8.
DEFAULT_ROUTING_FILE: 10.3, 16.8.
DEFAULT_TRAFFIC_FILE: 10.3, 16.8.
DEFAULT_VERIFY_CORRECTNESS: 10.3, 16.8.
disableWindowing: 10.4, 11.2.4, 11.2.6, 13, 13.1.4, 13.2.3, 15.2.4, 15.2.5, 16.6, 16.8, 17.2.9.
DisposeEntries: 6.1.5, 6.2.6, 16.5, 19.4.
DisposeIntervalsPQ: 9.1.7, 9.2.13, 11.2.6.
DisposePointsPQ: 9.1.7, 9.2.7, 11.2.6.
DisposeRoutingTable: 12.1.3, 12.2.5, 16.5.
dist: 13.2.2, 13.2.4.
doc: 16.4.
drand48: 16.5, 19.5.

END_EXPLORE_BUCKETS_FROM_: 8.
END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT: 8.2.3, 15.2.6, 17.2.4, 17.2.5.
END_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT: 8.2.1, 15.2.6, 17.2.4, 17.2.5.
END_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT: 8.3.1, 11.2.4, 11.2.6, 17.2.3, 17.2.5.
entry: 6, 6.1.5, 6.2.3, 6.2.4, 6.2.5, 6.2.6, 12.1.3, 12.2.2, 12.2.3, 12.2.6, 12.2.8, 12.2.9, 16.5, 17.1.3, 17.2.9, 19.3, 19.4, 19.5.
ENTRY: 5.1, 5.2, 6, 6.1.3, 6.1.4.
ENTRY_RECORD: 6, 6.1.3.
EOF: 6.2.3, 16.10.
exit: 16.9, 16.10, 18.5.
EXTRACT: 6.2.5, 7.2, 7.2.1.
EXTRACT_LEFT: 7.2.1, 7.2.3, 7.2.4.
ExtractPointsPQ: 9.1.7, 9.2.5, 11.2.6.

fabs: 15.2.5.
FALSE: 4.1, 4.1.1, 5.4.2.1, 5.4.2.2, 5.4.3.1, 5.4.3.2, 5.4.3.3, 5.4.4.1, 5.4.5.1, 5.4.5.2, 5.4.5.3, 5.4.6.1, 7.2.3, 7.2.4, 9.2.3, 10.3, 11.2.5, 11.2.6, 12.2.8, 12.2.9, 16.5, 16.6.
FAST_IS_COVERED_EMPTY_LEAF: 5.4.5.2, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4.
FAST_IS_FULL_LEAF: 5.4.3.3, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4.
FAST_IS_INTERNAL_NODE: 5.4.2.2, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4.
fflush: 6.2.5, 12.2.4, 15.2.4, 17.2.3, 17.2.4, 17.2.5, 17.2.9.
fgets: 18.5.
fields: 5.3.0.4, 5.3.0.6, 11.2.4, 12.2.4, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.4, 15.2.6.
fileName: 6.1.5, 6.2.3, 16.3, 16.10, 18.3, 18.5.

Find: 10.4, 13, 13.1.3, 13.1.5, 13.2.2, 13.2.3, 13.2.4, 15.2.5, 16.5, 16.6, 17.2.9.
FIND: 13, 13.1.3, 13.1.4, 13.2.2, 15.2.5, 17.2.9.
FIND_NO_WINDOWING: 13, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 15.2.5, 17.2.9.
Find_NoWindowing: 13, 13.1.4, 13.1.5, 13.2.2, 13.2.3, 15.2.5, 17.2.9.
first: 11, 11.2.2, 11.2.4, 11.2.5, 11.2.6.
firstChildIndex: 11.2.4.
firstFull: 11.2.6.
fopen: 6.2.3, 16.10, 18.5.
fprintf: 4.4.2, 4.4.3, 6.2.5, 9.2.4, 9.2.5, 9.2.6, 9.2.9, 9.2.10, 9.2.11, 9.2.12, 11.2.4, 12.2.3, 12.2.4, 12.2.6, 12.2.9, 15.2.4, 15.2.5, 16.5, 16.9, 17.2.3, 17.2.4, 17.2.5, 17.2.9, 18.4, 18.5, 18.7, 19.4.
free: 6.2.4, 6.2.6, 9.2.7, 9.2.13, 12.2.5, 12.2.6, 17.2.5.
fscanf: 6.2.3, 16.10.

GenerateTraffic: 19.3, 19.4, 19.5.
GET_ANCHOR_INDEX: 5.4.2.9, 12.2.4, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.4, 15.2.6.
GET_AST_INDEX: 5.3.0.8, 12.2.4, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.6.
GET_BASE_INDEX: 5.4.3.5, 12.2.4, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4.
GET_LEFT_WINDOW_WIDTH: 5.4.2.4, 12.2.4, 13.1.3, 13.2.2, 13.2.4, 15.2.6.
GET_LOG_STEP: 5.3.0.6, 12.2.4, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.4, 15.2.6.
GET_N_LEFT_BASES: 5.4.4.3, 12.2.4, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4.
GET_N_RIGHT_BASES: 5.4.4.5, 12.2.4, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4.
GET_NEXT_HOP_INDEX: 5.4.5.5, 12.2.4, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4.
GET_RIGHT_WINDOW_WIDTH: 5.4.2.6, 12.2.4, 13.1.3, 13.2.2, 13.2.4, 15.2.6.
GetMaxKeyIntervalsPQ: 9.1.7, 9.2.12, 11.2.6.
getMemoryAccesses: 15.1.3, 15.2.5.
GetMinKeyPointsPQ: 9.1.7, 9.2.6, 11.2.6.
getpid: 16.5, 19.5.
GetTime: 12.2.3, 14, 14.1.2, 14.2.5, 15.2.5.
GREATER_EQUAL_POWER2: 8.1, 8.1.1.

high: 12.2.10.

i: 6.2.2, 6.2.4, 6.2.5, 6.2.6, 6.2.7, 9.2.4, 9.2.5, 9.2.9, 9.2.10, 9.2.11, 11.2.4, 11.2.6, 12.2.2, 12.2.4, 12.2.6, 12.2.7, 13.2.2, 13.2.3, 13.2.4, 15.2.4, 15.2.5, 15.2.6, 16.5, 16.6, 16.8, 16.9, 17.2.5, 17.2.9, 18.7, 19.4, 19.5.
includerIndex: 12.2.2, 12.2.9.
IncreaseKeyIntervalsPQ: 9.1.7, 9.2.10, 11.2.6.
index: 9.1.5, 9.1.6, 9.1.7, 9.2.8, 9.2.9, 9.2.10, 9.2.11.
inFile: 6.2.3, 16.10, 18.5.
InitializeIntervalsPQ: 9.1.7, 9.2.8, 11.2.6.
InitializePointsPQ: 9.1.7, 9.2.2, 11.2.6.
inlineSearch: 10.4, 16.5, 16.6, 16.8.
input: 16.3, 16.6, 16.7.
inputLine: 18.5.
InsertIntervalsPQ: 9.1.7, 9.2.9, 11.2.6.
InsertPointsPQ: 9.1.7, 9.2.4, 11.2.6.
interval: 9.1.6, 9.2.8, 9.2.9, 9.2.10, 9.2.11, 9.2.12, 9.2.13.
INTERVAL: 9.1.5.
INTERVALS_PQ: 9.1.6.
intervalsPQ: 9.1.7, 9.2.8, 9.2.9, 9.2.10, 9.2.11, 9.2.12, 9.2.13, 11.2.6.
IP_ADDRESS: 4.3, 4.3.1.
IP_ADDRESS_SIZE: 4.3, 5.4.0.4, 6.2.3, 7, 7.1, 7.1.1, 7.2, 7.2.1, 8, 8.1.1, 11.2.5, 11.2.6, 15.2.4.
IS_COVERED_EMPTY_LEAF: 5.4.5.1, 5.4.5.2, 12.2.4, 15.2.6.
IS_EMPTY_LEAF: 5.4.5.3, 15.2.6.
IS_FULL_LEAF: 5.4.3.1, 5.4.3.3, 12.2.4, 15.2.6.

IS_FULL_OR_SPECIAL_LEAF: 5.4.3.2, 5.4.3.3, 5.4.5.2, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.6.
IS_INTERNAL_NODE: 5.4.2.1, 5.4.2.2, 5.4.3.2, 5.4.3.3, 5.4.5.2, 12.2.4, 15.2.4, 15.2.6.
IS_PREFIX: 7.2.3, 17.2.9.
IS_SPECIAL_LEAF: 5.4.4.1, 12.2.4, 13.2.2, 13.2.3, 13.2.4, 15.2.6.
IS_UNCOVERED_EMPTY_LEAF: 5.4.6.1, 12.2.4, 13.2.2, 13.2.3, 13.2.4, 15.2.6.
IsEmptyPointsPQ: 9.1.7, 9.2.3, 11.2.6.
isIncluded: 12.2.2, 12.2.9.

j: 6.2.2, 6.2.5, 6.2.7, 9.2.5, 9.2.11, 12.2.2, 12.2.7, 15.2.4, 15.2.5, 16.5, 17.2.9, 19.5.

k: 12.2.4, 15.2.5.
key: 9.1.3, 9.1.5, 9.1.7, 9.2.4, 9.2.5, 9.2.6, 9.2.9, 9.2.10, 9.2.11, 9.2.12, 16.3, 16.6, 16.7.

lastFull: 11.2.6.
left: 8, 8.1.1, 8.2.2, 8.2.3, 8.3, 8.3.1, 11, 11.2.2, 11.2.4, 11.2.5, 11.2.6, 15.2.3, 15.2.6, 17.1.3, 17.2.3, 17.2.4, 17.2.5, 17.2.6, 17.2.7, 17.2.8.
left_window_width: 5.4.2.3, 5.4.2.4.
leftCovered: 11.2.6.
leftWindowWidth: 11, 11.2.2, 11.2.4, 11.2.5, 11.2.6, 15.2.6.
left0length: 11.2.6.
lenght: 6.2.3.
length: 6.1.3, 6.2.3, 6.2.5, 6.2.7, 7, 7.0.3, 7.1, 7.1.1, 7.2.3, 7.2.4, 18.3, 18.4, 18.5, 18.6, 18.7, 19.5.
logBranch: 8.1.1, 17.1.3, 17.2.6, 17.2.7, 17.2.8.
logResult: 8.1.
logRootBranch: 10.4, 11.2.5, 15.2.4, 16.6, 16.7, 16.8, 16.9.
logStep: 5.3.0.5, 8.1.1, 11.2.2, 11.2.4, 11.2.5, 17.1.3, 17.2.3, 17.2.4, 17.2.5, 17.2.6, 17.2.7, 17.2.8.
low: 12.2.10.

m: 11.2.6.
main: 6, 6.1.4, 6.2.6, 12, 16.5, 18.4, 19.4.
malloc: 4.4.2, 6.2.3, 12.2.3, 16.7.
MAX: 4.4.1.
MAX_ANCHORS: 5.3.0.3, 11.2.3, 11.2.4, 12.2.3.
MAX_AST_LEVELS: 5.4.0.4, 10.4, 15.2.4, 16.6, 16.7, 16.8, 16.9.
MAX_AST_NODES: 5.4.0.3, 11.2.3, 11.2.4, 12.2.3.
MAX_BASES: 5.2.0.3, 12.2.8, 12.2.9.
MAX_ENTRIES: 6, 16.2, 16.5, 19.2, 19.4.
MAX_EXPERIMENTS: 15.2.2, 15.2.5.
MAX_IP_ADDRESS: 4.3.2, 8.1.1, 11.2.3, 11.2.4, 11.2.5, 11.2.6, 15.2.4, 15.2.6, 17.2.9, 19.5.
MAX_LINE: 18.2, 18.5.
MAX_MAX_SPECIAL_LEAF: 10.3, 16.9.
MAX_N_TIMES: 15.2.5.
MAX_NEXT_HOPS: 5.1.0.5, 12.2.6.
MAX_PREFIXES: 18.2, 18.4.
MAX_TRAFFIC: 16.2, 16.5, 19.2, 19.4.
MAX_WINDOW_WIDTH: 5.4.2.7, 11.2.5, 11.2.6.
maxAddresses: 16.3, 16.10.
maxDepth: 15.2.3, 15.2.4, 15.2.6.
maxEntries: 6.1.5, 6.2.3.
maxIntervals: 9.1.6, 9.1.7, 9.2.8, 9.2.9.
maxPoints: 9.1.4, 9.1.7, 9.2.2, 9.2.4.
maxSize: 18.3, 18.5.
maxSpecialLeaf: 5.4.1, 10.4, 11, 11.2.4, 11.2.5, 11.2.6, 15.2.4, 16.6, 16.7, 16.8, 16.9.
medianPoint: 13.

mid: 12.2.10.
mimM: 11.2.5.
MIN: 4.4, 15.2.5.
MIN_C: 10.3, 16.9.
MIN_LOG_ROOT_BRANCH: 10.3, 16.9.
MIN_MAX_SPECIAL_LEAF: 10.3, 16.9.
MIN_N_REPEAT_SEARCH: 10.3, 16.9.
minM: 11.2.2, 11.2.5, 11.2.6.
minR: 11.2.2, 11.2.5, 11.2.6.
minTime: 15.2.5.
minWidth: 4.4.3.

n: 6.1.5, 6.2.5, 11.2.2, 11.2.4, 11.2.5, 11.2.6, 12.2.2, 12.2.10, 19.5.
n_left_bases: 5.4.4.2.
n_right_children: 5.4.4.4.
nAddresses: 15.1.3, 15.2.5, 16.10.
nAnchorAccesses: 13.1.5, 13.2.4, 15.2.5.
nAnchors: 11.1.3, 11.2.2, 11.2.3, 11.2.4, 12.2.3.
nAnchorTimes: 15.2.5.
nASTAccesses: 13.1.5, 13.2.4, 15.2.5.
nASTNodes: 11.2.3, 12.2.3.
nASTTimes: 15.2.5.
nBaseAccesses: 13.1.5, 13.2.4, 15.2.5.
nBases: 11.1.3, 11.2.3, 11.2.4, 12.2.2, 12.2.3, 12.2.8, 12.2.9, 13.2.2, 13.2.3, 13.2.4.
nBaseTimes: 15.2.5.
nBits: 18.5.
nBuckets: 11.2.6.
nCoveredEL: 15.2.3, 15.2.4, 15.2.6.
nDefaultNextHops: 15.2.5.
nEmpties: 11.2.6.
nEmptiesLeft: 11.2.6.
nEmptiesRight: 11.2.6.
nEntries: 6.1.5, 6.2.3, 6.2.4, 6.2.5, 6.2.6, 12.1.3, 12.2.2, 12.2.3, 12.2.6, 12.2.8, 12.2.9, 16.5, 17.1.3, 17.2.9, 19.3, 19.4, 19.5.
newKey: 11.2.6.
newText: 16.7.
nExaminedEntries: 12.2.2, 12.2.8, 12.2.9.
nExperiments: 15.1.3, 15.2.5.
next_AST_node: 13.
NEXT_HOP: 5.1, 5.1.0.6.
next_hop_index: 5.4.5.4.
nextFree: 11.2.2, 11.2.4.
nextHop: 5.5.0.3, 6.1.3, 6.2.3, 6.2.5, 12.2.2, 12.2.3, 12.2.4, 12.2.5, 12.2.6, 12.2.8, 12.2.9, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.4, 15.2.5, 17.2.9, 18.3, 18.4, 18.5, 18.7.
nextHopIndex: 5.2.0.4, 11.2.4, 11.2.6, 12.2.4, 12.2.8, 12.2.9, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4.
nextHopSize: 5.5.0.3, 12.2.3, 12.2.4, 15.2.4.
nFullL: 15.2.3, 15.2.4, 15.2.6.
nFulls: 11.2.6.
nIntervals: 9.1.6, 9.2.8, 9.2.9, 9.2.11, 9.2.12.
nLeaves: 15.2.4.
nLeavesPerLevel: 15.2.3, 15.2.4, 15.2.6.
nNextHopAccesses: 13.1.5, 13.2.4, 15.2.5.
nNextHops: 12.2.2, 12.2.3, 12.2.6, 12.2.8, 12.2.9.

nNextHopTimes: 15.2.5.
nNodesLeft: 11.2.2, 11.2.4, 11.2.5, 11.2.6.
nNodesRight: 11.2.2, 11.2.4, 11.2.5, 11.2.6.
node: 5.3.0.8, 5.4.2.1, 5.4.2.2, 5.4.2.4, 5.4.2.6, 5.4.2.9, 5.4.3.1, 5.4.3.2, 5.4.3.3, 5.4.3.5, 5.4.4.1, 5.4.4.3, 5.4.4.5, 5.4.5.1, 5.4.5.2, 5.4.5.3, 5.4.5.5, 5.4.6.1, 13.2.2, 13.2.3, 13.2.4.
nodeIndex: 13.2.2, 13.2.3, 13.2.4.
nPoints: 9.1.4, 9.2.2, 9.2.3, 9.2.4, 9.2.5, 9.2.6.
nPrefixes: 18.3, 18.4, 18.5, 18.7.
nRepeatSearch: 10.4, 15.2.5, 16.5, 16.6, 16.7, 16.8, 16.9.
nSpecialL: 15.2.3, 15.2.4, 15.2.6.
nTimes: 15.2.5.
nTraffic: 16.5, 19.3, 19.4, 19.5.
nTrafficInside: 19.5.
NULL: 4.4.2, 6.2.4, 9.2.4, 9.2.5, 9.2.6, 9.2.7, 9.2.9, 9.2.10, 9.2.11, 9.2.12, 9.2.13, 10.3, 16.4, 16.5, 16.6, 18.5, 19.4.
number: 8.1.
nUncoveredEL: 15.2.3, 15.2.4, 15.2.6.
nUniqueEntries: 6.2.4, 12.2.3.

oldAnchor: 11.2.5.
oldLeftWindowWidth: 11.2.5.
oldLogStep: 11.2.5.
oldNNodesLeft: 11.2.5.
oldNNodesRight: 11.2.5.
oldRightWindowWidth: 11.2.5.
oldStep: 11.2.5.
options: 16.4.
out: 4.4.3, 6.1.5, 6.2.5, 12.1.3, 12.2.3, 12.2.4, 15.1.3, 15.2.4, 15.2.5, 17.1.3, 17.2.3, 17.2.4, 17.2.5, 17.2.6, 17.2.7, 17.2.8, 17.2.9, 18.3, 18.7.

parameters: 5.4.1, 11, 11.1.3, 11.2.2, 11.2.3, 11.2.4, 11.2.5, 11.2.6, 12.1.3, 12.2.3, 13, 13.1.4, 13.2.3, 15.1.3, 15.2.4, 15.2.5, 16.3, 16.5, 16.6, 16.7, 16.8, 16.9, 17.1.3, 17.2.9.
PARAMETERS: 10.4.
ParametersHelp: 16.3, 16.4, 16.7.
Parser: 16.3, 16.4, 16.6.
percInside: 19.3, 19.4, 19.5.
perror: 6.2.3, 16.10, 18.5.
point: 5.2.0.4, 5.3.0.4, 9.1.4, 9.2.2, 9.2.4, 9.2.5, 9.2.6, 9.2.7, 11.2.4, 11.2.6, 12.2.4, 12.2.8, 12.2.9, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.6.
POINT: 9.1.3.
POINTS_PQ: 9.1.4.
pointsPQ: 9.1.7, 9.2.2, 9.2.3, 9.2.4, 9.2.5, 9.2.6, 9.2.7, 11.2.6.
position: 7.2.
posNextHop: 18.5.
PREFIX: 7.0.3.
prefix: 7.1, 7.1.1, 7.2.3.
PREFIX_LEFT_EXTREME: 7.1, 12.2.9, 17.2.9, 19.5.
PREFIX_MATCH: 7.2.4, 17.2.9.
PREFIX_RIGHT_EXTREME: 7.1.1, 12.2.9, 19.5.
PRINT_WORD: 4.4.3, 12.2.3, 15.2.4, 15.2.5, 16.5.
PrintBucketsAnchorGivenBranch: 17.1.3, 17.2.7.
PrintBucketsAnchorGivenStep: 17.1.3, 17.2.4, 17.2.7.
PrintBucketsLeftGivenBranch: 17.1.3, 17.2.6.

PrintBucketsLeftGivenStep: 17.1.3, 17.2.3, 17.2.6.
PrintEntries: 6.1.5, 6.2.5, 16.5.
printEntries: 10.4, 16.5, 16.6, 16.8.
printf: 6.2.5.
PrintRoutingPrefixes: 18.3, 18.4, 18.7.
printRoutingTable: 10.4, 16.5, 16.6, 16.8.
PrintRoutingTable: 12.1.3, 12.2.4, 16.5.
PrintRoutingTableStatistics: 15.1.3, 15.2.4, 16.5.
priority: 11.2.6.
ptr: 4.4.2.

qsort: 6.2.4, 6.2.7, 12.2.6, 12.2.7.

r: 11.2.6.
randNumber: 19.5.
ReadEntries: 6, 6.1.4, 6.1.5, 6.2.3, 16.5, 19.4.
ReadRoutingPrefixes: 18.3, 18.4, 18.5.
ReadTraffic: 16.3, 16.5, 16.10.
realloc: 4.4.2, 12.2.3, 12.2.8.
RecursivelyBuildBaseVector: 12.2.2, 12.2.8, 12.2.9.
REMOVE: 7.2.2.
result: 8.1.
right: 8, 8.1.1, 8.2, 8.2.1, 8.3, 8.3.1, 11, 11.2.2, 11.2.4, 11.2.5, 11.2.6, 15.2.3, 15.2.6, 17.1.3, 17.2.3, 17.2.4, 17.2.5, 17.2.6, 17.2.7, 17.2.8.
right_window_width: 5.4.2.5, 5.4.2.6.
rightCovered: 11.2.6.
rightWindowWidth: 11, 11.2.2, 11.2.4, 11.2.5, 11.2.6, 15.2.6.
right0length: 11.2.6.
root: 15.2.3, 15.2.6.
rootIndex: 11.2.2, 11.2.4.
routing_file: 10.4, 16.5, 16.6, 16.8.
ROUTING_TABLE: 5.5.0.4.
ROUTING_TABLE_STRUCTURE: 5.5.0.3.
RunSearchTest: 15.1.3, 15.2.2, 15.2.5, 16.5.

s: 13.1.5, 13.2.2, 13.2.3, 13.2.4, 15.2.5, 18.3, 18.6.
segmentLength: 19.5.
SET_ANCHOR_INDEX: 5.4.2.8, 11.2.4.
SET_AST_INDEX: 5.3.0.7, 11.2.4.
SET_BASE_INDEX: 5.4.3.4, 11.2.4.
SET_COVERED_EMPTY_LEAF: 5.4.5, 11.2.4.
SET_FULL_LEAF: 5.4.3, 11.2.4.
SET_INTERNAL_NODE: 5.4.2, 11.2.4.
SET_LEFT_WINDOW_WIDTH: 5.4.2.3, 11.2.4.
SET_LOG_STEP: 5.3.0.5, 11.2.4.
SET_N_LEFT_BASES: 5.4.4.2, 11.2.4.
SET_N_RIGHT_BASES: 5.4.4.4, 11.2.4.
SET_NEXT_HOP_INDEX: 5.4.5.4, 11.2.4.
SET_RIGHT_WINDOW_WIDTH: 5.4.2.5, 11.2.4.
SET_SPECIAL_LEAF: 5.4.4, 11.2.4.
SET_UNCOVERED_EMPTY_LEAF: 5.4.6, 11.2.4.
SetDefaultParameters: 16.3, 16.6, 16.8.
shift: 17.2.4.
SortAndRemoveDuplicatesEntries: 5.2, 6.1.5, 6.2.4, 12, 12.2.3, 19.4.

sprintf: 16.7.
sqrt: 15.2.5.
srand48: 16.5, 19.5.
startClock: 14.2.2, 14.2.3, 14.2.5.
state: 16.3, 16.6.
StatisticFind: 13, 13.1.5, 13.2.2, 13.2.4, 15.2.5.
stderr: 4.4.2, 9.2.4, 9.2.5, 9.2.6, 9.2.9, 9.2.10, 9.2.11, 9.2.12, 11.2.4, 12.2.6, 12.2.9, 16.5, 16.9, 18.4, 18.5, 19.4.
stdevTime: 15.2.5.
stdout: 16.5, 18.4, 19.4.
step: 8, 8.1.1, 8.2, 8.2.1, 8.2.2, 8.2.3, 8.3, 8.3.1, 11, 11.2.2, 11.2.4, 11.2.5, 11.2.6, 13, 15.2.6, 17.2.3, 17.2.4, 17.2.5, 17.2.6, 17.2.7, 17.2.8.
stopClock: 14.2.2, 14.2.4, 14.2.5.
string: 6.1.3, 6.2.3, 6.2.5, 6.2.7, 7, 7.0.3, 7.1, 7.1.1, 7.2, 7.2.1, 7.2.2, 7.2.3, 7.2.4, 16.5, 18.3, 18.4, 18.5, 18.7.
strlen: 16.7.
strncmp: 18.5.
strstr: 18.5.
strtod: 16.6, 19.4.
strtol: 16.6, 18.6, 19.4.
SUB_INTERVAL: 17.2.2.
subIntervals: 17.2.5.
subLeft: 8, 8.2, 8.2.1, 8.2.2, 8.2.3, 8.3, 8.3.1, 11.2.4, 11.2.6, 15.2.6, 17.2.2, 17.2.3, 17.2.4, 17.2.5.
subRight: 8, 8.2, 8.2.1, 8.2.2, 8.2.3, 8.3, 8.3.1, 11.2.4, 11.2.6, 15.2.6, 17.2.2, 17.2.3, 17.2.4, 17.2.5.
sumDepthTimesWeight: 15.2.3, 15.2.4, 15.2.6.
sumLeafDepths: 15.2.3, 15.2.4, 15.2.6.
sumNAnchorAccesses: 15.2.5.
sumNASTAccesses: 15.2.5.
sumNBaseAccesses: 15.2.5.
sumNNextHopAccesses: 15.2.5.

t: 12.1.3, 12.2.4, 12.2.5, 13.1.5, 13.2.2, 13.2.3, 13.2.4, 15.1.3, 15.2.4, 17.1.3, 17.2.9.
table: 12.2.3, 15.1.3, 15.2.5, 16.5.
tempAddress: 16.5, 19.5.
tempAnchorTable: 12.2.3.
tempAST: 12.2.3.
tempBase: 12.2.8.
tempInterval: 9.2.9, 9.2.10, 9.2.11.
tempNextHop: 12.2.6.
tempPoint: 9.2.4, 9.2.5.
testData: 15.1.3, 15.2.5, 16.5.
text: 16.3, 16.7.
time: 15.2.5.
timeSum: 15.2.5.
time2Sum: 15.2.5.
totalLength: 19.5.
totalMemory: 15.2.4.
totalWeight: 15.2.3, 15.2.4, 15.2.6.
traffic: 16.3, 16.5, 16.10, 19.3, 19.4, 19.5.
traffic_file: 10.4, 16.5, 16.6, 16.8.
Translate: 18.3, 18.5, 18.6.
TraverseAST: 15.2.3, 15.2.4, 15.2.6.

TRUE: 4.1, 4.1.1, 5.4.2.1, 5.4.2.2, 5.4.3.1, 5.4.3.2, 5.4.3.3, 5.4.4.1, 5.4.5.1, 5.4.5.2, 5.4.5.3, 5.4.6.1, 7.2.3, 7.2.4, 9.2.3, 11.2.5, 11.2.6, 12.2.3, 12.2.9, 13, 13.1.3, 13.1.4, 13.2.2, 13.2.3, 13.2.4, 15.2.5, 16.2, 16.5, 16.6, 17.2.9, 18.6.

uint32_t: 4.0.2.
useFixedBranch: 11.2.5.
useInline: 15.1.3, 15.2.5, 17.1.3, 17.2.9.

v: 12.2.2, 12.2.10.
verbose: 12.1.3, 12.2.3, 15.1.3, 15.2.5.
VERBOSE: 16.2, 16.5.
verifyCorrectness: 10.4, 16.5, 16.6, 16.8.
VerifyFindCorrectness: 16.5, 17.1.3, 17.2.9.

weight: 15.2.6.
*WHILE_EXPLORE_BUCKETS_FROM_*: 8.
WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_LEFT: 8.2.2, 15.2.6, 17.2.4, 17.2.5.
WHILE_EXPLORE_BUCKETS_FROM_ANCHOR_TO_RIGHT: 8.2, 15.2.6, 17.2.4, 17.2.5.
WHILE_EXPLORE_BUCKETS_FROM_LEFT_TO_RIGHT: 8.3, 11.2.4, 11.2.6, 17.2.3, 17.2.5.
windowWidth: 13.2.2, 13.2.4.
WORD: 4.2.

x: 12.2.2, 12.2.10.