Universidad Carlos III de Madrid

Algorithms and Data Structures (ADS)
Bachelor in Informatics Engineering
Computer Science Department

A friendly notebook on Data Structures and Algorithms.

Authors: Isabel Segura Bedmar
Harith Al-Jumaily
Julian Moreno Schneider
Juan Perea

June 2011

Preface

The purpose of this document is to provide study material that can be used for independent study by the students of the subject 'Data Structures and Algorithms'. We have tried to write it in a student-friendly way that encourages students to learn as well as enjoy.

The document reviews the main concepts of the subject providing clear examples to help students. Each chapter also proposes a set of exercises to reinforce students' knowledge. Most of the information has been sourced from the books [**?**, **?**].

Chapter 1 introduces the concepts of data structure and algorithm and provides some advices to achieve a good design sw.

Chapter 2 introduces analysis of algorithms.

Chapter 3 does not cope with any data structure in particular, but rather presents the concept of recursivity.

Chapter 4 reviews the main linear data structures: lists, stacks and queues.

Chapter 5 introduces the tree abstract data type, the tree traversal algorithms, binary trees, binary search trees and AVL trees.

Chapter 6 introduces the tree abstract data type, the tree traversal algorithms, binary trees, binary search trees and AVL trees.

Chapter 7 presents binary search trees.

Chapter 8 reviews AVL trees.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents concepts like data structures and algorithms.

How do you usually resolve a problem (for example, to check if a number is primer or not, or to cook a cake)?. Firstly, you must define and understand the problem. (=**Analysis**). Then, you must design a set of steps or operations to solve it. (=**Algorithms**). Finaly, you must try that these steps achieve to solve it (=**Tests**). If the solution is a sw program, you must translate these instructions (algorithms) from natural language to a programming language in order to the computer can run them (=**implementation**). Also, we must check that the solution works well (=**tests**) and must correct the errors. Finally, the end users must use the program.

Therefore, to resolve a problem, the first step is to define a set of instructions for solving it. An **algorithm** is a finite set of instructions for solving a problem.

Rather than an algorithm, we need to design the more suitable way of organizing the data of the problem. A **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently. When you design a program for a given problem, you must find the data structure more efficient to resolve it. The algorithm for calculating the multiple numbers of 2 is very easy, but *What data structure must you use to calculate and store them? and for storing the names of the students in your course?*. If you must develop a program that resolves these problems, you just use an array to store them. However, if you must develop a program that allows you to easily create and manage your family tree (or any taxonomy), you should use a tree. The choice of a data structure is often a fairly difficult one. It must always be taken based on the operations that are to be performed on the data.

Do you dare to solver your first problem. Please, write an algorithm to calculate the maximum of two numbers. You can use the java language or pseudocode to write it. After, you can compare your solution with the proposed one in Figure 1.1.

Now we are trying to solve a more difficult problem. Given an array of integers with n elements which range from *0* to n without any repition, find the number that does not occur in the array.

```java
public class Tester {
    public static void main(String args[]) {
        int[] aNums=new int[10]; /**Declare an array of int with size 100*/
        /**To test the method showMaxMin, we have randomly generated 100 numbers (int)*/
        for (int i=0; i<100; i++) {
            aNums[i]= ((int) Math.random() * 100);
            System.out.println(aNums[i]);
        }
        /**Now, we show the maximum and minimum elements in the array aNums*/
        showMaxMin(aNums);
    }
    /**
     * To show the maximum and minimum element in the array
     * @param aNums array of int
     */
    public static void showMaxMin(int[] aNums) {
        if (aNums==null) {
            System.out.println("Array is empty");
            return;
        }
        /**Set up max and min*/
        int min=Integer.MAX_VALUE; //=pot(2,31)-1
        int max=Integer.MIN_VALUE; //=-(pot(2,31)-1)

        /**Traverse the array, comparing each element with the previous maximum and minimum*/
        for (int i=0; i<aNums.length;i++) {
            if (aNums[i]<min) min=aNums[i];
            if (aNums[i]>max) max=aNums[i];
        }
        System.out.println("Minimum is " + min);
        System.out.println("Maximum is " + max);
    }
```

Figure 1.1: Java implementation of an algorithm to show the maximum and minimum elements in an array. To test the method, the main method randomly create an array of 100 integer numbers

It is clear that if elements in the array can range from 0 to n, every element can take n+1 different values, however the array can only also store n elements. For example, if n=4, for the array is 0,1,3,4, then 2 is the number.

It is very easy. Add the numbers from 0 to n. Add the elements in the array (a[0]+...+a[n-1]). The final solution is the difference between those numbers (see Figure 1.2).

Now, given a number N, define an algorithm to look for the the prime number from 0 to N. The basic idea is to create an array of N elements and set it with the numbers from 1 to N. The algorithm traverse the array seting to 0 those elements that are not prime numbers. When an element is divisible by some of the previous elements in the array (!=0) is not a primer number and must is set to 0 (see Figure 1.3). You can find a detail description of this algorithm Sieve of Eratosthenes.

Of course, you have already known some data structures (such as String, Integer, Long, Boolean, etc (they are primitive types) or arrays), so the goal is this course is to present more complex data structures like lists, stacks, queues, trees, and graphs, which are very useful to resolve different problems. You can use and combine them to resolve complex problems.

```java
public class Tester {
    public static void main(String args[]) {
        int aNums[]={3,1,2,0,5};
        for (int i=0; i<aNums.length;i++) System.out.print(aNums[i]+", ");
        System.out.println(" => " + findNotInArray(aNums));

    }

    /** Given an array of integers with n elements which range in [0,n]
     *  without any repetition, find the number that does not occur in the array.
     */
    public static int findNotInArray(int[] aNums) {
        if (aNums==null) {
            System.out.println("Array is empty");
            return -1;
        }
        int n=aNums.length;
        int sumN=n; /**to store the sum from 0 to n*/
        int acc=0; /**to store the sum of the elements in the array*/
        for (int i=0; i<n; i++) {
            acc+=aNums[i];
            sumN+=i;

        }
        return sumN-acc;
    }
```

Figure 1.2: Given an array of integers with n elements which range from 0 to n without any repition, find the number that does not occur in the array.

1.1 Exercises

You must research on the following concepts: Top-Down Design, Abstraction and Encapsulation. Sum the up and write examples that help to understand these concepts.

```java
public class Tester {
    public static void main(String args[]) {
        int N=10; firstPrimeNumbers(N);
        N=100; firstPrimeNumbers(N);
    }
    /** Shows the prime numbers in [1,N].*/
    public static void firstPrimeNumbers(int N) {
        if (N<=0) return;
        int aPrimes[]=new int[N];
        System.out.println("The " + N + " first prime numbers :" );
        /**sets the array from 1 to N*/
        for (int i=0; i<N; i++) aPrimes[i]=i+1;
        /**Traverses the array from 2 to N-1.
         * Every element in array (!=0) is divided by the previous elements in array (!=0).
         */
        int j=1;
        while (j<N) {
            if (aPrimes[j]!=0) {
                System.out.println(aPrimes[j]);
                int k=j+1;
                while (k<N) {
                    /**If there is a previous element that divides aPrimes[k] =>
                     * it is not a prime number.
                     */
                    if (aPrimes[k]%aPrimes[j]==0) aPrimes[k]=0;
                    k++;
                }
            }
            j=j+1;
        }
    }
```

Figure 1.3: Java implementation of the Sieve of Eratosthenes algorithm that shows the first prime numbers from 0 to N.

Chapter 2

Algorithm Analysis

Most of the information has been sourced from the books [**?**, **?**].

2.1 Introduction

In mathematics and computer science, an **algorithm** is an effective method for solving a problem, expressed as a pre-written set of well-defined, ordered and finite instructions and rules. We can use a more-or-less formal language, such as natural language, pseudocode, or a programming language like Java, to express these algorithms.

For example, if we want to calculate the summation of the first N integers, we start with a result value of 0, then loop from 1 to N, and for each value i, we add i to the result. In Java, we write this as:

```
static long func1(int n) {
        long sum=0;
        for (int i=1; i¡=n; ++i) sum += i;
        return sum;
}
```

We are used to deal with algorithms in our life: if we want *to look for a word in a dictionary*, we usually open the dictionary by the middle, then search again in the first or the second half depending on the alphabetical order, and so on, or we can look at every page in order until we find the page that contains our word, which is obviously less efficient.

In Mathematics, we can use (at least) two methods for *calculating the list of prime numbers (any number that can only be divided by 1 and itself) that are less than N*: we can iterate over odd numbers (and, for each candidate, test it against all prime numbers less tham or equal its square root), or we can use the more complex but more efficient Sieve of Eratosthenes algorithm (see

the Wikipedia article[1]). The following list of steps describes the Eratosthenes' algorithm:

1. Create an array of integers containing the values from 2 to n: (2, 3, 4, ..., n).
2. Set a variable x to 2 (which is the first firme number)
3. Traverse the array from x to N, setting all multiples of x to 0.
4. The following number after x and greater than 0, is the next prime. Then, set the variable x to this number.
5. Repeat the 3-4 steps until x^2 is greater than n.

At the end, the non-cero elements in the array are the prime numbers from 0 to n.

We will use these examples in this chapter to illustrate the concepts shown.

2.2 Algorithm analysis

Most algorithms are designed to deal with problems that have variable sized inputs: this input size can determine the amount of resources the algorithm consumes until it completes. Typically, an algorithm can consume two types of resources for its completion: *time*, and storage (*memory* or disk space). Algorithm analysis tries to determine the amount of resources needed to execute it, in the form of a function related to the input size of the algorithm.

Taking a look at the first example mentioned in the introduction, analyzing the summation algorithm would result in counting the number of times each simple instruction is executed:

```
static long func1(int n) {
        long sum=0; → 1
        for (int i=1; i≤n; i++) → i=1:1 ; i≤n:n+1 ; i++:n
                sum += i; → n
        return sum; → 1
}
```

Thus, given n, this algorithm will execute a total of

$$1 + 1 + (n + 1) + n + n + 1 = 3n + 4 \tag{2.1}$$

single instructions; if c is the time it takes to execute the longest instruction, then the running time of or algorithm will be, at most, c(3n+4). Let us suppose c=1 millisecond; then the running time will be (3n+4) milliseconds.

[1] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Similarly, we can easily see that searching time in a dictionary is always limited, whichever the chosen algorithm, by a multiple of the dictionary size. Being N the number of pages in the dictionary, if we iterate through all the pages, it is obvious that we will need to look at the whole set of N pages if the searched word is in the last one. But if we divide the dictionary in halfs, we will only need to open the dictionary a few times; i n fact, doubling the dictionary size will result in one page looked more. We can then infer that running time will be limited by a multiple of log2N. This is not relevant if a dictionary had a few pages, but becomes an important issue for huge dictionaries.

When we estimate running time, we have to be aware that we can be lucky and find the searched word in the page we open the dictionary by at the first try (best case), but we can also be unlucky and not find the word until the last page (worst case), so maybe we want to study an average case.

When calculating prime numbers, there are no best and worst case scenarios; running time will only depend on N. But, in this case, running time won't be limited by a multiple of N, as the bigger the number we are checking, the longer it will take to check if it's prime (checking if an odd number i is prime might need $\sqrt{i}/2$ iterations, so running time for each i will be limited by a multiple of $\sqrt{i}$). Therefore, the overall running time will be limited by a multiple of $\sqrt{1}+\sqrt{2}+\sqrt{3}+\sqrt{5}+\sqrt{7}+\sqrt{11}+\ldots\sqrt{N}$, which is limited by a multiple of NN.

2.3 Performance versus Memory

Though memory space (or disk space, if we are talking about, for example, a database) is an almost unlimited resource in modern computing, it can still be an issue for some algorithms. For example, an algorithm for a computer that plays chess might need to store all the possible moves it has studied.

Sometimes, the same problem can be solved by an algorithm that penalizes performance and an algorithm that penalizes memory space. For example, the Sieve of Eratosthenes algorithm achieves a better performance, but it needs to keep an array of booleans, which has a size that is multiple of the problem size, while the other algorithm will need no extra storage.

Another way to increment performance by using more memory is to change the way data are stored. For example, back to our dictionary, we redistribute the words and use a function to calculate the page number in which a word must be inserted or searched for, (this is called a hash function). This way, searching time will be constant, whichever the dictionary size, but memory space (the number of pages in the dictionary) will grow considerably.

As an example, we can use the following function: Let us assign a value to each letter (A=0, B=1, ..., Z=25), and let us use the first three letters of each word (L1, L2 and L3). Then, the page number will be:

$$PageNumber = 26^2 * L1 + 26 * L2 + L3 \tag{2.2}$$

So if we want to store (or look for) 'ABBEY', we will find it in:

$$PageNumber = 26^2 *' A' + 26 *' B' +' B' = 26^2 * 0 + 26 * 1 + 1 = 0 + 26 + 1 = 27 \tag{2.3}$$

This algorithm is much faster, but the storage penalty is really important: such dictionary would have 263 = 17576 pages, most of which will be empty, and perhaps other pages will not have enough space to hold all the words they should.

2.4 Asymptotic analysis

When analyzing algorithms, we use asymptotic notations to show in a mathematical way how resource consumption scales with input size. The aim of asymptotic analysis is to focus on the shape of the function curve, and to define functions as simple as possible that limit either as an upper bound (big O), as a lower bound (big Omega), and as both (big Theta). In this chapter, we will focus on upper bound limits, and just mention lower bound limits.

2.4.1 Upper bound limit big O

We have seen in the previous examples how algorithm efficiency for big input sizes can be measured in terms of a limiting simple function. Such function, that describes the growing behaviour of an algorithm for input sizes tending to infinity is known as asymptotic notation or 'big O' notation.

The mathematical definition of big o is a s follows:

- Consider a function f(N) which is non-negative for all integers N¿=0 . "f(N) is big o g(N)", or "f(N) O(g(N))", of "f(N) has order of g(N) complexity", if there exists an integer n0, and a constant c¿0, such that, for all integers n¿=n0, then f(n)¡=cg(n). In a more comprehensive way, this definition says that for values of n greater than a given value n0, g(N) is proportional to f(N), or worse. In other words, it says that g(N) grows at least as fast as f(N) for big values of N.

Chapter 3

Recursion

Most of the information has been sourced from the books [**?**].

Some algorithms and mathematical functions can be defined in a recursive way. This happens when the algorithm or function being defined is used in its own definition. For example, the factorial function of an integer number N can be defined in an iterative way as the product of all the positive integers that are less than or equal to N:

$$N! = 1 * 2 * 3 * \ldots * (N-1) * N \tag{3.1}$$

Since $(N-1)! = 1 * 2 * 3 * \ldots * (N-1)$, it turns out that:

$$N! = (N-1)!N \tag{3.2}$$

This definition, while correct, is still incomplete, as it doesn't say how to calculate the factorial of 0 (or 1). So we need to define a base case, at which the recursion will stop. The complete recursive definition of the factorial function is:

$$0! = 1 \tag{3.3}$$

$$N! = (N-1)!N, \quad for\ N > 0 \tag{3.4}$$

Another simple recursive algorithm is the Euclidean algorithm[1] to calculate the greatest common divisor of two numbers a, b (being a¿=b), which can be recursively defined as:

$$gcd(a,b) = a \qquad if\ b = 0 \tag{3.5}$$

$$gcd(a,b) = gcd(b, amodb), \quad otherwise \tag{3.6}$$

In this example, it's not so obvious that recursion reduces the problem towards the base case, but we still can see that the numbers become smaller in each iteration.

[1] http://en.wikipedia.org/wiki/Euclidean_algorithm

Finally, another typical example of a recursive function is the definition of the Fibonacci numbers[2], formed by a series of numbers, starting by 0, 1, in which each number is obtained by summing the previous two numbers in the series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... In this case, the recursive definition is much clearer than an iterative one:

$$Fib(0) = 0 \tag{3.7}$$

$$Fib(1) = 1 \tag{3.8}$$

$$Fib(N) = Fib(N-2) + Fib(N-1), \quad for\ N > 1 \tag{3.9}$$

As a conclusion, the definition of a function of algorithm is recursive when it is formed by:

1. A set of one or more simple base cases, to stop recursion.
2. A set of one or more rules that reduce complex cases towards the base case(s).

It is important -and not always easy- to guarantee that the recursive algorithm will not end up in an infinite loop.

3.1 Loop equivalence and examples

Almost all recursive algorithms can be solved in an iterative way, using 'for' or 'while' loops. In some cases, the recursive solution will be more elegant and easier to understand and implement. In other cases, the iterative solution should be chosen (mostly due to the limitations exposed in the next section). Some code for the recursive algorithms mentioned in the introduction follows, along with its equivalent iterative algorithm:

3.1.1 Factorial

Both implementations are equally simple and elegant.

```
static long factorialRec(int n) {
            if (n ¡ 2) {
              return 1;
            }else {
              return n * factorialRec(n  1);
            }
}

static long factorialIt(int n) {
            long fact = 1;
            (for int i=2; i¡=n; i++) fact*=i;
            return fact;
}
```

[2] http://en.wikipedia.org/wiki/Fibonacci_number

3.1.2 Euclidean algorithm

In this case, the recursive implementation looks more elegant than the iterative solution, as the latter even needs an auxiliary variable to avoid problems, making the code more obfuscated:

```
static long euclideanIt(long a,long b) {
        while (b¿0){
                long aux=a;
                a=b;
                b=aux%b;
        }
}

static long euclideanRec(long a,long b) {
        if (b==0) {
          return a;
        }else {
          return euclideanRec(b,a%b);
        }
}
```

3.2 Limitations

However, being in most cases more elegant than the equivalent iterative solution, recursive algorithms must be used with care, as they have some limitations.

3.2.1 Stack usage and overflow

Computers use an execution stack (also known as call stack[3]) to store some necessary information related to all the running functions (from main to the currently running function). For each running function, this information includes the returning address (the address into the caller function to which execution should return when the function finishes) as well as local variables and parameters. Execution stack size is normally a quite limited resource (sometimes a few kB). This is normally enough for most applications, but an uncontrolled (or not too well estimated) recursive algorithm can easily cause a crash in the form of a stack overflow exception. If we take a look back at the examples related to the Euclidean algorithm, the iterative implementation will make a constant use of the execution stack. No matter the number of iterations, the stack will only hold one copy of the parameters 'a' and 'b' of the local variable 'aux' (as well, of course, as the returning address). But if we take the recursive algorithm, for each recursive call, the system will create a copy in the stack of the parameters parameters 'a' and 'b' and of the returning address. This is not a problem in the case of the Euclidean algorithm, is it's normally resolved in a few iterations,

[3] http://en.wikipedia.org/wiki/Call_stack

but can really be an issue in other recursive algorithms that need more recursive calls and/or more memory space for each iteration. For example, calculating 'fibonacciRec(5000)' will cause a stack overflow with a stack size of 32kB (4 bytes for 'n' 4 bytes for the returning address, in a 32 bit architecture, multiplied by 4999 calls, this makes 39992 bytes) issues. One typical example is the calculation of the first N Fibonacci numbers. The iterative implementation has a linear complexity (it's O(n)), while the recursive implementation has an exponential complexity (it's O(2n)). This happens because fibonacciRec(i) will be called from fibonacciRec(i+1) and from fibonacciRec(i+2). What's more, fibonacciRec(i+1) will be called from fibonacciRec(i+2) and from fibonacciRec(i+3), and so on, which means fibonnaciRec(i) will be called a total number of 2n-i times, causing the recursive implementation to be completely inefficient.

3.3 Cases of use

However, being aware of the mentioned limitations, there are cases in which the recursive solution should be considered. A complete discussion can be found here1. In this chapter, we will only mention some typical paradigms. There are no systematic approaches to neither of these paradigms, it takes some time and practice to understand and master them, however it's important to know they can be considered for some kinds of problems.

3.3.1 Divide and conquer strategy

A divide and conquer recursive algorithm[4] will break the problem down in several subproblems of the same type but with a reduced size, until the problem is so simple that it can be directly solved. Some examples of efficient divide and conquer algorithms are: Array sorting algorithms such as quicksort[5] and mergesort[6]. These algorithms split the array in several parts, and then call themselves recursively to sort each of these parts. The base case for both algorithms is an one-sized array. Gaming algorithms like the towers of Hanoi[7]. In this well-known game, we assume that, if we know how to move a tower formed by N disks from stack A to stack B using stack C as an auxiliary stack, then moving a tower formed by N+1 disks is as simple as moving the top N disks from stack A to the auxiliary stack C, then move disk N+1 from A to B, and then move again the top N disks from the auxiliary stack C over the disk already moved to stack B. The base case in this algorithm is when N=1, and we only need to move the disk from stack A to stack B.

[4] http://en.wikipedia.org/wiki/Fibonacci_number
[5] http://en.wikipedia.org/wiki/Quicksort
[6] http://en.wikipedia.org/wiki/Mergesort
[7] http://en.wikipedia.org/wiki/Hanoi_towers

3.3.2 Backtracking

Backtracking algorithms[8] are useful for solving certain computational problems when, at a given point of the problem, there are several ways to follow up, some of which may lead to a solution, and some of which may not. They are specially useful when the problem has several solutions, and we are looking for all them.

[8] http://en.wikipedia.org/wiki/Backtracking

Chapter 4

Lists, Stacks and Queues.

Most of the information has been sourced from the books [**?**, **?**].

4.1 Lists

A *list* is a collection of n elements stored in a linear order. The most common way for storing lists is using an *array* data structure. Each element in a list can be refered by an index in the range $[0, n-1]$ indicating the number of elements that preced e in the list. This representation provides that all operations performed on a given element take O(1) time. An array stores all elements of the list contiguously in memory and requires to initially know the maximum size of the list. This may produce an unnecessary waste of memory and other cases insufficient memory. A possible solution is to use a dynamic array (*java.util.ArrayList*) that is able to be reallocated when the space reserved for the dynamic array is exceeded. Unfortunately, this reallocating of the elements of a dynamic array is a very expensive operation. [1]

A linked list is a data structure that consists of a sequence of nodes such that each node contains a reference to the next node in the list. This representation does not require that elements of the list are stored contiguously in memory. Also, it just uses the space actually needed to store the elements of the list. On the other hand, linked lists do not mantain index numbers for the nodes and allow only sequential access to elements, while arrays allow constant-time random access. As it will be shown in following sections, the linked list data structure allows us us to implement some important abstract data structures such as stacks, queues, deques.

An implementation of a linked list may include the following methods:

- *isEmpty()*: test whether or not a list is empty.
- *size()*: return the number of elements of the list.

[1] Most of the information has been sourced from the books [**?**, **?**].

- *first()*: return the first element of the list. An error occurs if the list is empty.
- *last()*: return the last element of the list. An error occurs if the list is empty.
- *next(v)*: return the next node of v in the list.
- *prev(v)*: return the previous node of v in the list.
- *remove(v)*: removes the node v the list.
- *addFirst(v)*: add the node v at the beginning of the list.
- *addLast(v)*: add the node v at the end of the list.
- *addBefore(v,new)*: add the node new just before the v node.
- *addAfter(v,new)*: add the node new just after the v node.

Exercise: Look for more information on the main differences between arrays (static and dynamic) and linked lists. Write an outline that brings out the main advantages and disadvantages for each data structure.

4.2 Singly Linked Lists

A **linked list** is a sequence of **nodes**. Each node is an object that stores a reference to an element and a reference to the following node in the list. This link to the next node is called *next*. The order of the list is represented by the next links.

The first node of a linked list is the **head** of the list. The last node of a linked list is the **tail** of the list. The next reference of the tail node points to null. A linked list defined in this way is known as a **singly linked list**.

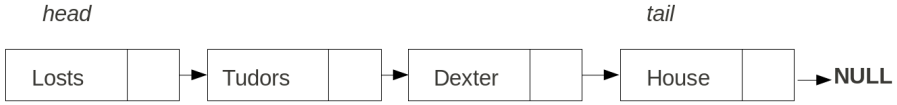


Figure 4.1: Example of a singly linked list containing my favorite series orderby preference. Each node contains a reference to the name of a TV serie and a reference to the next node (the following TV serie). The next reference of the last node (tail) links to *null*.

4.2.1 How can you implement a singly linked list

Firstly, we implement a *Node* class as shown in Figure 4.2. This implementation uses the generic parameterized type ¡E¿, which allows to store elements of any specified class (that is, you will use the Node class to create objects containing *String, Integer, Long*, etc and any other class that you specified).

```
/**
 * Example of Node class for a singly linked list of Objects.
 */
public class Node<E> {
    private E element;
    private Node<E> next;
    /**Constructor*/
    public Node(E e, Node<E> n) {
        element=e;
        next=n;
    }
    /**This method returns the element of this node*/
    public E getElement() {     return element;     }
    /**
     * This method returns the next node to this node
     * @return Node
     */
    public Node<E> getNext() {      return next;    }
    /**
     * Methods to modifier the properties of the Node class
     */
    public void setElement(E e) {       element=e;  }
    public void setNext(Node<E> n) {        next=n; }
}
```

Figure 4.2: Implementation of a Node of a singly linked list.

Figure 4.3 shows the partial implementation for a singly linked lis that only uses the reference to the head of the list (*head*), an instance variable to store the number of elements of the list (*size*) and a constructor method that sets the head node to null. For example, you may modify this class adding a new constructor method that has a node as input parameter and links the head of the list to this parameter.

4.2.2 How can you insert a new element in a singly linked list

The easiest case is when the new element is inserted at the head of the list. For example, I would like to add the TV serie 'Heidi' at the head of the above list. This TV series is my all time favorite serie, so it must be the first of the list

```java
/**A singly linked list*/
public class SinglyLinkedList<E> {
    /**Head node of the list*/
    protected Node<E> head;
    protected int size;
    /**Constructor that sets the head node to a node class*/
    public SinglyLinkedList() {
        head=null;
        size=0;
    }
    public int getSize() {return size;};
    public boolean isEmpty() {return (head==null);};

    /**Search and update methods
     * ..... */
```

Figure 4.3: We define the head of the list as a Node. The constructor sets this node to null.

(see 4.4). The following steps describe the process of insertion at the head of the list:

1. Create a new node. The name of the new serie and a reference to the same object as head (that is, next links to node that contains the serie *Losts*) must be passed to the constructor method.

2. Once you have created the node, you must set *head* (property of SinglyLinkedList) to point to new node.

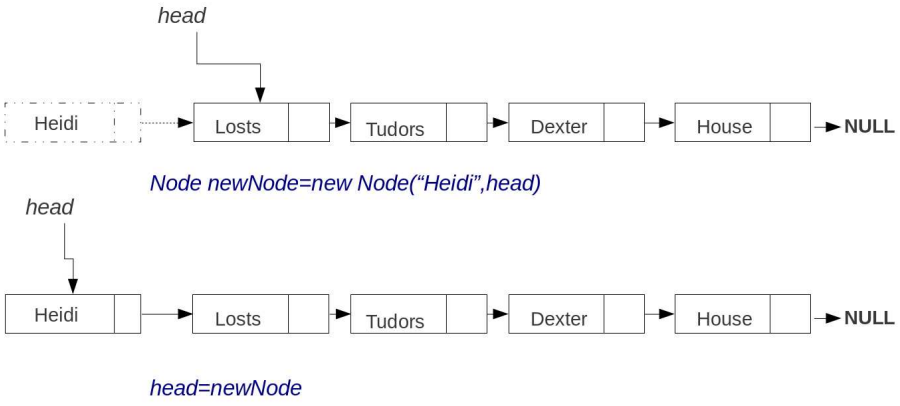


Figure 4.4: Insertion of an element at the head of a singly linked list.

Now, it is your turn. Please, write a new method called *insertHead* in the *SinglyLinkedList* class that inserts a new element at the head of the list. Figure **??** shows the implementation of this method.

```
/**Inserts the node v at the beginning of the list*/
public void addFirst(Node<E> newHead) {
    /**This must points to the old head*/
    newHead.setNext(head);
    /**Now, we must make that the head points to the newHead*/
    head=newHead;
    /**Increase the size of the list*/
    size++;
}
```

Figure 4.5: This method implements the insertion operation of an element at the beginning of a singly linked list.

To insert an element at the end of the list is very easy if you add a reference to the tail node, that is, an instance variable (with type Node¡E¿) to store the reference the last node in the list. For example, imagine that I like 'The Simpson', but I like it than less 'House', so I should insert it at the end of the list. I must follow the following steps:

1. Create a new node with the element 'The Simpson' and its next reference sets to null because this node will be the last node.

2. Then, the tail reference itself to this new node.

Figure 4.6 shows the above example. Please, try yourself defining the Node tail in the SinglyLinkedList class and adding the method *addLast*. You can find the solution in this new implementation in Figure ẽffig:addLastSList.

Now, take few minutes and think about how you can insert an element at the end of the list when you do not keep the tail reference in your implementation of the singly linked list. You should examine the list until you find node with a next reference to null. Do you dare to do it?. You can find a possible implementation in Figure 4.8.

4.2.3 How can you remove an element in a singly linked list

Now, let me show you how to remove at the head of the list. It is very easy!!!. You only need to set the head reference to its next node. This operation is illustrated in Figure 4.9 and its implementation is shown in Figure 4.10.

In order to remove the last node or an node at a give position of the list, we must access its previous node. Thus, the only way to find this previous node is to traverse the list from its beginning until to find it. Traversing the list may

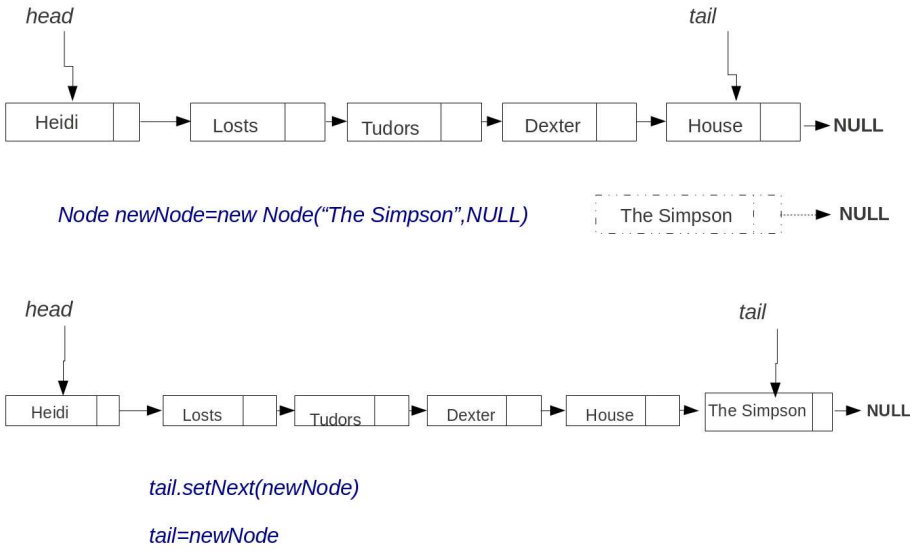


Figure 4.6: Insertion of an element at the end of a singly linked list. The class must have a property to store the tail of the list.

involve a big number of operations, taking a long time when the size of the list is big. The following section presents an effective solution for this problem.

```java
/**A singly linked list*/
public class SinglyLinkedList<E> {
    /**Head and tail nodes of the list*/
    protected Node<E> head, tail;

    protected int size;

    public SinglyLinkedList() {
        head=tail=null;
        size=0;
    }

    /**Inserts the node newTail at the end of the list*/
    public void addLast(Node<E> newTail) {
        /**This must points to null by its reference next*/
        newTail.setNext(null);
        /**In order to link the list with the new node, the tail
         * points to the newHead by its reference next*/
        tail.setNext(newTail);
        /**Now, we must set the tail node to newTail*/
        tail=newTail;
        /**If the list was empty, the tail must points to the newHead*/
        if (head==null) head=newTail;
        /**Increase the size of the list*/
        size++;
    }

    /**Inserts the node newHead at the beginning of the list*/
    public void addFirst(Node<E> newHead) {
        /**This must points to the old head*/
        newHead.setNext(head);
        /**Now, we must make that the head points to the newHead*/
        head=newHead;
        /**If the list was empty, the tail must points to the newHead*/
        if (tail==null) tail=newHead;
        /**Increase the size of the list*/
        size++;
    }
```

Figure 4.7: Implementation of the insertion operation of an element at the end of a singly linked list.

```java
/**
 * This Method inserts a element at the end of the list.
 * Since we do not keep a reference to the last node of the list
 * (that is, an instance variable tail), we traverse the list until
 * finding a node whose next reference is null.
 */
public void insertTail(Node<E> newTail) {
    newTail.setNext(null);
    if (head==null) {
        /**The list is empty, we can insert the node
        at the beginning of the list*/
        addFirst(newTail);
    } else {
        Node<E> aux=head;
        /** The list is traversed from the beginning to the last node
         * using an auxiliary node that initially is set to head.*/
        while (aux.getNext()!=null) aux=aux.getNext();
        aux.setNext(newTail);

    }
    size++;
}
```

Figure 4.8: Implementation of the insertion operation of an element at the end of a singly linked list without keeping the tail reference.

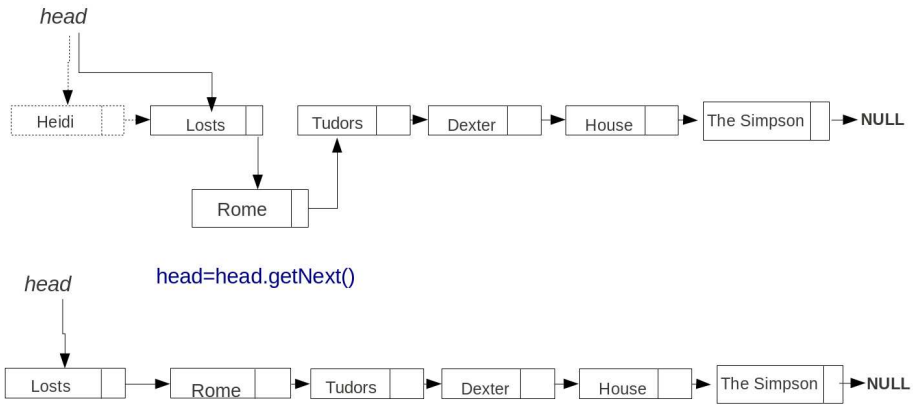


Figure 4.9: Removal of the head of the list.

```java
/**Removes the head of the list.
 * If the list is empty an exception is throwing*/
public void removeFirst() throws EmptyListException {
    if (isEmpty()) throw new EmptyListException("List is empty");
    head=head.getNext();
    if (head==null) tail=null;
    size--;
}
```

Figure 4.10: This method removes the first element of the list.

4.3 Doubly Linked Lists

The main drawback of singly linked lists is that insering or removing a node at the middle or the end of a list, it is necessary to visit all nodes from its head until the node just before the place where you want to insert it or the node that you want to remove. This operation is time consuming because we do not have a quick access to the node before the one that you want to remove or the position where you want to insert.

Let me ask you the following question: *How can you implement a linked list to improve the access to nodes?*. Figure 4.11 gives you the key. This representation allows to traverse the list in both directions.

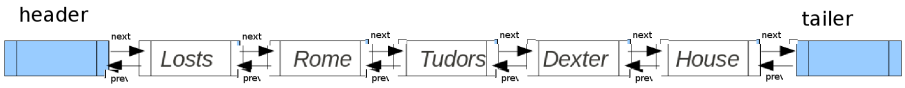


Figure 4.11: A doubly linked list storing my all time favorite TV series.

How can you define a node for a doubly linked list?. We need a mechanism that allow us to traverse the list from the beginning to the end and from the end to the beginning. The Node class (see Figure4.2) used in the implementation of a singly linked list only allowed us to go from left to right by the its instance variable *next*, which references to the following node in the list. Therefore, it would be very useful to define other instance variable to reference it previous node in the list. This variable is called *prev*. This implementation (see Figure 4.12) of a node for a doubly linked list makes easier to insert or remove an element at the end as well as in the middle of the list.

Now, we define the implementation of a doubly linked list. In order to facilitate the programming tasks, we can use two special nodes (called sentinels): *header* and *tailer*. The sentinel nodes do not store any reference to elements.

The *header* stores the node just before the head of the list (the header nodes points to the head of the list by its next reference; the prev reference of the head of the list must point to the header node). The *tailer* is the node after the tail of the list (the last element of the list must point to tailer by its next reference, while the tailer must poing to this last node by its prev reference) When the list is empty, the header and tailer nodes must point to each other.

Figure 4.15 describes the partial implementation of a doubly linked list. You can see how header and tailer sentinels are defined as objects of the above *DoublyNode* class. Also, a size property has been defined to store the number of elements at the list. We have defined a constructor that creates an empty list, that is, header and tailer sentinels are instantiated as *DoblyNode* objects (although, we do not give any value to their element property) and they point to each other.

Figures 4.14 and 4.13 show examples of removing and adding an element, respectively. As you can see in Figure **??** to insert a new element at the begin-

```java
package lists;

/**A class for nodes in a doubly linked list*/
public class DoublyNode<E> {
    protected E element;
    /**references to the next and previous nodes*/
    protected DoublyNode<E> next;
    protected DoublyNode<E> prev;
    /**Constructor*/
    public DoublyNode(E e, DoublyNode<E> p, DoublyNode<E> n) {
        element=e;
        prev=p;
        next=n;
    }
    /**This method returns the element of this node*/
    public E getElement() {return element;}
    /** This method returns a reference to its next node*/
    public DoublyNode<E> getNext() {return next;}
    /** This method returns a reference to its previous node*/
    public DoublyNode<E> getPrev() {return prev;}
    /**Methods set*/
    public void setElement(E e) {element=e;}
    public void setNext(DoublyNode<E> n) {next=n;}
    public void setPrev(DoublyNode<E> p) {prev=p;}
}
```

Figure 4.12: Implementation of a node for a doubly linked list.

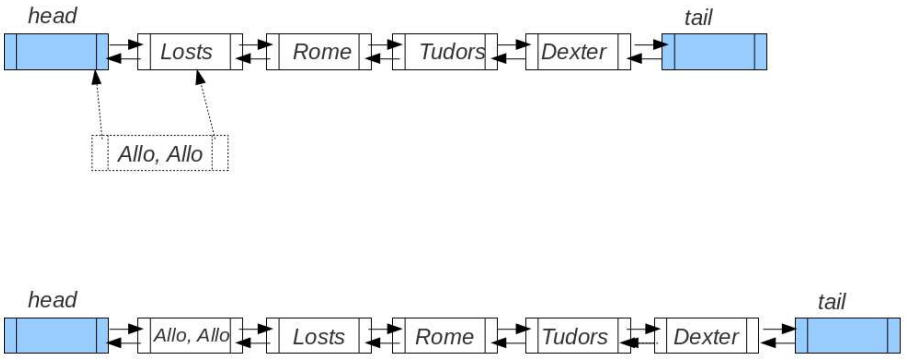


Figure 4.13: Inserting an element at the start of the list.

ning of the list or removing the last element of the list is very easy. Likewise, it is very easy to implement the methods *addLast()* and *removeFirst()*. *Do you*

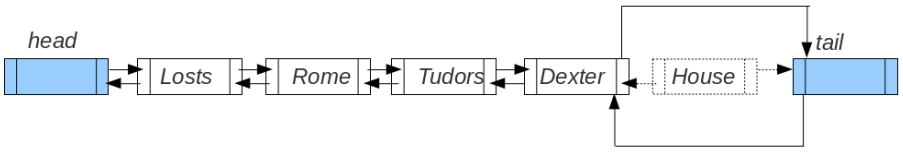


Figure 4.14: Removing an element from the end of the list.

```
public class DoublyLinkedList<E> {
    /**A sentinel node that is just before the head of the list. */
    protected DoublyNode<E> header;

    /**A sentinel node that is just after the last element of the list*/
    protected DoublyNode<E> tailer;

    protected int size;

    /**This constructor creates an empty list*/
    public DoublyLinkedList() {
        header=new DoublyNode<E>(null, null, null);
        tailer=new DoublyNode<E>(null, null, null);
        /**Sentinels must point to each other*/
        header.setNext(tailer);
        tailer.setPrev(header);
        size=0;
    }
    public int getSize() {      return size;    }
    public boolean isEmpty() { return (size==0);    }

    public void removeLast() throws EmptyListException {
        if (isEmpty()) throw new EmptyListException("List is empty");
        DoublyNode<E> lastNode=tailer.getPrev();
        DoublyNode<E> penultNode=lastNode.getPrev();
        tailer.setPrev(penultNode);
        penultNode.setNext(tailer);
        size--;
    }
    public void addFirst(DoublyNode<E> newNode){
        DoublyNode<E> oldNewNode=header.getNext();
        newNode.setNext(oldNewNode);
        newNode.setPrev(header);
        oldNewNode.setPrev(newNode);
        header.setNext(newNode);
        size++;
    }
```

Figure 4.15: Partial Implementation of a doubly linked list including the definition of the sentinel nodes *header* and *tailer*, the constructor method and the *addFirst()* and *removeLast()* methods.

dare to implement them?. You can find the implementation of these operation in Figure 4.16

```
public void addLast(DoublyNode<E> newNode) {
    DoublyNode<E> oldLastNode=tailer.getPrev();
    oldLastNode.setNext(newNode);
    newNode.setPrev(oldLastNode);
    newNode.setNext(tailer);
    tailer.setPrev(newNode);
    size++;
}


public void removeFirst() throws EmptyListException {
    if (isEmpty()) throw new EmptyListException("List is empty");
    DoublyNode<E> firstNode=header.getNext();
    DoublyNode<E> newlstNode=firstNode.getPrev();
    newlstNode.setPrev(header);
    header.setNext(newlstNode);
    size--;
}
```

Figure 4.16: Methods *addLast()* and *removeFirst()* of a doubly linked list.

4.3.1 How can you insert an element in the middle of a doubly linked list?

Doubly linked lists allow an efficient manner to access and modify their elements since they provide an easier way to insert and remove in the middle of the list than single linked lists. Figure 4.17 shows an example of a insertion in the middle of a doubly linked list. I have just watched the TV serie 'Bones'. I like more than 'Losts', but less than 'Allo, Allo", so I should add it just after the 'Allo, Allo' node. Firstly, I must define a new node with the element 'Bones', its next node must point to the node 'Losts' and its prev node to the node 'Allo, Allo'. Then, the nodes 'Allo, Allo' and 'Losts' must point to the node 'Bones' by their next and prev nodes respectively (see Figure 4.17). Now, imagine that I no longer like the 'Losts' serie. To remove it, their before node (that is, 'Allo, Allo') and after node ('Rome') must point to each other by their next and prev nodes respectively. Figure 4.18 includes the java code of the methods for inserting and removing elements in the middle of a doubly linked list. A full implementation of a doubly linked list can be found in the following link.

All methods in the implementation of a list using a doubly linked list take O(1). For a list of n elements, the space used is O(n).

4.4 Stacks

A **stack** is a collection of objects that are added and removed according to the the **Last-In First-out (LIFO)** principle. To understand better this principle, think about a stack of plates (Figure 4.19), *how do you add and take off plates from the stack?*. Normally, you add plates to the top of the stack and you take

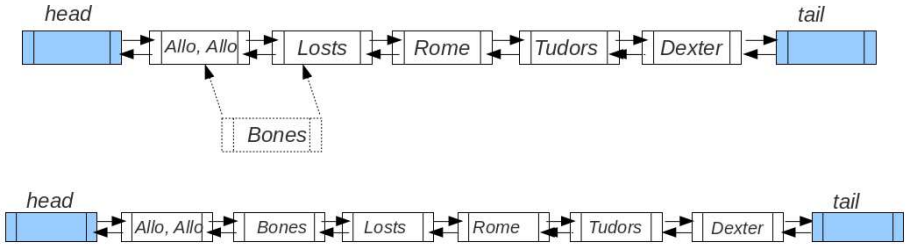


Figure 4.17: To add a new node after the .

```
/**Insert the node newNode before the v node*/
public void addBefore(DoublyNode<E> v, DoublyNode<E> newNode) throws IllegalStateException {
    DoublyNode<E> aux=getPrev(v);
    newNode.setPrev(aux);
    aux.setNext(newNode);
    newNode.setNext(v);
    v.setPrev(newNode);
    size++;
}

/**Insert the node vNode after the v node*/
public void addAfter(DoublyNode<E> v, DoublyNode<E> newNode) throws IllegalStateException {
    DoublyNode<E> aux=getNext(v);
    newNode.setNext(aux);
    aux.setPrev(newNode);
    newNode.setPrev(v);
    v.setNext(newNode);
    size++;
}

public void remove(DoublyNode<E> v) throws IllegalStateException {
    DoublyNode<E> sig=getNext(v);
    DoublyNode<E> ant=getPrev(v);
    ant.setNext(sig);
    sig.setPrev(ant);
    size--;
}
```

Figure 4.18: Methods addBefore, addAfter, remove for a doubly linked list.

off them from the top of the stack. This is just the *LIFO* principle.

Formally, a *stack* is an abstract data structure that is characterized by the following operations:

- *push(e)*: add the element e to the top of the stack.
- *pop()*: remove the top element from the stack and return it.

Other additionally operations are:

- *size()*: return the size of the stack.
- *isEmtpy()*: return true if the stack is empty; false eoc.
- *top()*: return the top element in the stack, without removing it.

Figure 4.19: Stack of plates.

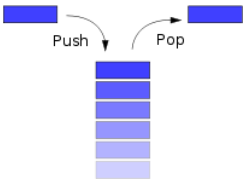


Figure 4.20: Push and pop operations.

Operation	**Stack**	**Output**
push('h')	(h)	-
push('e')	(h,e)	-
top()	(h,e)	e
push('l')	(h,e,l)	-
push('l')	(h,e,l,l)	-
push('o')	(h,e,l,l,o)	-
top()	(h,e,l,l,o)	o
push('!')	(h,e,l,l,o,!)	-
top()	(h,e,l,l,o,!)	!
size()	(h,e,l,l,o,!)	6
isEmpty()	(h,e,l,l,o,!)	false
pop()	(h,e,l,l,o)	!

Table 4.1: This table shows a secuence of operations on a stack of characters

This data structure is very useful for many applications which require to store the secuence of operations in order to reverse or undo them. For example, web browsers store the urls recently visited on a stack in order to allow users to visit the previously urls by pushing the back button. Likewise, stacks can be used to provide an *undo* mechanism to the text editors.

The *java.util* package already includes an implementation of the stack data structure due to its importance. It is recommended to use the *java.util.Stack* class, however in this section we design and implement ourselves a stack. First of all, we define an **interface** to declare the methods of the data structure (see Figure 4.21). You can note that this interface has been defined using the generic parameterized type E, which allows to store elements of any specified class. Also, we have defined the *EmptyStackException* class that will throw an exception when the methods *pop()* and *top()* are called on an empty stack.

```
package lists;
/** Interface for a stack: collection of elements that are inserted
 * and removed based on the LIFO principle.
 */
public interface Stack<E> {
    /**Returns the size of the stack*/
    public int size();

    /**Returns true if the stack is empty, otherwise false*/
    public boolean isEmpty();

    /**Returns the top element of the stack and remove it.
     * Throws an exception if the stack is empty. */
    public E pop() throws EmptyStackException;

    /**Returns the top element of the stack without remove it.
    * Throws an exception if the stack is empty. */
    public E top() throws EmptyStackException;

    /**Add the element e at the top of the stack*/
    public void push(E e);

}
```

Figure 4.21: Interface Stack. This interface uses the generic parameterized type E to contain elements of any specified class in the stack.

There are several ways to implement the Stack class. A simple way to represent a stack is to store its elements into an array. Thus, the stack consists of an array and an integer variable to indicate the index of the top element. Figure 4.23 shows a java class implementing an array-based stack. This class implements the interface Stack<E>. This implementation is based on the use of an array (for storing the elements of a stack. The instance variable *top* stores the index in which the top of the stack is stored in the array. Also, the maximum size of the array is defined in a constant *MAXCAPACITY*. Another instance variable (*capacity*) stores the actual capacity of the stack. The main drawback of this implementation is that it is necessary to initially know the maximum size of the stack. Thus, we have defined the *FullStackException* class that will throw an exception when the *push* method is called on a full stack (see Figure 4.22)

Now, you must write the code to build a stack containing the operations shown in Table 4.1. Table 4.2 summarizes the computational complexity for

```java
package lists;
/**
 * This Runtime exception will be thrown when the methods
 * top and pop are performed on an empty stack.
 */
public class EmptyStackException extends RuntimeException {

    public EmptyStackException(String message) {
        super(message);
    }
}
```

Figure 4.22: An exception is thrown when the methods pop and top are performed on an empty stack.

each method of the array-based implementation of a stack. The methods *size()* and *isEmpty* take O(1) time because they only access the instance variable *top*. The methods *top()* and *pop()* take constant time because they call the method *isEmpty()* and access the top element in the array (the method *pop()* also decreases the instance variable top). All of the previous operations take O(1). Likewise the method *push()* also takes O(1).

Methods	**Time**
isEmtpy(), size()	O(1)
top(), pop()	O(1)
push()	O(1)

Table 4.2: Performance of an array-based implementation of a stack

Therefore, the array-based implementation is simple and efficient. However, the main drawback of this implementation is that it is necessary to know the maximum size of the stack. In some cases, this may cause an unnecessary waste of memory or an exception when the stack reaches this maximum size and it is not possible to add store elements.

Another implementation that does not have the size limitation is to use a linked list to represent a stack. That is, the elements of a stack are stored into the nodes of a linked list. Of course, we may use the *java.util.ArrayList* class or any API java class implementing lists to represent the stack, however, we provide ourselves implementation of a linked list in order to improve your knowledge and practice about linked lists. First of all, we define a java class to implement a generic node for a singly linked list (see Figure 4.25). Figure 4.26 shows the code of the linked list-based implementation. This class defines the instace variable *top* that stores the top element of the stack. We have decided

```java
package lists;
/**An array-based implementation to represent a stack * */
public class ArrayStack<E> implements Stack<E> {
    /**This constant is the maximum possible size of the array*/
    public final static int MAXCAPACITY=1000;

    /**Array for storing the elements of the stack*/
    protected E stack[];
    /**index for the top element in the stack*/
    protected int top=-1;
    /**stores the actual capacity (number of elements) of the stack*/
    protected int capacity=-1;

    /**Constructor. Initially, it creates an array of a given size*/
    public ArrayStack(int s) {
        capacity=s;
        stack=(E[]) new Object[s];
    }

    /**Returns the size of the stack*/
    public int size() { return (top+1); }
    /**Returns true if the stack is empty, eoc false*/
    public boolean isEmpty() { return (top<0);  }

    /**Returns the top element of the stack and remove it.
     * Throws an exception if the stack is empty. */
    public E pop() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty");
        E temp=stack[top];
        stack[top]=null;
        top--;
        return temp;
    }

    /**Returns the top element of the stack without remove it.
    * Throws an exception if the stack is empty. */
    public E top() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty");
        return stack[top];
    }

    /**Add the element e at the top of the stack*/
    public void push(E e) throws FullStackException {
        if (size()==capacity) throw new FullStackException("Stack is full");
        top++;
        stack[top]=e;
    }

}
```

Figure 4.23: An array-based implementation of a stack

that the top of the stack is stored at the head of the linked list. This fact allows to the operations *pop(), top()* and *push()* take constant time (see Table 4.3) because they only need to access the first element (head) of the list. If the top element of the stack was stored at the end of the list, then it would be necessary to trasverse all elements of the list, every time you would need to access the top element. Thus, the previous methods would take O(n) time. We also note that the method *push()* does not throw an exception related to the size overflow problem since in this implementation the size of the stack is not limited.

```java
package lists;
/**
 * This Runtime exception will be thrown when the methods
 * push is called on stack that is full.
 */
public class FullStackException extends RuntimeException {

    public FullStackException(String message) {
        super(message);
    }
}
```

Figure 4.24: An exception is thrown when the method push is performed on a full stack.

```java
/**
 * Example of Node class for a singly linked list of Objects.
 */
public class Node<E> {
    private E element;
    private Node<E> next;
    /**Constructor*/
    public Node(E e, Node<E> n) {
        element=e;
        next=n;
    }
    /**This method returns the element of this node*/
    public E getElement() {     return element;     }
    /**
     * This method returns the next node to this node
     * @return Node
     */
    public Node<E> getNext() {      return next;    }
    /**
     * Methods to modifier the properties of the Node class
     */
    public void setElement(E e) {       element=e;  }
    public void setNext(Node<E> n) {        next=n; }
}
```

Figure 4.25: A java class for implementing a node of a generic singly linked list.

Stacks have many interesting applications. You can find some applications such as reversing arrays or matching parentheses and HTML tags in in Chapter 5 (Stacks and Queues) (pages 199-203) in the book [**?**].

```java
package lists;
/**A linked list-based implementation of a stack.
 * The top element of stack is stored at the beginning of the list*/
public class LinkedListStack<E> implements Stack<E> {
    /**This variable stores the reference to the top element of the stack*/
    protected Node<E> top;
    /**Stores the number of elements of the stack*/
    protected int size=-1;

    /**Constructor. It creates an empty stack*/
    public LinkedListStack() {
        size=0;
        top=null;
    }
    /**Returns the size of the stack*/
    public int size() { return (size); }
    /**Returns true if the stack is empty, eoc false*/
    public boolean isEmpty() { return (size<=0);    }
    /**Returns the top element of the stack and remove it.
     * Throws an exception if the stack is empty. */
    public E pop() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty");
        E temp=top.getElement();
        top=top.getNext();
        size--;
        return temp;
    }
    /**Returns the top element of the stack without remove it.
    * Throws an exception if the stack is empty. */
    public E top() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty");
        return top.getElement();
    }

    /**Add the element e at the top of the stack*/
    public void push(E e) {
        Node<E> newNode=new Node<E>(e,top);
        top=newNode;
        size++;
    }
```

Figure 4.26: A Linked-list based implementation of a stack.

4.5 Queues

Another important linear data structure is the queue. A **queue** is a collection of objects that are managed according to the the **First-In First-out (FIFO)** principle (see Figure 4.27), that is, only element at the front of the queue can be accessed or deleted and new elements must added at the end (*rear*) of the queue. In order to understand better this principle, think about a line of people

Methods	Time
isEmtpy(), size()	O(1)
top(), pop()	O(1)
push()	O(1)

Table 4.3: Running times of the array-based implementation of a stack

waiting a bus. Normally, the first person in the line will be the first one on getting onto the bus. If one arrives last, this should put oneself at the rear of the line. Queues are a nature option of many applications that requiere to process their requests according to FIFO principle, such as reservation systems for airlines, cinemas or many other public services.

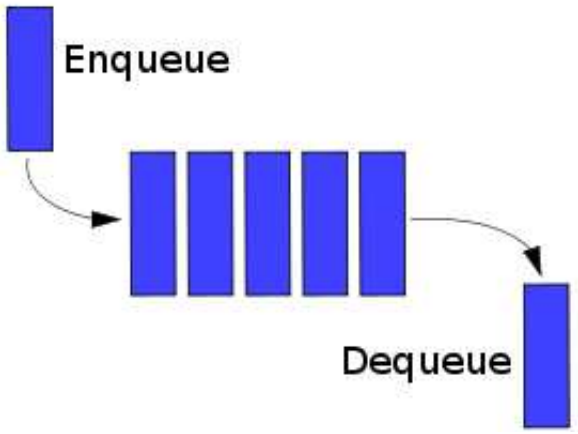


Figure 4.27: Representation of a FIFO Queue.

Formally, a *queue* is an abstract data structure that is characterized by the following operations:

- *enqueue()*: add a element at the rear (end) of the queue.
- *dequeue()*: return and remove the first element of the queue, that is, the element at the front of the queue. If this one is empty, then this method should throw an exception.
- *front()*: return the element at the front of the queue. If this one is empty, then this method should throw an exception.

In addition, similar to the *Stack* ADT, the queue ADT can also include the *size()* and *isEmpty()* methods.

Figure 4.28 shows a java interface for this ADT. It uses the generic parameterized type E, which allows to store elements of any specified class. Also, the

Operation	Queue	Output
enqueue('h')	(h)	-
enqueue('e')	(h,e)	-
front()	(h,e)	h
dequeue()	(e)	h
dequeue()	()	e
enqueue('h')	(h)	-
enqueue('o')	(h,o)	-
enqueue('l')	(h,o,l)	-
front()	(h,o,l)	h
enqueue('l')	(h,o,l,l)	-
front()	(h,o,l,l)	h
enqueue('a')	(h,o,l,l,a)	-
size	(h,o,l,l,a)	6
enqueue('!')	(h,o,l,l,a,!)	-
size	(h,o,l,l,a,!)	7
front()	(h,o,l,l,a,!)	h
dequeue()	(o,l,l,a,!)	h

Table 4.4: A secuence of operations on a queue of characters

EmptyQueueException class has been defined to throw an exception when the methods *dequeue()* and *front()* are called on an empty queue.

4.5.1 A circular array-based implementation of a queue

Likewise with the *Stack* ADT, we can use an array to represent a queue. Thus, elements of a queue are stored in an array. *What is the more efficient option for storing the front of the queue*:

1. at the first position of the array (that is, Array[0]) and adding the following elements from there.

2. as the last element of the array, that is, a new element is always inserted at the first position of the array and the

The former one is not an efficient solution because each time the method *dequeue()* is called, all elements must moved to its previous cell, taking O(n) time. The second one is also an inefficient solution since each time a new element will be inserted (*enqueue()*), elements in the array must be moved to their following position, taking O(n) time.

In order to achieve constant time for the methods of the *Queue* interface, we can use a circular array to store the elements and two instance variables *front* and *rear* to keep the index storing the first element of the queue and the index to store a new element. Each time we remove the first element of the queue, we should increase the variable *front*. Likewise, each time we add a new element,

```java
package lists;
/** Interface for a queue: collection of elements that are managed
 * based on the FIFO principle:first in- first out
 */
public interface Queue<E> {
    /**Returns its size */
    public int size();

    /**Returns true if the queue is empty, otherwise false*/
    public boolean isEmpty();

    /**Returns the element at the front of the queue and remove it.
     * Throws an exception if the queue is empty. */
    public E dequeue() throws EmptyQueueException;

    /**Returns the element at the front of the queue without remove it.
    * Throws an exception if the queue is empty. */
    public E front() throws EmptyQueueException;

    /**Add the element e at the rear of the queue*/
    public void enqueue(E e);

}
```

Figure 4.28: A java interface for the *Queue* ADT.

```java
package lists;

public class EmptyQueueException extends RuntimeException {
    public EmptyQueueException(String message) {
        super(message);
    }
}
```

Figure 4.29: An exception is thrown when the methods *dequeue()* and *front()* are performed on an empty queue.

we store it into the position *rear* at the array and increase the value of this variable.

Figure 4.30 shows three different configuration of a queue implemented using a circular array. The first case ($front \leq rear \leq length(array)$) is the normal configuration. The second and thirds examples illustrate the configuration in which the rear reaches the lenght of the array and it is necessary to store a new element at the first position of the array. When *rear* reaches *front*, it implies that the queue is empty. Each time we need to increase the *rear* or *front*, we must estimate their the module value

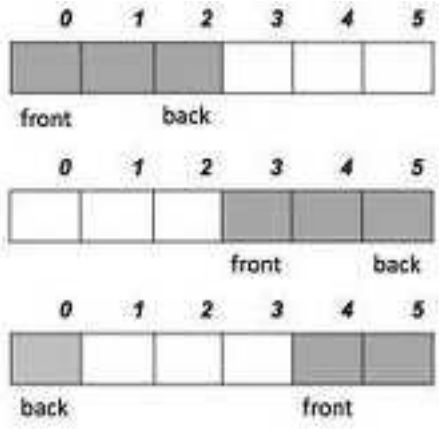


Figure 4.30: Three different configurations of a queue.

Each method in this implementation takes O(1) since they only involve a constant number of arithmetic opreations, comparisons and assignments. The only drawback of this implementation is that the size of the queueu is limited to the size of the array. However, if we are able to provide a good estimation of the size of the queue, this implementation is very efficient. Figure 4.31 shows a circular array-based implementation of a queue.

4.5.2 A linked list-based implementation of a queue

A linked list also provides an efficient implementation of a queue (see Figure 4.32). For efficiency reasons, the front of the queue is stored at the first node of the list and we also define a variable to store the tail of the list. These two variables allow all methods take O(1) time because they only need a constant number of simple statements. The main advantage of this implementation compared to the array-based implementation is that it is no necessary to specify a maximum size for the queue.

Please, write a java program to assign the turn to every journalist in the TV program '59 seconds'. You can find an interesting problem based on the use of a queue in in Chapter 5 (Stacks and Queues) (pages 212) in the book [**?**].

4.6 Double-Ended Queues (Dqueues)

Figure 4.33 compares the three ADT: stack (LIFO: last in, first out), queue (FIFO: first in, first out) and dqueue. A *double-ended queue* ADT (dqueue is pronunced like deck) is power than stack and queue because it supports insertion and deletion at both its front and its rear. The main methods of the dqueue ADT are:

- *addFirst(e)*: insert a new element at the head of the queue.

```java
package lists;
public class ArrayQueue<E> implements Queue<E> {
    protected E queue[];
    protected int front=-1;
    protected int rear=-1;
    public ArrayQueue(int maxElem){
        queue=(E[]) new Object[maxElem];
    }
    public int size() {return (queue.length - front + rear) % queue.length; }
    public boolean isEmpty() {      return (front==rear);   }
    public E dequeue() throws EmptyQueueException {
        if (front == rear)
            throw new EmptyQueueException ("Queue is empty");
        E temp=queue [front];
        front = (front + 1) % queue.length;
        return temp;
    }
    public E front() throws EmptyQueueException {
        if (front == rear)
            throw new EmptyQueueException ("Queue is empty");
        return queue [front];
    }
    public void enqueue(E e) throws FullQueueException  {
        if (size()== queue.length-1) throw new FullQueueException("Queue is full");
        queue [rear] = e;
        rear = (rear + 1) % queue.length;
    }
}
```

Figure 4.31: A circular array-based implementation of a queue.

- *addLast(e)*: insert a new element at the end of the queue.
- *removeFirst()*: remove the element at the front of the queue. If the queue is empty, then an exception is thrown.
- *removeLast()*: remove the element at the end of the queue. If the queue is empty, then an exception is thrown.
- *getFirst()*: return the element at the head of the queue. If the queue is empty, then an exception is thrown.
- *getLast()*: return the element at the end of the queue. If the queue is empty, then an exception is thrown.
- size(): return the size of the queue.
- isEmpth(): return a boolean indicating if the queue is empty.

The *java.util.LinkedList¡E¿* class already defines all the methods of a dequeue. Of course, you can use this class when you need to use a dequeue in your future applications. But first, you must learn how the dequeue ADT can be defined and implemented. Figure 4.34 contains a java interface for the dqueue

```java
public class LinkedListQueue<E> implements Queue<E> {
    /**This variables point to the head and tail of the queue.
     * For efficiency reasons, the front is stored into the first node*/
    protected Node<E> front, tail;
    protected int size=-1;
    public LinkedListQueue() {
        size=0;
        front=tail=null;
    }
    public E dequeue() throws EmptyQueueException {
        if (isEmpty()) throw new EmptyQueueException("Queue is empty");
        E temp=front.getElement();
        front=front.getNext();
        size--;
        if (size==0) tail=null;
        return temp;
    }
    public void enqueue(E e) {
        Node<E> newNodo=new Node(e,null);
        if (isEmpty()) front=newNodo;
        else tail.setNext(newNodo);
        tail=newNodo;
        size++;
    }
```

Figure 4.32: A Linked List-based implementation of a queue.

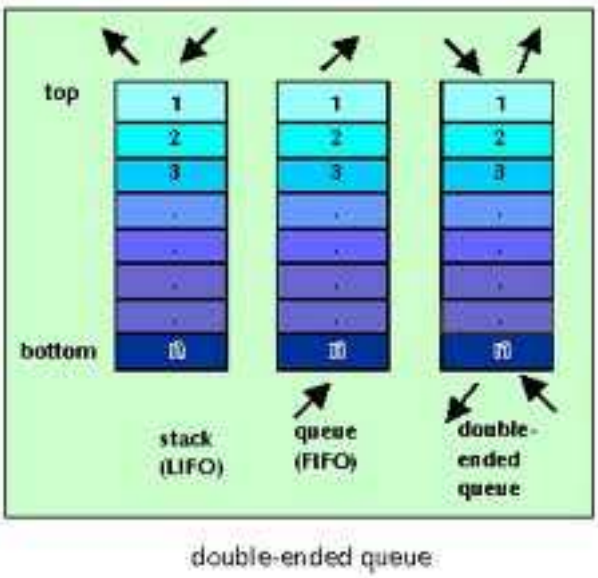


Figure 4.33: The dequeu ADT a queue that allows invertion and deletion at both its front and its rear (source: http://t3.gstatic.com/.

Operation	Queue	Output
addFirst('k')	(k)	-
addLast('l')	(k,l)	-
removeFirst()	(l)	-
addLast('o')	(l,o)	-
addFirst('a')	(a,l,o)	-
removeFirst()	(l,o)	-
removeLast()	(o)	-
removeLast()	empty	-
removeFirst()	empty	exception
isEmpty()	empty	true
addLast('s')	(s)	-
isEmpty()	(s)	false

Table 4.5: Secuence of operations on a dqueue of characters

ADT. A dequeu is a list of elements, hence we may use a linked list to implement a dqueue. A singly linked list is not an efficient solution because the dqueue allows insertion and removal at both the head and the tail of the list. While the insertion or removal of the first element at the dqueue just take O(1), the insertion or removal of the last element take O(n) because we should trasverse all nodes until to reach the last node. However, if we implement the dqueue ADT using a doubly linked list, all insertion and removal operations take O(1) times.

Figure 4.36 shows a fragment of the implementation of a deque using a doubly linked list. This class defines two sentinal nodes to reference the head and tail of the deque. Figure 4.35 shows the implementation for a node of a doubly linked list.

```java
package lists;
/**Interface for a double-ended queue.
 * The java.util.LinkedList class implements these methods*/
public interface Dqueue<E> {
    /**Returns its size */
    public int size();

    /**Returns true if the dqueue is empty, otherwise false*/
    public boolean isEmpty();

    /**Inserts an element at the beginning of the dqueue.*/
    public void insertFirst(E e);

    /**Inserts an element at the end of the dqueue.*/
    public void insertLast(E e);

    /**Returns the element at the front of the dqueue.
     * Throws an exception if the dqueue is empty. */
    public E removeFirst() throws EmptyDqueueException;

    /**Returns the element at the end of the dqueue.
     * Throws an exception if the dqueue is empty. */
    public E removeLast() throws EmptyDqueueException;

    /**Returns the element at the beginning of the dequeu.
    * Throws an exception if the dqueue is empty. */
    public E getFirst() throws EmptyDqueueException;

    /**Returns the element at the end of the dequeu.
    * Throws an exception if the dqueue is empty. */
    public E getLast() throws EmptyDqueueException;
```

Figure 4.34: An interface for a double-ended queue ADT.

```java
package lists;

/**A class for nodes in a doubly linked list*/
public class DoublyNode<E> {
    protected E element;
    /**references to the next and previous nodes*/
    protected DoublyNode<E> next;
    protected DoublyNode<E> prev;
    /**Constructor*/
    public DoublyNode(E e, DoublyNode<E> p, DoublyNode<E> n) {
        element=e;
        prev=p;
        next=n;
    }
    /**This method returns the element of this node*/
    public E getElement() {return element;}
    /** This method returns a reference to its next node*/
    public DoublyNode<E> getNext() {return next;}
    /** This method returns a reference to its previous node*/
    public DoublyNode<E> getPrev() {return prev;}
    /**Methods set*/
    public void setElement(E e) {element=e;}
    public void setNext(DoublyNode<E> n) {next=n;}
    public void setPrev(DoublyNode<E> p) {prev=p;}
}
```

Figure 4.35: This class implements a node of a doubly linked list.

```java
package lists;
/**A doubly-linked list based implementation of a deque*/
public class DoublyLListDeque<E> implements Dqueue<E> {
    /**We define two sentinal nodes to reference the head and tail of the list */
    protected DoublyNode<E> header, tailer;
    protected int size;
    /**Constructor*/
    public DoublyLListDeque() {
        header=new DoublyNode<E>();
        tailer=new DoublyNode<E>();
        /**header and tailer must point to  each other*/
        header.setNext(tailer);
        tailer.setPrev(header);
        size=0;
    }
    /**Returns the last element at the deque.*/
    public E getLast() throws EmptyDqueueException {
        if (isEmpty()) throw new EmptyDqueueException("Deque is empty");
        /**tailer points to the last node of the deque by its reference prev*/
        return tailer.getPrev().getElement();
    }
    /**Removes the first element at the deque.*/
    public E removeFirst() throws EmptyDqueueException {
        if (isEmpty()) throw new EmptyDqueueException("Deque is empty");
        /**header points to the first node of the deque by its reference next*/
        DoublyNode<E> firstNode=header.getNext();
        E temp=firstNode.getElement();
        /**The current second node will be the first node*/
        DoublyNode<E> secondNode=firstNode.getNext();
        header.setNext(secondNode);
        secondNode.setPrev(header);
        size--;
        return temp;
    }
    /**Inserts the element e at the end of the deque*/
    public void insertLast(E e) {
        DoublyNode<E> oldLastNode=tailer.getPrev();
        DoublyNode<E> newLastNode=new DoublyNode<E>(e,oldLastNode,tailer);
        oldLastNode.setNext(newLastNode);
        tailer.setPrev(newLastNode);
        size++;
    }
```

Figure 4.36: A doubly linked list class for implementing a deque.

Chapter 5

Trees.

Most of the information has been sourced from the books [**?**, **?**].

5.1 General Trees

Trees are a natural organization for data and support algorithms much faster than the linear data structures. They are widely used in computing to represent file systems, web sites, databases, graphical user interfaces, etc.

A tree is an abstract data type to store elements that have hierarchical relationships between them. Figure 5.1 shows the tudor family tree whose root is the node 'Eduardo III'. This node has three children 'Eduardo','Juan' and 'Edmundo' (they are siblings because they are children of the same parent). The parent (direct ancestor) of the node 'Enrique VI' is the node 'Enrique V'. The organigram of a company, an arithmetic expression or the rules of a grammar can be also represented by trees (see Figure 5.2, 5.3 and 5.4).

5.1.1 Properties

A tree is a set of nodes (elements) that maintain a hierarchical (*parent-child*) relationship between them. The top element of a tree is called *root*. The following properties must be satisfied:

- Every node has zero or more children. Nodes without children are called *leaves* or *externals*, while nodes with children are *internals*.
- Every node, except the root, has an unique parent node.

An *empty tree* is a tree that does not contain any nodes. We can formally define *ancestor* and *descendent* terms:

- u is *ancestor* of v (v is *descendent* of u) $\leftrightarrow$ u=v or u is ancestor(parent(v)).
- u is *descendent* of v if v is *ancestor* of u.

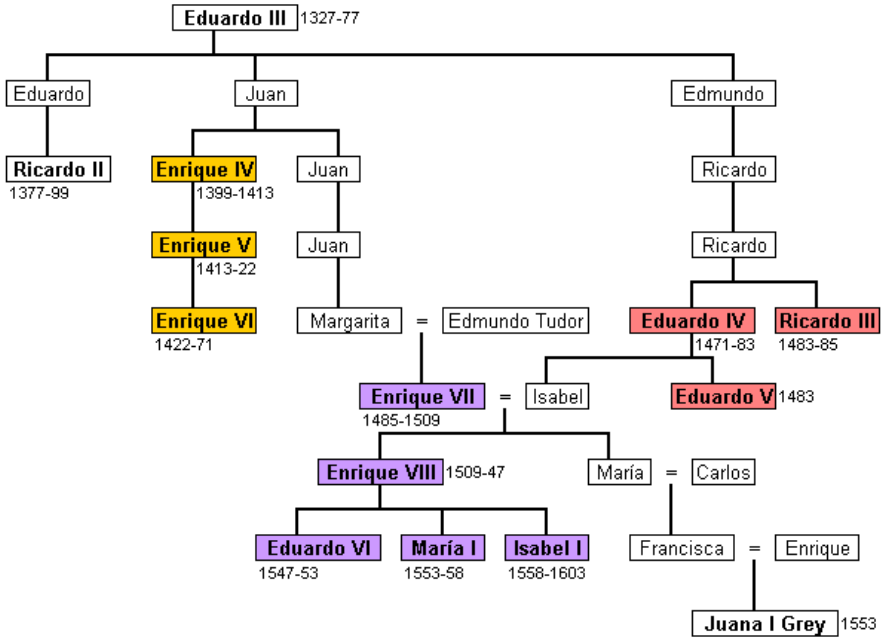


Figure 5.1: Tudor family tree.

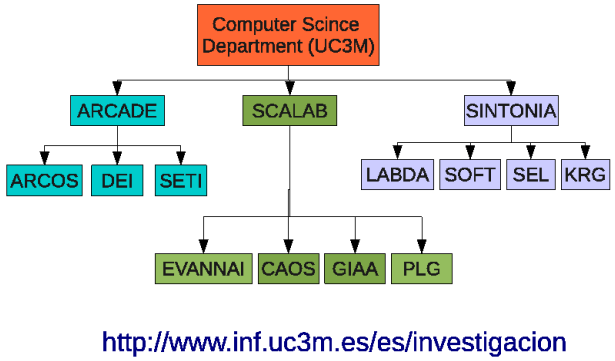


Figure 5.2: UC3M Computer Science Department's organigram.

Figure 5.1 shows that 'Eduardo IV' and 'Ricardo III' are descedent of 'Edmundo'. You can also see that 'Enrique VII' is ancestor of 'Isabel I'.

A tree is *ordered* if there is a linear order among siblings. For example, Figure 5.5 shows an ordered tree of integers.

The *depth* of a node is the number of its ancestors. The depth of the root is 0. For example, in the tree of Figure 5.5, the node storing the value *12* has

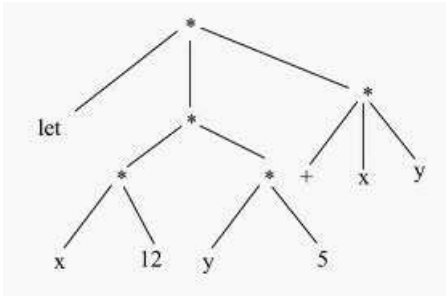


Figure 5.3: An arithmetic expression.

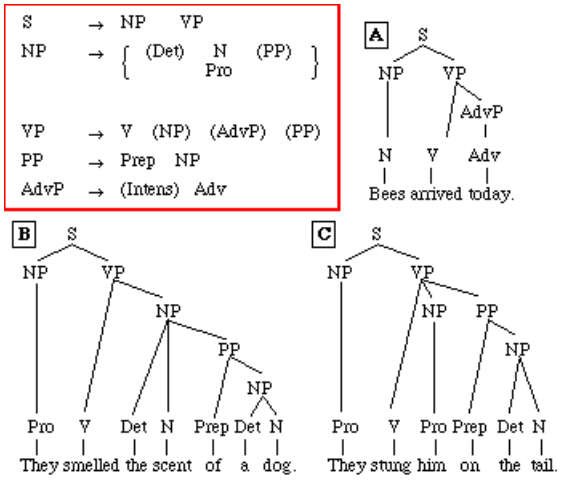


Figure 5.4: A sintax grammar.

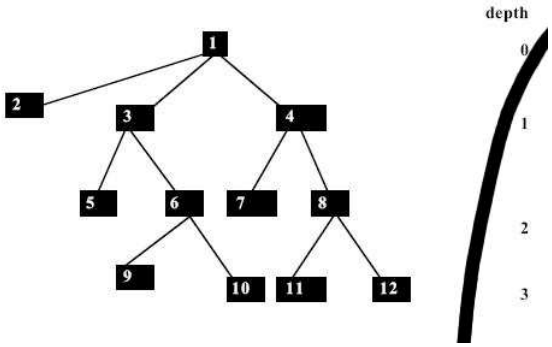


Figure 5.5: An ordered tree of integers.

depth 3, the node storing *6* has depth 2 and the node storing *4* has depth 1.

5.1.2 Tree Abstract Data Type

A tree should support the following methods:

- *isEmpty()* returns true if the tree is empty.
- *isRoot(v)* returns true if the node v is the root of the tree.
- *isLeave(v)* returns true if v is a leave.
- *isInternal(v)* returns true if v is an internal node.
- *root()* returns the root of the tree. If the tree is empty, an error occurs.
- *parent(v)* returns the parent of the node v. If v is the root, an error occurs.
- *children(v)* returns the children of the node v. If the tree is ordered, this methods stores the children in order.
- *size()*: returns the number of nodes in the tree.
- *iterator()*: returns an interator of all the elements stored at nodes of the tree.
- *positions()*: returns an iterable collection of all the nodes of the tree.
- *replace(v,e)*: replaces the node v with the node e.

5.1.3 Implementing a Tree

First of all, we must define a java interface to represent the tree ADT (see Figureẽffig:interfaceTree). You can see that its methods may throw the exception *InvalidTreePositionException* if the node is invalid (null). Also, if the tree is empty, the method root throws the exception *EmptyTreeException*, and the method parent throws the exception BoundaryViolationException if the node is the root of the tree.

A common way to implement a tree is to use a linked structure in which each node n has the following properties (see Figure 5.7):

- The element stored at the node.
- A link to its parent. If n is the root, then this field is *null*.
- A collection (for example, an array or a list) containing the links to its children.

```java
package trees;
import java.util.Iterator;
/**An interface for a tree where nodes can have an arbitrary number of children*/
public interface Tree<E> {
    /**Returns the number of nodes of the tree*/
    public int size();
    /**Returns true if the tree is empty, false e.o.c*/
    public boolean isEmpty();
    /**Returns an iterator of the elements stores in the tree*/
    public Iterator<E> iterator();
    /**Returns an iterator of the nodes that stores the elements in the tree*/
    public Iterator<NodeTree<E>> positions();
    /**Returns the root of the tree. If the tree is empty throws an exception*/
    public NodeTree<E> root() throws EmptyTreeException;
    /**Returns the parent of the node v.
     * If v is the root throws the exception BoundaryViolationException.*/
    public NodeTree<E> parent(Position<E> v)
        throws InvalidTreePositionException, BoundaryViolationException;
    /**Returns an iterable collection of the children of a given node*/
    public Iterable<Position<E>> children(Position<E> v)
        throws InvalidTreePositionException;
    /**Returns true if the node v has children.*/
    public boolean isInternal(Position<E> v) throws InvalidTreePositionException;
    /**Returns true if the node v is a leave.*/
    public boolean isLeave(Position<E> v) throws InvalidTreePositionException;
    /**Returns true if the node v is the root of the tree.*/
    public boolean isRoot(Position<E> v) throws InvalidTreePositionException;
    /**Replaces the element stored in the node v by the element e*/
    public E replace(Position<E> v, E e)  throws InvalidTreePositionException;
}
```

Figure 5.6: An interface for a tree.

5.2 Tree Traversal Algorithms

This section presents algorithms for traversing a tree. Before we should define two concepts like **depth** and **height**:

- The **depth** of a node v is the number of ancestors of v. For example, the depth of the node '1' in the tree shown in figure 5.5 is 0 because this node is the root of the tree. However, for node '10' its depth is 3, because this node has three ancestors. Its implementation is shown in Figure 5.9

Its running time is $O(d_v)$ where d_v is the depth of the node v, because the algorithm only performs a constant-time recursive step for each ancestor of v.

The height of a node v in a tree can be defined recursively as follows:

- if v is a leave node then its height is 0.
- if v is an internal node then its height is 1 plus the maximum height of a child of v.

Therefore, the height of a Tree is the height of its root. For example, the height of the tree shown in Figure 5.5 is 4 because its root has depth equals to 4. The height of the node '6' is 1 since all its nodes ('9' and '10') are already leave nodes. Its implementation is shown in Figures 5.10 and **??**.

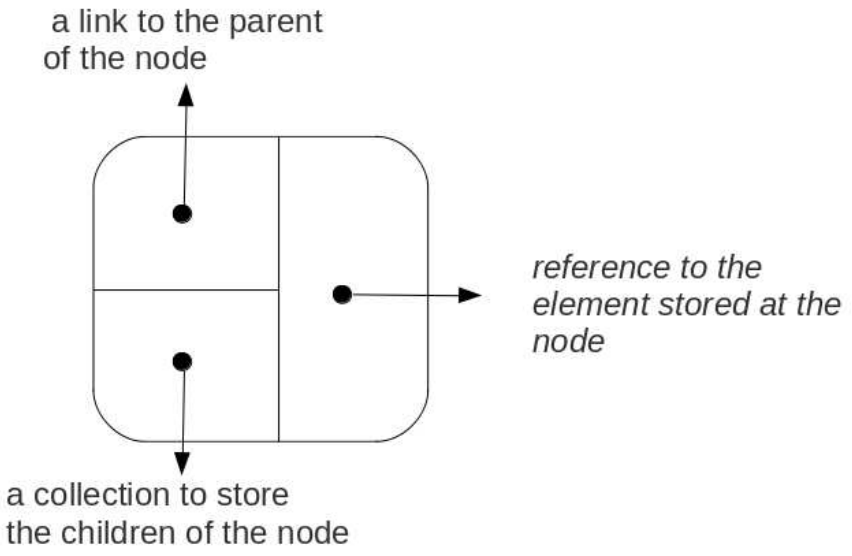


Figure 5.7: Each node of a tree is represented as a node with the fields: its value, link to its parent and a collection of its children.

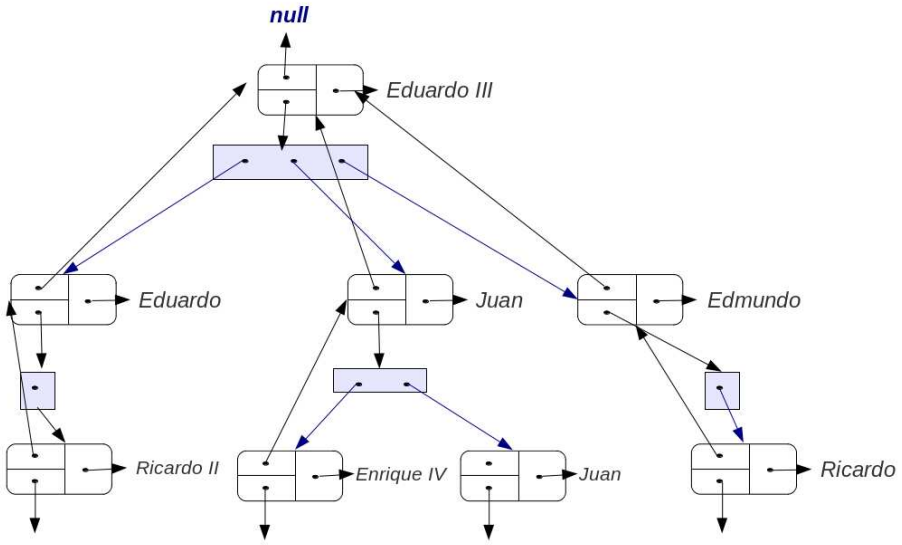


Figure 5.8: This figure shows the representation using a linked structure of the general tree shown in Figure 5.1. It only shows the third first levels of the tree.

Methods	Time
isEmtpy(), size()	O(1)
parent(), root	O(1)
isLeave(), isInternal(), isRoot()	O(1)
positions(), iterator()	O(n)
replace()	O(1)
children(n)	O(num of children of v)

Table 5.1: Performance of an linked structure-based implementation of a binary tree

```
/**
 * computes the depth of a node v, that is, the number of ancestors of v
 * @param T
 * @param v
 * @return
 */
public static <E> int depth(Tree<E> T, NodeTree<E> v) {
    if (T.isRoot(v)) return 0;
    else return 1 + depth(T, T.parent(v));
}
```

Figure 5.9: Implementation of the depth method.

```
/**
 * computes the height of a node v in the tree T. If the v is a leave then its height is 0.
 * If the v is an internal node then its height is 1 + maximum height of all its children nodes
 * @param <E>
 * @param T
 * @param v
 * @return
 */
public static <E> int height(Tree<E> T, NodeTree<E> v) {
    if (T.isLeave(v)) return 0;

    int h=0;
    for(NodeTree<E> w:T.children(v)) h=Math.max(h,height(T,w));
    return h;
}
```

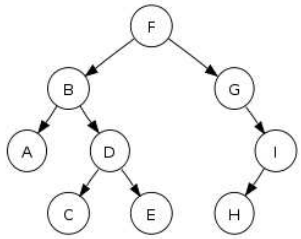
Figure 5.10: Implementation of the height of a node.

5.3 Preorder Traversal

In a preorder traversal, first we must visit the root, then we must visit the subtrees of its children. If the tree is ordered then this output is also ordered. Figure 5.13 shows the implementation of this preorder traversal.

```java
/**
 * computes the height of a T.
 * @param T
 * @return
 */
public static <E> int height(Tree<E> T) {
    int h=0;
    for(NodeTree<E> w:T.nodes()) h=Math.max(h,height(T,w));
    return h;
}
```

Figure 5.11: Implementation of the height of a tree.



In this binary search tree

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right); note how this produces a sorted sequence
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

Figure 5.12: Preorder Traversal.

```java
/**
 * A public method to visit nodes of T in a preorder way.
 * @param <E>
 * @param T
 * @return
 */
public static <E> String toStringPreorder(Tree<E> T) {
    return preorder(T,T.root);
}
private static <E> String preorder(Tree<E> T, NodeTree<E> v) {
    String s=v.element.toString();
    for (NodeTree<E> node:T.children(v)) s = s+ ", " + preorder(T,node);
    return s;
}
```

Figure 5.13: Implementation of the preorder Traversal.

5.4 Postorder Traversal

In a postorder traversal, first we must visit the children of the root (in a postorder way), and finally we must visit the root. Figure 5.14 shows the implementation of this postorder traversal.

```
public static <E> String toStringPostorder(Tree<E> T) {
    return postorder(T,T.root);
}
private static <E> String postorder(Tree<E> T, NodeTree<E> v) {
    String s="";
    for (NodeTree<E> node:T.children(v)) s = s + preorder(T,node)+ ", ";
    s=s+v.element;
    return s;
}
```

Figure 5.14: Implementation of the postorder Traversal.

Chapter 6

Binary Trees

Most of the information has been sourced from the books [**?**, **?**]

6.1 Definition.

The order of a **binary** tree is always 2, that is, every node has at most two children (usually called *left child* and *right child*). Given an internal node v, the subtree rooted by its left child is called *left subtree* and the one rooted by its right child is called *right subtree*. For example, Figure 6.1 shows a simple binary tree rooted with a node whose value is 2. The left child of the root node is the subtree rooted by the node whose value is 7, while the right child is the subtree rootd by the node with value 5.

Besides, a binary tree can be defined in a recursive fashion. A binary tree is either empty, or is made of a single node, whose the left and right children are binary trees too.

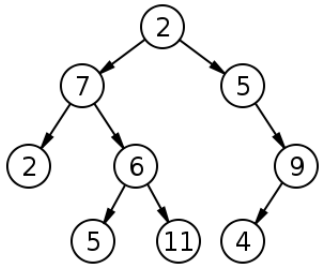


Figure 6.1: Example of a simply binary tree.

A binary tree is ordered if for every node v, the values in its left subtree are less than the value in v, and all values in its right subtree. The binary tree shown in Figure 6.1 is not ordered, but Figure 6.2 shows an ordered binary tree because all their nodes fulfill alphabetical order.

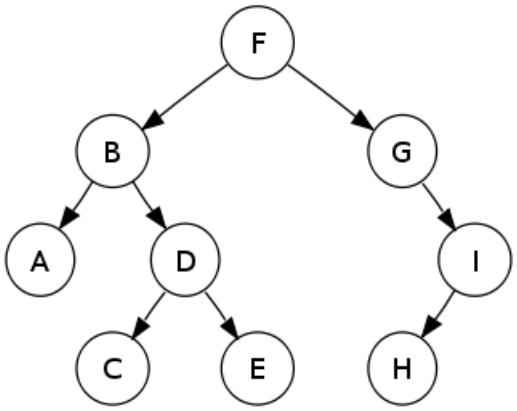


Figure 6.2: Example of an ordered binary tree.

A *full* binary tree (also called *proper*) is a tree in which every internal node has two children (see Figure 6.3). The trees shown in Figures 6.1 and 6.2 are not full binary trees, because both have internal nodes with only one child.

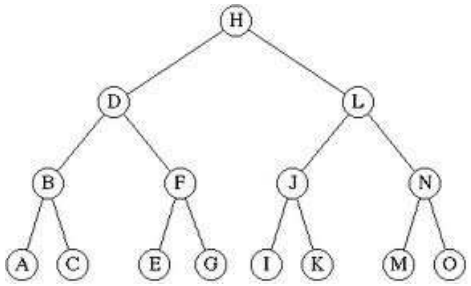


Figure 6.3: Example of a full (proper) binary tree.

The full binary trees are often used to represent decision trees. These are models to assist the decision maker in finding the

Full binary trees are often used to represent decision trees. A decision tree is a model of decision to assist the decision maker in finding the option depending on whether the answer is 'Yes' or 'No'. Figure 6.4 illustrates a decision tree that allows to order three elements A, B and C. The internal nodes represent comparison operations and the leaves represent the possible outcomes.

Besides, binary trees can be useful to represent arithmetic expressions. The internal nodes are operators, while the external nodes represent variables or constants. The value of an internal node can be calculated by applying its operation to the values of its children. Figures 6.5 and 6.6 show two binary trees representing arithmetic expresion. The former three has variables and constants in its leaves, but the second one only has constants. The value associated with

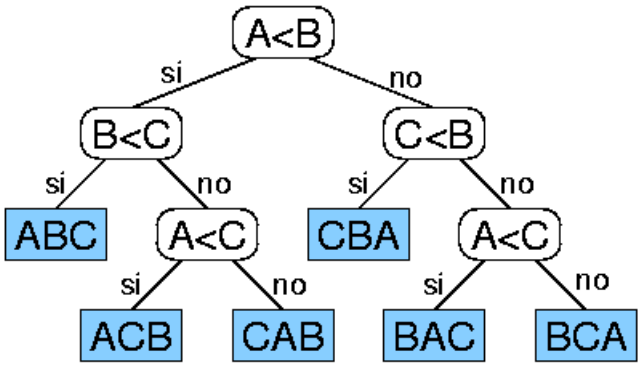


Figure 6.4: Example of a decision tree to order three elements A, B and C.

each internal node is shown next to each node. For example, the root has the value 4 (the final outcome of the arithmetic expression).

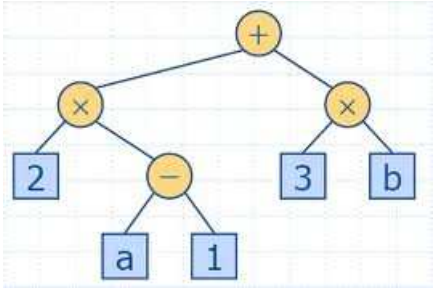


Figure 6.5: Example of a binary tree to represent the arithmetic expression: 2 x (a-1) + 3 x b.

6.2 The binary tree ADT

A binary tree is a specialization of a tree. Also, the definition of a binary tree requires at least the following methods:

- hasLeft(v): returns true if the node v has left child.
- hasRight(v): returns true if the node v has right child.
- left(v): returns the left child of the node v. If v does not have left child, an error occurs.
- right(v): returns the right child of the node v. If v does not have left child, an error occurs.

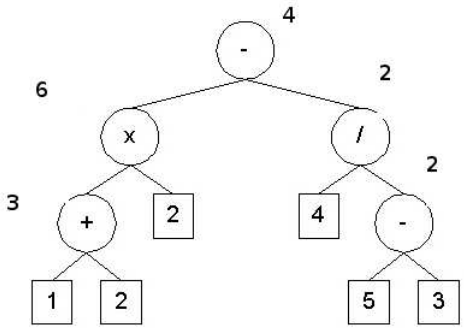


Figure 6.6: Example of a binary tree to represent the arithmetic expression: (1+2) x 2 - 4 / (5-3).

You may want to have an additional field to store its parent

Figure 6.7 shows an interface for the binary tree ADT. This interface extends the interface Tree(see code fragment in Figure 5.6). If the tree is ordered, then the method children(v) returns first the left child followed by the right one.

```
package trees;
/**This interface is a  specialization of a tree.
 * Every node can have 0, 1 or 2 children*/
public interface BinaryTree<E> extends Tree<E> {
    /**Given a node n, this method returns the left child of n*/
    public Position<E> left(Position<E> n) throws InvalidTreePositionException,
                                                  BoundaryViolationException;
    /**Given a node n, this method returns the right child of n*/
    public Position<E> right(Position<E> n) throws InvalidTreePositionException,
                                                   BoundaryViolationException;
    /**Given a node n, this method checks if node v has left child*/
    public Position<E> hasLeft(Position<E> n) throws InvalidTreePositionException;
    /**Given a node n, this method checks if node v has right child*/
    public Position<E> hasRight(Position<E> n) throws InvalidTreePositionException;
}
```

Figure 6.7: An interface for the binary tree ADT.

6.3 Properties of a binary tree

Firstly, we must define two concepts: **level** (or depth) of a node and **height** a tree.

Given a node v, we can define its level in a recursive fashion, as follows:

- If v is the root of the tree, its level is 0.
- otherwise, its level (or level) is 1 + the level of its parent node.

Figure 6.8 represents an ordered binary tree and shows the level for each node.

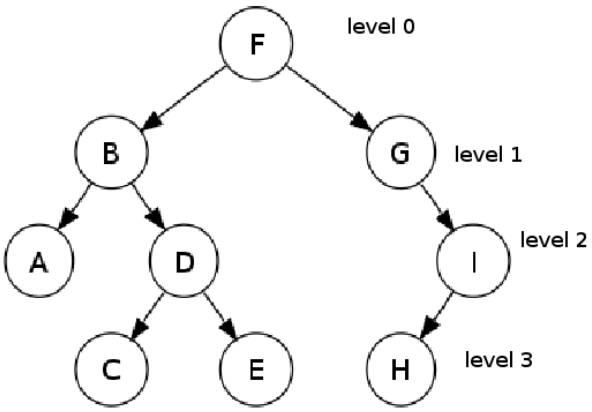


Figure 6.8: Level (or depth) of a node.

The height of a tree can be defined as 0 if the tree is empty, and otherwise, as 1 + plus the maximum value between the height of the left subtree of its root and the height of the right subtree of its root. Figure 6.9 shows the methods to calculate these properties of a tree.

```
/**Given a node v, calculates its level or depth*/
public int level(NodeTree<E> n) {
    if (n==root) return 0;
    else return 1 + level(n.getParent());
}
/**Returns the height of a tree, that is, the maximum level + 1*/
public int height() {
    if (isEmpty()) return 0;
    else {
        CBinaryTree<E> leftChild=new CBinaryTree<E>(left(root));
        CBinaryTree<E> rightChild=new CBinaryTree<E>(left(root));
        return (1 + Math.max(leftChild.height(), rightChild.height()));
    }
}
```

Figure 6.9: Implementation of the methods to obtain the height of a tree and the level of a node.

The level 0 of a nonempty binary tree only has one node (root) ($=2^0$), the first level has at most two nodes ($=2^1$), the second level has at most four nodes

(=2^2), the third level hast at most eight nodes (=2^3), and so on. You can see that the maximum number of nodes on levels grows exponentially. In general, a binary tree in its leven n has at most 2^n nodes.

Now, you must try to demostrate the following properties:

1. **A full binary tree of height h has ($2^(h+1)-1$) nodes**.

Let #T denotes the number of nodes of T and #Level$_i$ the number of nodes in the level i. Then,

$$\#T = \#Level_0 + \#Level_1 + \#Level_2 + \ldots + \#Level_h = 2^0 + 2^1 + 2^2 + \ldots + 2^h \tag{6.1}$$

You can find a tip in the Appendix A: Useful Mathematical Facts [**?**] to solve this equation. In particular, you should use the proposition A.14:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \tag{6.2}$$

$$\#T = 2^0 + 2^1 + 2^2 + \ldots + 2^h = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1 \tag{6.3}$$

2. **In an full binary tree, the numer of external nodes (leaves) is 1 more than the number of internal nodes**.

Let e, i denotes the number of leaves and the number of internal nodes in the tree T, respectively. So, we must demostrate that:

$$e = i + 1 \tag{6.4}$$

To justify this property, we are applying the induction principle (please, you must study and look up Section 4.3.3 Induction and Loop Inva riants in the book [**?**]). Concisely, given a statement P(n), the induction principle proves its correctness for n=1 (2,3). Then, this principle assumes that the statement is held for an arbitrary n, and tries to prove for n+1.

You can easily check (by drawing trees) the following statements.

- If the tree has 1 nodes, the property 2 is satisfied.
- If the tree has 3 nodes, the property 2 is satisfied.
- If the tree has 5 nodes, the property 2 is satisfied.

Note tha the tree neve has a pair number of nodes because it is a full binary tree. Now, we assume true for a tree with n nodes ((a) $e_n = i_n + 1$). What does it happen if we add new external nodes?. We cannot add only one leave becuase this violate the property of full binary tree. So, we must add two nodes. It is clear that a leave turns into an internal node in the new tree, that is, (b) e_{n+2}=e_n-1 and (c) i_{n+2}=i_n+1:

$$e_{n+2} =_{(b)} (e_n - 1) + 2 = e_n + 1 =_{(a)} i_n + 1 + 1 =_{(c)} i_{n+2} + 1 \tag{6.5}$$

3. **The number of leaves satisfies the following equation**:

$$h \leq e \leq 2^h \tag{6.6}$$

You can use the induction principle to prove $h \leq e$.

- if $h = 1$, it is obvious that $1 \leq e$, becuase $n = e = 1$.
- Now, we assume that $h \leq e$, and we must prove for $h + 1 \leq e_{new}$, that, it is the number of leaves in the new tree. If we increase the height of the tree, we must add at leat two nodes and a leave turns into internal node, therefore, (a) $e_{new} = (e - 1) + 2$

$$h + 1 \leq_{h \leq e} e + 1 = (e - 1) + 2 =_{(a)} e_{new} \tag{6.7}$$

Now, we will demostrate the second part of the equation ($e \leq 2^h$).

$$n = e + i =_{2ndproperty} e + (e - 1) = 2e - 1 =_{1stproperty} 2^{h+1} - 1 \tag{6.8}$$

$$2e = 2^{h+1} \Leftarrow e = 2^h \tag{6.9}$$

For every nonfull tree, it is obvious that $e \leq 2^h$

The following properties can be proved based on the three previous properties. Please, try yourself!!!:

4. **The number of internal nodes satisfies the following equation:**

$$h \leq i \leq 2^h - 1 \tag{6.10}$$

5. **The total number of nodes satisfies the following equation:**

$$2h + 1 \leq n \leq 2^{h+1} - 1 \tag{6.11}$$

6. **The height (h) of a tree satisfies the following equation:**

$$log_2(n + 1) - 1 \leq h \leq \frac{n - 1}{2} \tag{6.12}$$

7. **The height (h) of a tree satisfies the following equation:**

$$log_2(e) \leq h \leq e - 1 \tag{6.13}$$

6.4 Linked Structure-based implementation of binary tree

To implement a binary tree, we can use a linked structured of nodes to represent each node of the tree. Each node (see Figure 6.10) the linked structured has the following fields:

- the element stored in the node.
- a reference to its parent. If the node is the root, then this filed is null.
- a reference to its left child. If the node does not have left child, this field is null.
- a reference to its right child. If the node does not have right child, this field is null.

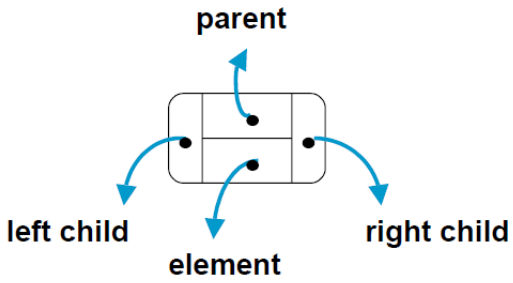


Figure 6.10: A node of a linked structure to represent a node of a binary tree.

Figure6.11 shows the representation of a binary tree (whose root is A) by a linked list of nodes. Figure 6.12 shows an interface for representing binary tree nodes. It has methods to set and return the parent, the left child, the right child and the element stored at a node. Figure 6.13 shows the clas BTNode which contains four fields: an element, its parent, its left child and its right child. Also, this class implements the methods defined in the interface BTPosition.

Now, we are defining the class *LinkedBinaryTree* that stores a reference to the root of the tree and also the total number of nodes (size). This class implements the interface BinaryTree (see Figure 6.7). Also, the class has a constructor without arguments that returns an emtpy tree. Besides, the additional methods are defined:

- *addRoot(e)*: creates a new node for storing the element e. This new node is the root of the tree. The method only works when the tree is empty, otherwise an error will occur.[1]

[1]If we define a constructor with a node as input argument that sets the root as this node, then we do not need the method addRoot.

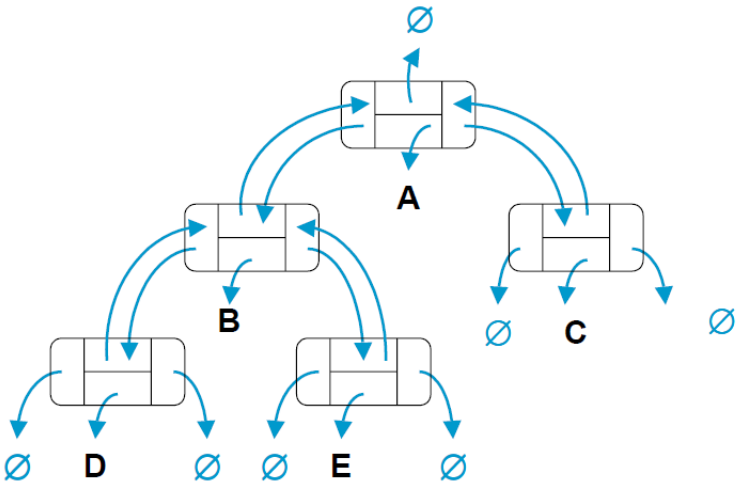


Figure 6.11: A linked structure for representing a binary tree.

- *addLeft(n,e)*: creates a new node for storing the element e and adds this new node as the left child of the node n. If the n already has a left child, an error will occur.

- *addRight(n,e)*: creates a new node for storing the element e and adds this new node as the right child of the node n. If the n already has a right child, an error will occur.

- *attach(n,* T_1, T_2*)*: attaches the binary trees T_1,T_2 as left and right subtrees of the leave n, respectively. If n is not an external node, then an error will occur.

- *height()*: returns the height of the tree.

These additional methods allow us to build a binary tree by creating the root using the method addRoot and adding the left and right children using the methods addLeft and addRight, repeatedly. You can find the implementation of the class `LinkedBinaryTree.pdf`LinkedBinaryTree. Please, add code to the main method to create a binary tree and test the class. For example, you can try to build the previous examples presented for this section.

Table **??** summarizes the computational complexity for each method in a linked structure implementation of a binary tree.

The method *size()* takes O(1) time because it only uses the instance variable *size*. The methods *isEmpty(),isRoot(),getRoot()* take O(1) time since they only access the instance variable *root*. The methods *hasLeft(), left(), hasRight(), right()* take O(1) because they only access the instance variables left,right,parent (of the BTNode class), respectively. Likewise, the methods sibling(), isLeave(),

```java
/**An interface for representing a node of a binary tree*/
public interface BTPosition<E> {

    /**Returns the element stored at the node*/
    public E getElement();
    /**Returns the parent of the node*/
    public BTPosition<E> getParent();
    /**Returns the left child of the node.
     * If the node is root, then returns null*/
    public BTPosition<E> getLeft();
    /**Returns the right child of the node.
     * If the node does not have left returns null*/
    public BTPosition<E> getRight();

    /**Sets the element stored at this node*/
    public void setElement(E e);
    /**Sets the parent of this node*/
    public void setParent(BTPosition<E> p);
    /**Sets the left node of this node*/
    public void setLeft(BTPosition<E> l);
    /**Sets the right node of this node*/
    public void setRight(BTPosition<E> r);

    }
```

Figure 6.12: An interface to represent a node of a binary tree.

isInternal() takes O(1) time becuase they only access instances variables. Since the method *children()* just need to access two instance variables (left and right children of a give node), it only takes O(1) time. The method *positions* uses an recursive method *preorderPositions* that traverses the tree and stores its nodes in a list. Thus, *positions* takes O(n) time. Likewise, *iterator* also takes O(n) since it uses the method *positions*. The methods *replace* and *addRoot* takes O(1) time because they access and use one node. The methods *insertLeft*, *insertRight* and *remove* takes O(1) time because they access and modify a constant number of nodes.

6.5 Array-based implementation of a binary tree

Another alternative to implement a binary tree is to store the nodes of the tree in an array. The root of the tree is stored in the first position in the array, its left child in the second position, its right child in the third position, and so on (see Figure 6.14).

Since we cannot know the maximum size that the tree may reach, the best

```java
package trees;
/**This class implements a node of a  binary tree. The node stores a reference
 * to an element, a reference to its parent, a reference to its left child, and
 * a reference to its right child.*/
public class BTNode<E> implements BTPosition<E> {
    private E element;
    private BTPosition<E> parent;
    private BTPosition<E> left;
    private BTPosition<E> right;
    public BTNode(E e, BTPosition<E> p, BTPosition<E> l, BTPosition<E> r){
        setElement(e);
        setParent(p);
        setLeft(r);
        setRight(r);
    }
    /**Returns the element stored at the node*/
    public E getElement() { return element; }
    /**Returns the parent of the node*/
    public BTPosition<E> getParent() {  return parent;  }
    /**Returns the left child of the node.
     * If the node is root, then returns null*/
    public BTPosition<E> getLeft() { return left;   }
    /**Returns the right child of the node.
     * If the node does not have left returns null*/
    public BTPosition<E> getRight() {   return right; }
    /**Sets the element stored at this node*/
    public void setElement(E e) {element=e; }
    /**Sets the parent of this node*/
    public void setParent(BTPosition<E> p) { parent=p;  }
    /**Sets the right node of this node*/
    public void setRight(BTPosition<E> r) { right=r;    }
    /**Sets the left node of this node*/
    public void setLeft(BTPosition<E> l) { left=l; }
```

Figure 6.13: A class for implementing binary tree nodes.

option is to use an ArrayList to store its nodes (we do not recommend to use the zero-positon of the arraylist).

It is possible to know the position of a given node from the position of its parent. For every node, Figure 6.14 clearly demostrates the following claims:

- The position of its left child in the arraylist is: *pos(left)=2*pos(node)*.
- The position of its right child in the arraylist is: *pos(right)=2*pos(node)+1*

Now, you should give the arraylist-based implementation of a binary tree. Then, you will have to estimate the running times of its main methods.

Methods	Time
isEmtpy(), isRoot(), size()	O(1)
hasLeft(), getLeft(), hasRight(), getRight, parent(), sibling	O(1)
isLeave(), isInternal()	O(1)
positions(), iterator()	O(n)
replace(), addRoot()	O(1)
insertLeft(), insertRight(), remove()	O(1)

Table 6.1: Performance of an linked structure-based implementation of a binary tree

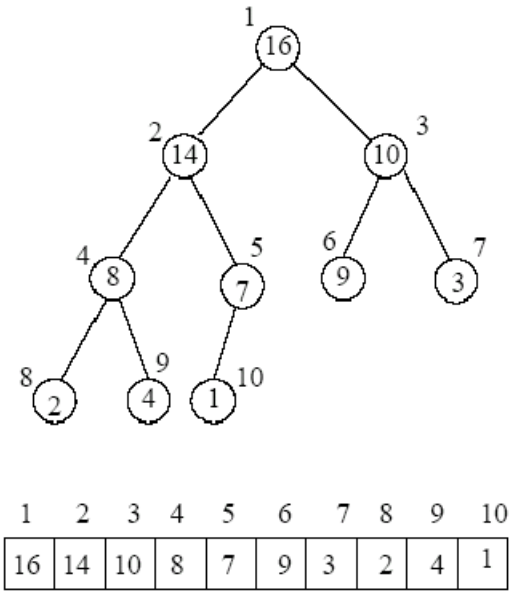


Figure 6.14: Running times for a binary tree implemented with an linked structure

Methods	Time
isEmtpy(), isRoot(), size()	O(1)
hasLeft(), getLeft(), hasRight(), getRight, parent()	O(1)
isLeave(), isInternal()	O(1)
positions(), iterator()	O(n)
replace(), addRoot()	O(1)
insertLeft(), insertRight(), remove()	O(1)

Table 6.2: Running times for a binary tree implemented with an arraylist

Chapter 7

Binary Search Trees (BST).

7.1 Definition.

Most of the information has been sourced from the books [**?**, **?**].

A binary search tree is a binary tree T such that each internal node v of T stores an entry (k,x) such that:

- Keys stored at nodes in the left subtree of v are less than or equal to k.
- Keys stored at nodes in the right subtree of v are greater than or equal to k.

Keys provide a way of performing a search by making a comparison at each internal node v, which can stop at v or continue at v's left or right child. Binary trees are an excellent data structure for storing the entries of a dictionary, assuming we have an order relation defined on the keys. The main property is the possibility of storing the keys in an ordered way. An inorder traversal of the nodes of a binary search tree should visit the keys in nondecreasing order. Each node may store a key and a value. Key and value may be objects from different classes. For example, If a binary search tree is used to store a telephone directory, its nodes may consist of a String key (name) and a Long value (the phone number). Keys cannot be null. Binary search trees may store duplicate keys depending on our decision. Figure7.1 shows two examples of binary search trees with numeric keys and without values.

7.2 Implementation of a binary search tree.

Figure7.3 shows the implementation of a node for a binary search tree. You can observe that the node has an object Key of the class E (any class), an object value of the class F (any class), a reference to its parent node, and references to its left and right children, respectively. Figure 7.2 shows two examples of trees that are not binary search trees because their keys are not ordered.

Some of the possible operations of the Binary Search Tree ADT are:

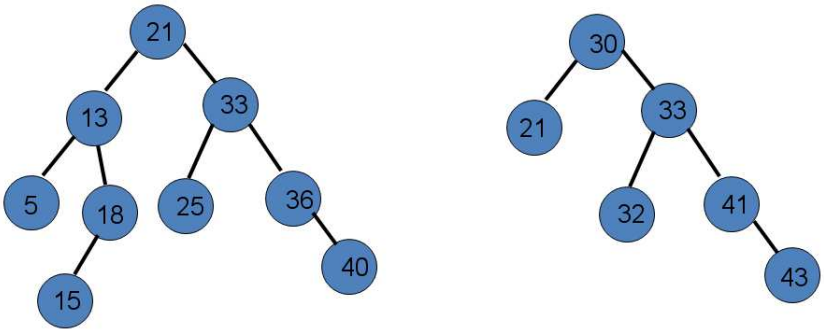


Figure 7.1: Example of binary search trees.

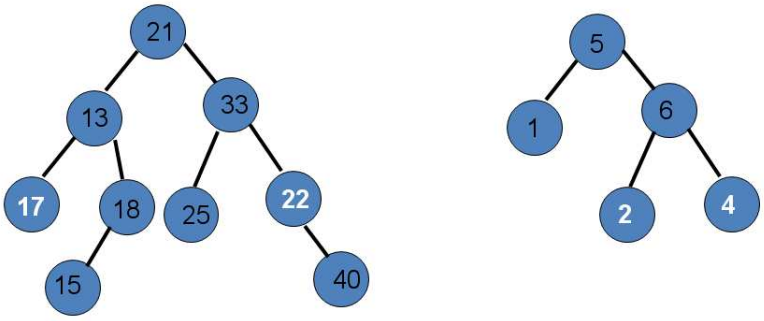


Figure 7.2: These trees are not binary search trees.

- isEmpty(): tests if the tree is empty.
- clear(): sets the tree to null (that is, its root is null and size sets to 0).
- firstKey(): returns the first key in the tree (that is, the lowest key). Figure 7.4 shows the implementation of this method.
- lastKey(): returns the last key in the tree (that is, the highest key).
- getValue(E k): returns the value stored in the node whose key is k; e.o.c it returns null.
- find(E k): this method returns a node with key k, if it exists. Returns null if k is not found. Due to the ordered keys of the tree, it is needed to test if the search key is less than, equal to, or greater than the key stored at node v (see Figure 7.5). If k is smaller then the search continues in the left subtree; if k is equal then the search terminates successfully; if k

```java
/**Implementation of a node for a binary search tree * */
public class BBNodeTree<E,F> {
    E key;
    F value;
    BBNodeTree<E,F> parent;
    BBNodeTree<E,F> left;
    BBNodeTree<E,F> right;

    public BBNodeTree(E k, F v, BBNodeTree<E,F> p,
            BBNodeTree<E,F> l, BBNodeTree<E,F> r) {
        key=k;
        value=v;
        parent=p;
        left=l;
        right=r;
    }
}
```

Figure 7.3: Implementation of a node of a binary search tree.

is greater then the search continues in the right subtree. If we reach an empty (null) node, then the search terminates unsuccessfully. Figure 7.6 shows the implementation of its operations.

- findAll(k): this method returns a list of all nodes with keys equal to k.

Figure 7.7 shows the recursive implementation of the method find (called searchNode). Worst-case running time of searching in a BST is simple. The algorithm searchNode is recursive and executes a constant number of primitive operations for each recursive call. Each recursive call of searchNode is made on a child of the previous node. That is, searchNode is called on the nodes of a path of T that starts at the root and goes down one level at a time. Thus, the number of such nodes is bounded by h + 1, where h is the height of T. For example, we would spend O(1) time per node encountered in the search, method find on tree T runs in O(h) time, where h is the height of the binary search tree T. Admittedly, the height h of T can be as large as n, but we expect that it is usually much smaller. findAll(k) has O(h+s) where s is the number of returned nodes. Indeed, we will show how to maintain an upper bound of O(logn) on the height of a search tree T using AVL. There are additional methods for searching through predecessors and successors of a key or entry, but their performance is similar to that of find.

Binary search trees allow implementations of the insert and remove operations using algorithms that are fairly straightforward, but not trivial. We now explain some operations more difficult than the previous ones.

- insert(k,x): inserts a node with key k and value x (see Figure 7.8). If there

```java
public class ABB<E,F>  {
    BBNodeTree<E,F> root;
    int size;
    public ABB() {
        root=null;
        size=0;
    }

    /**returns the first key at the tree (that is the lowest key)*/
    public E firstKey() {
        if (isEmpty()) return null;
        else {
            BBNodeTree<E,F> f=root;
            while (f.left!=null) f=f.left;
            return f.key;
        }
    }
```

Figure 7.4: This implementation returns the lowest key in a tree.

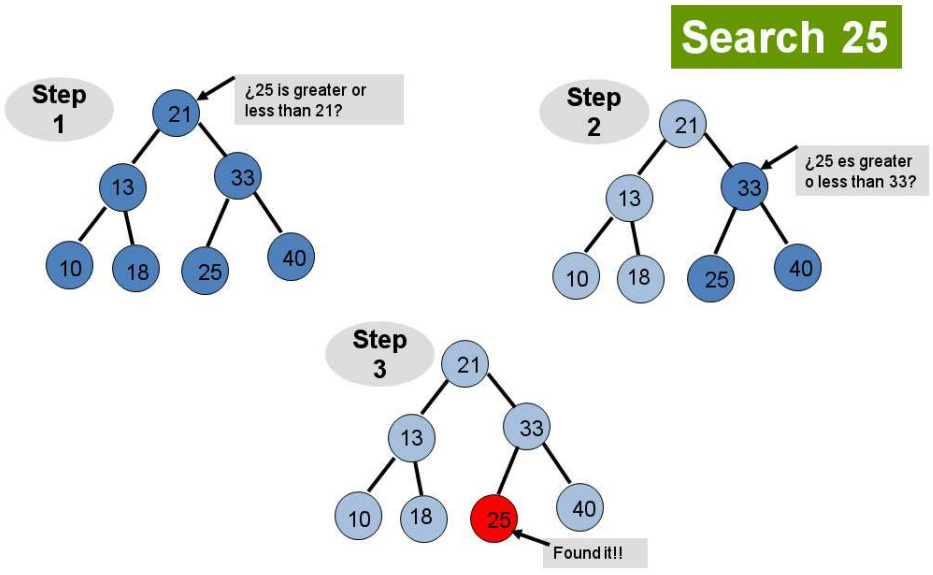


Figure 7.5: Searching a node whose key is 25.

is some node with this key, then its value is replaced by x. To insert a new node in the tree, we have to find the proper place to set the node. For this purpose, we will traverse the tree by means of a recursive method shown in Figure 7.9.

```java
/** Returns the first node with key equals to k in the tree
 * @param k
 * @return
 */
public BBNodeTree<E,F> find(E k) {
    BBNodeTree<E,F> node = root;
    int c;
    while (node!=null) {
        c=((Comparable)k).compareTo(node.key);
        if (c==0) break;
        else if (c<0) node=node.left;
        else node=node.right;
    }
    return node;
}
```

Figure 7.6: The implementation of the iterative find.

```java
public BBNodeTree<E,F> searchNode(E key){
    return searchNode(key,root);
}

private BBNodeTree<E,F> searchNode(E key, BBNodeTree<E,F> node){
    if(node==null)return null;
    int c=((Comparable)k).compareTo(node.key);
    if(c==0) return node;
    else if (c<0)searchNode(key, node.left);
    else if (c>0)searchNode(key, node.right);
}
```

Figure 7.7: The recursive implementation of the method find.

- remove(k): Removes a node with key equals to e, and return it.
- removeAll(k); removes all nodes whose keys are equals to e.

The implementation of the remove(k) operation is a bit more complex, since we do not wish to create any "holes" in the tree. We begin our implementation of operation removeNode(k) using a recursive method that traverses the tree T searching a node w that has the key k. If there is no node with key k in the tree T, we return null (and we are done). If there is a node w, then we wish to remove it and we distinguish three cases (of increasing difficulty):

- If the node w is a leaf, then we should find its parent and free the node w (see Figure 7.10). Its implementation is shown in Figure 7.11.

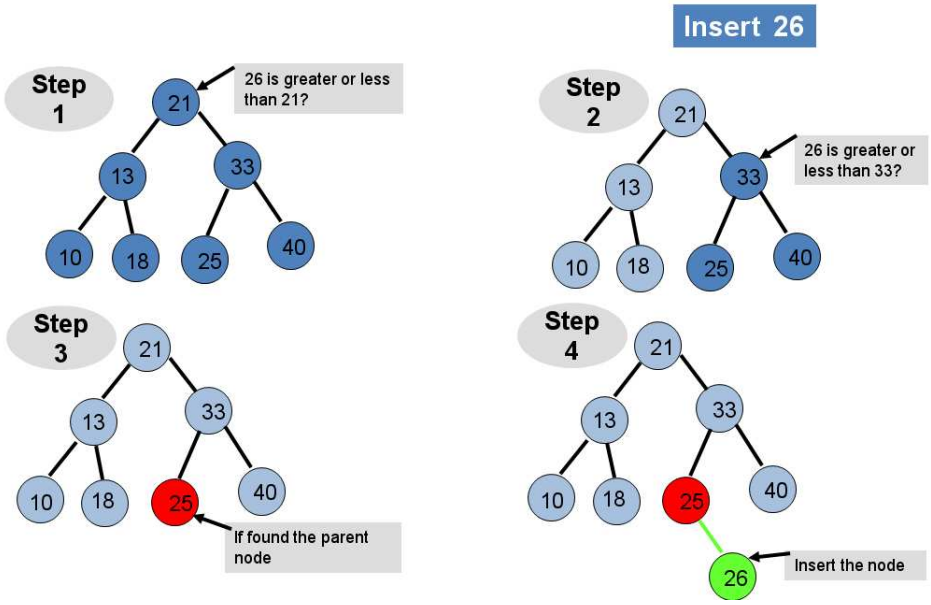


Figure 7.8: Inserting a node whose key is 26.

- If one of the children of node w is null, we simply remove w and restructure T by replacing w with its non-null child. (See Figure 7.12). Its implementation is shown in Figure 7.13.

- If both children of node w are internal nodes (non-null), we cannot simply remove the node w from T, since this would create a "hole" in T. Instead, we proceed as follows (see Figure 7.14; its implementation is shown in Figure 7.15.):

  - We find the first node y that follows w in an inorder traversal of T. Node y is the left-most internal node in the right subtree of w, and is found by going first to the right child of w and then down T from there, following left children.
  - We save the entry stored at w in a temporary variable t, and move the entry of y into w. This action has the effect of removing the former entry stored at w.
  - We remove node y from T. This action replaces y with its right child.
  - We return the entry previously stored at w, which we had saved in the temporary variable t.

As with searching and insertion, this removal algorithm traverses a path from the root to an external node, possibly moving an entry between two nodes of this path.

```
public F insert(E k, F v) {
    BBNodeTree<E,F> search=root;
    BBNodeTree<E,F> parent=null;
    int c=0;
    F tmp=v;
    while (search!=null) {
        parent=search;
        c=((Comparable)k).compareTo(search.key);
        if (c==0) {
            tmp=search.value;
            break;
        } else if (c<0) search=search.left;
         else search=search.right;
    }

    if (search==null) {
        if (parent==null) addRoot(k,v);
        else {
            BBNodeTree<E,F> newNode = new BBNodeTree<E,F>(k,v,parent,null,null);
            if (c<0) parent.right=newNode;
            else parent.left=newNode;
            size++;
        }
    }
    return tmp;
}
```

Figure 7.9: Implementation of the method insert.

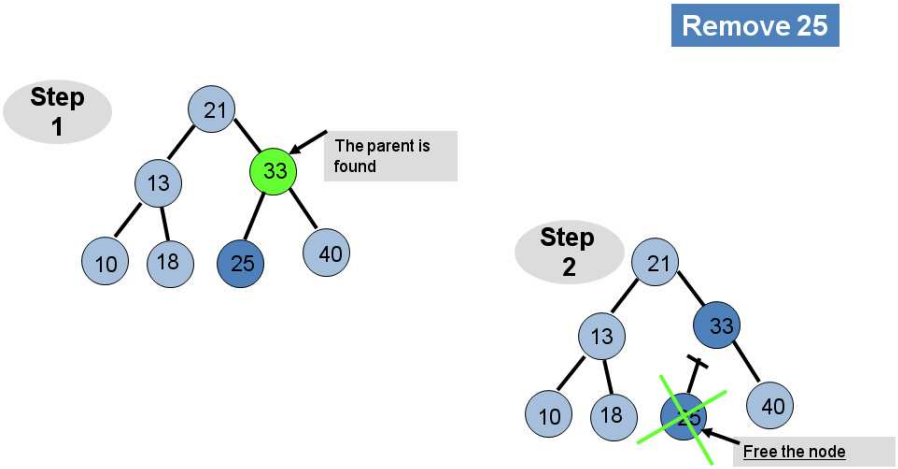


Figure 7.10: Removing a leaf.

```
public F remove(E k) {
    if (isEmpty()) return null;

    BBNodeTree<E,F> nodeRemove=root;
    return delete(k,nodeRemove);
}

private F delete(E k, BBNodeTree<E,F> nodeRemove) {
    int c=((Comparable)k).compareTo(nodeRemove.key);
    if (c<0) return delete(k, nodeRemove.left);
    else if (c>0) return delete(k, nodeRemove.left);

    F tmp=nodeRemove.value;

    BBNodeTree<E,F> p=nodeRemove.parent;

    if (isLeaf(nodeRemove)) {
            nodeRemove.parent=null;
            if (p.left==nodeRemove) p.left=null;
            else if (p.right==nodeRemove) p.right=null;
```

Figure 7.11: Part of the implementation of the method remove (case: removing a leaf node).

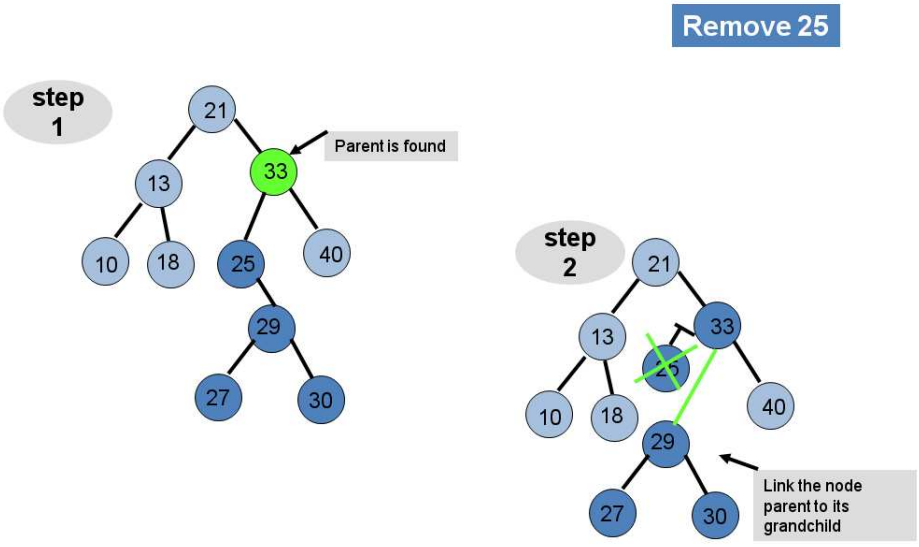


Figure 7.12: Removing a node with an only node.

7.3 Performance of a Binary Search Tree

The analysis of the search, insertion, and removal algorithms are similar. We spend O(1) time at each node visited, and, in the worst case, the number of

```
} else if (!hasLeft(nodeRemove)) {
        nodeRemove.parent=null;
        BBNodeTree<E,F> newChild=nodeRemove.right;
        p.right=newChild;
        newChild.parent=p;
        size--;
} else if (!hasRight(nodeRemove)) {
        nodeRemove.parent=null;
        BBNodeTree<E,F> newChild=nodeRemove.left;
        p.left=newChild;
        newChild.parent=p;
        size--;
} else {
```

Figure 7.13: Part of the implementation of the method remove (case: removing a node with an only child).

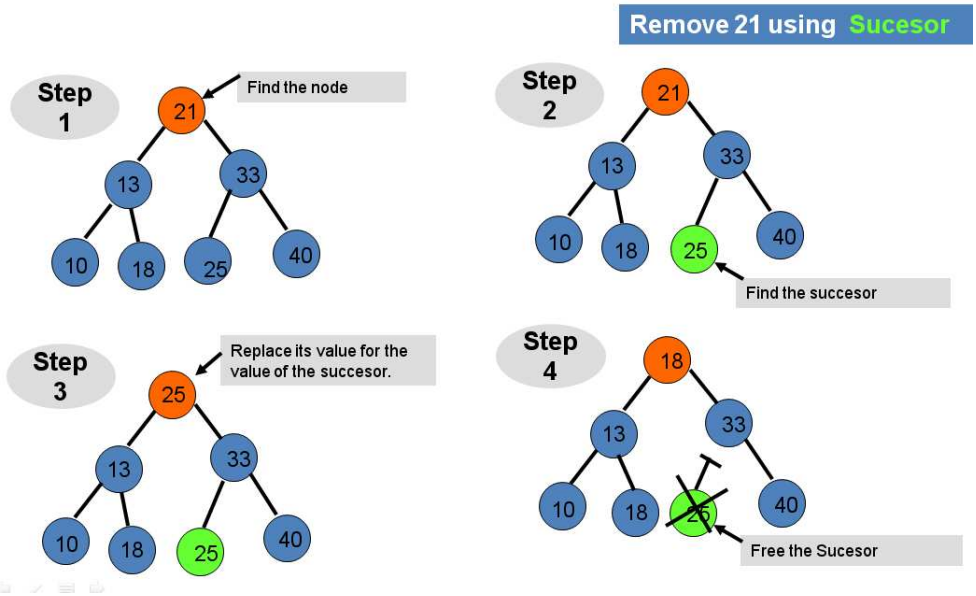


Figure 7.14: Removing a node with two children.

nodes visited is proportional to the height h of T. Thus, a binary search tree T is an efficient implementation of a dictionary with n entries only if the height of T is small. In the best case, T has height h = log(n + 1), which yields logarithmic-time performance for all the update operations. In the worst case, however, T has height n, in which case it would look and feel like an ordered list implementation. Such a worst-case configuration arises, for example, if we insert a series of entries with keys in increasing or decreasing order. This case is shown in Figure 7.16.

```
        size--;
} else {
        //Look for the succesor of the node (firstKey in its right child) or its predeces
        //(lastKey in its left child) and replace it by it
        //We decide to use the succesor
        BBNodeTree<E,F> sucessor=nodeRemove.right;
        while (sucessor.left!=null) sucessor=sucessor.left;
        //Copy the key and the value of the succesor to the nodeRemove
        nodeRemove.key=sucessor.key;
        nodeRemove.value=sucessor.value;
        //We should free the sucessor
        if (sucessor==nodeRemove.right) {
            sucessor.parent=null;
            nodeRemove.right=nodeRemove.right.right;
        }               else {
            BBNodeTree<E,F> parentSuc=sucessor.parent;
            sucessor.parent=null;
            parentSuc.left=null;
        }
        size--;


}
```

Figure 7.15: Part of the implementation of the method remove (case: removing a node with two children).

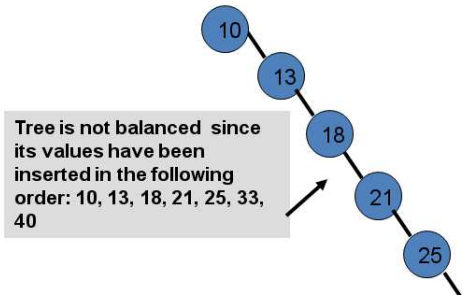


Figure 7.16: Part of the implementation of the method remove (case: removing a node with two children).

The performance of a binary search tree is summarized in Table 7.1. Should be taken into account that search and update operations varies dramatically depending on the tree's height. We can assume that, on average, a binary search tree with n keys has expected height O(log(n)).

Methods	**Time**
isEmtpy(), size()	O(1)
find, insert, remove	O(h)
findAll()	O(h+s)

Table 7.1: Performance of a binary search tree

Chapter 8

AVL Trees.

8.1 Definition.

An AVL tree is a self-balancing binary search tree, where the heights of the two child subtrees of any node differ by at most one. Insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Any binary search tree T that satisfies the height-balance property is said to be an AVL tree, named after the initials of its inventors: Adel'son-Vel'skii and Landis.

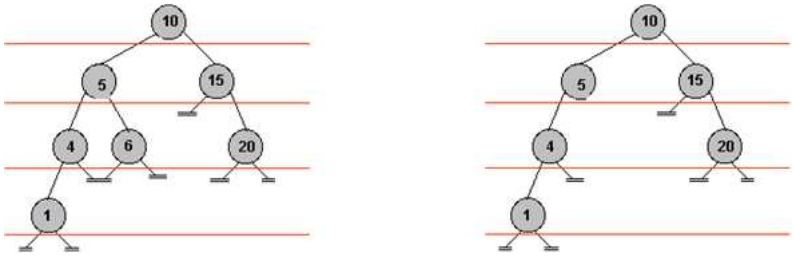


Figure 8.1: The second tree is not AVL because the Left-height and the Right-height of the node 5 differ by 2 nodes.

The balance factor of a node is the height of its left subtree minus the height of its right subtree (sometimes opposite) and a node with balance factor 1, 0, or 1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees. Formally, the balance factor of a node can be defined as follows:

$$Bf = Hr - Hl \tag{8.1}$$

where H_R is the height of its right subtree and H_L is the height of its left subtree.

Based on the possible values of Bf, we can define the following cases:

- If $Bf = 0$ the the left and the right subtrees of a node are the same height.
- If Bf =1 then the tree is balanced in height, but the right subtree is a higher level.
- If $Bf = -1$ then the tree is balanced in height, but the left subtree is a higher level.
- If $Bf \preceq 2$ or $Bf \succeq 2$ then the tree must be balanced.

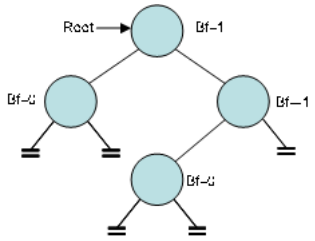


Figure 8.2: Balance factor of a node.

The insertion and removal operations for AVL trees are similar to those for binary search trees, but with AVL trees we must perform additional computations called rotations.

8.2 Operations

8.2.1 Insertion

An insertion in an AVL tree T begins as in an insert operation for a (simple) binary search tree. After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains 1, 0, or +1 then no rotations are necessary. However, if the balance factor becomes 2 then the subtree rooted at this node is unbalanced.

For example, Figure 8.3 shows a sequence of integer nodes (40, 33, 46, 6, 8, 24, 18, 22, 25, 60) inserted in an empty AVL tree. The balance factor of the node 33 is $Bf = -2$, this means that the tree must be balanced.

There are four cases which need to be considered for balancing an AVL tree, of which two are symmetric to the other two. These cases are shown as follows:

Right-Right Simple Rotation (RR) In Figure 8.4, the node a has $Bf = -2$. In order to balance the tree, b becomes the new root, a becomes the left child of b, c becomes the right child of b.

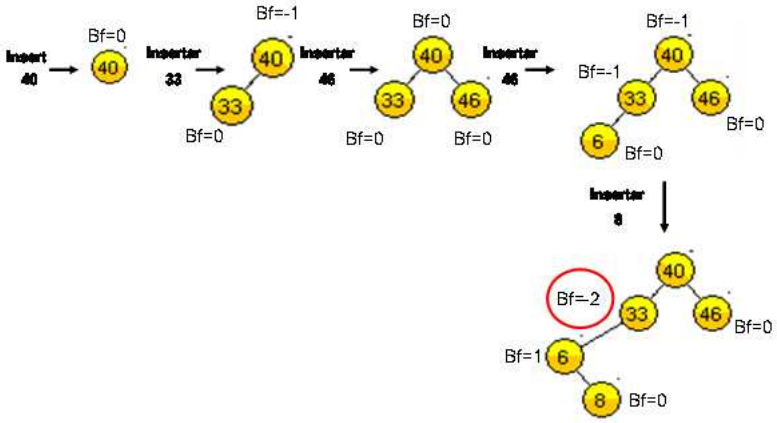


Figure 8.3: Insertion in an AVL tree.

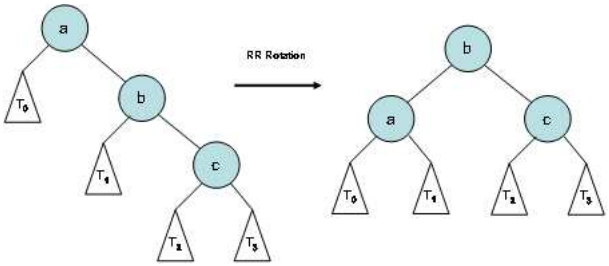


Figure 8.4: Right-Right Simple Rotation.

Left-Left Simple Rotation (LL) In Figure 8.5, the node a has $Bf = 2$. In order to balance the tree, b becomes the new root, a becomes the right child of b, c becomes the left child of b.

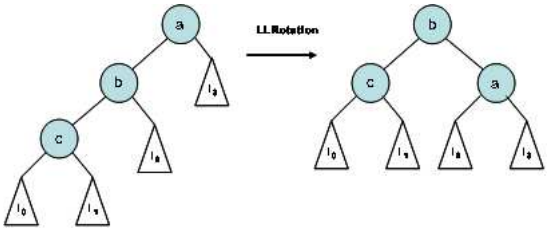


Figure 8.5: Left-Left Simple Rotation.

Right-Left Simple Rotation (Double RL) In Figure 8.6, it is necessary two rotations in order to balance the node a:

- First rotation: c becomes the right child of a, b becomes the right child of c.
- Second rotation: c becomes the new root, a becomes the left child of c.

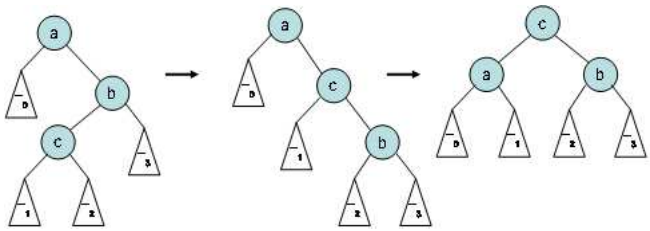


Figure 8.6: Right-Left Simple Rotation.

Left-Right Simple Rotation (Double LR) In Figure 8.7, it is necessary two rotations in order to balance the node a:

- First rotation: c becomes the left child of a, b becomes the left child of c.
- Second rotation: c becomes the new root, a becomes the right child of c.

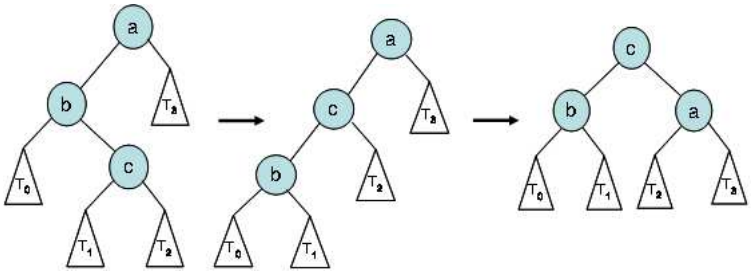


Figure 8.7: Left-Right Simple Rotation.

8.2.2 Deletion

If the node is a leaf or has only one child, remove it. Otherwise, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as a replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed. Fox example, if the node whose value is 5 is deleted from the tree shown in Figure 8.8, the root has $Bf = 2$ and we should balance it by a Right-Right rotation.

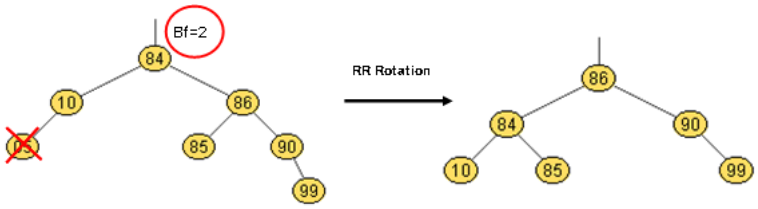


Figure 8.8: Deletion of a leaf node in an AVL tree requiring a RR rotation.

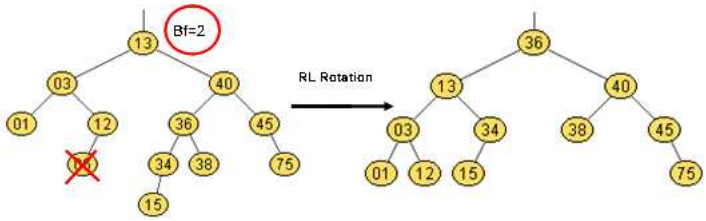


Figure 8.9: Deletion of a leaf node in an AVL tree requiring a RL rotation .

8.2.3 An example

Check if the binary tree show in Figure 8.11 is AVL:

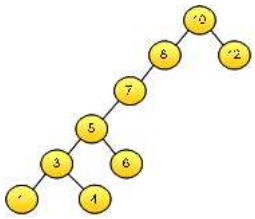


Figure 8.10: is this binary search AVL?.

We must calculate the balance factors of each node, as follows:

Since tree is not an AVL tree (Bf of the node 7 is -3(, this means that the tree must be balanced as follows:

1. RR Rotation is applied on the node 7, (a=7, b=5, c=3) the result of the new tree is shown in Figure **??**:

We must check checked the tree shown in Figure 8.12 to see if it is an AVL tree or not. The tree is not an AVL tree because the balance factor of the node 8 is $Bf = -3$, this means that the tree must be balanced. A RR Rotation is applied on the node 8, (a=8, b=5, c=3) the result of the new tree is shown in Figure 8.14. Once checked the new tree, it is an AVL

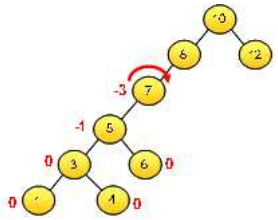


Figure 8.11: The tree is not an AVL tree because the balance factor of the node 7 is $Bf = -3$.

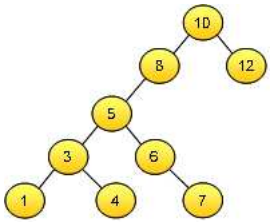


Figure 8.12: A RR rotation is applied on the tree in order to balance the node 7.

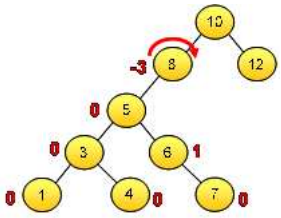


Figure 8.13: The tree is not an AVL tree because the balance factor of the node 8 is Bf = -3.

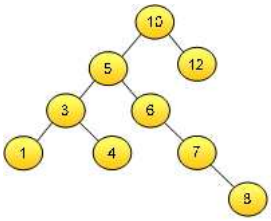


Figure 8.14: The tree is not an AVL tree because the balance factor of the node 8 is Bf = -3.

The tree shown in Figure 8.15 is not an AVL tree because the balance factor of the node 6 is Bf = 2 and, therefore, it must be balanced. A LL Rotation is

applied on the node 6, (a=6, b=7, c=8) and the result of the new tree is shown in Figure 8.16. The previous tree is not an AVL tree because the balance factor of the node 10 is Bf = -2 (see Figure 8.17), this means that the tree must be balanced. We can apply a RR Rotation on the node 10, (a=10, b=5, c=3) the result of the new tree is an AVL tree (see Figure 8.18).

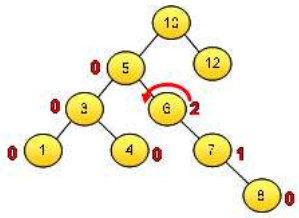


Figure 8.15: The previous tree is not an AVL tree because the balance factor of the node 6 is Bf = 2.

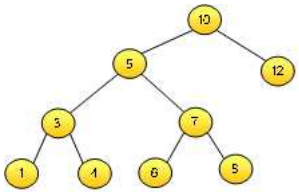


Figure 8.16: LL Rotation is applied on the node 6.

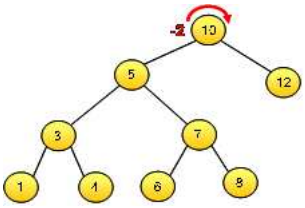


Figure 8.17: The previous tree is not an AVL tree because the balance factor of the node 10 is Bf = - 2.

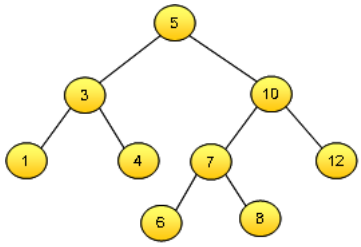


Figure 8.18: AVL Tree achieved by applying a RR rotation on the node 10.