5-2017

A Well-Designed, Tree-Based, Generic Map Component to Challenge the Progress towards Automated Verification

Nicodemus M.J. Mbwambo
Clemson University, nmbwamb@g.clemson.edu

A WELL-DESIGNED, TREE-BASED, GENERIC MAP COMPONENT TO CHALLENGE THE PROGRESS TOWARDS AUTOMATED VERIFICATION

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

---

by
Nicodemus M. J Mbwambo
May 2017

Accepted by:
Dr. Murali Sitaraman, Committee Chair
Dr. Brian Dean
Dr. Feng Luo

ABSTRACT

This thesis presents a non-trivial candidate software component assembly that presents an opportunity and a challenge to the progress towards automated verification. It presents an opportunity because the data abstraction implementation can serve as a proof of concept of the idea that well-designed and well-annotated software components with mathematical specifications and well-engineered implementation(s) lead to generated verification conditions (VCs) of correctness that are "obvious" to prove. It presents a challenge because verification of the implementation involves multiple theories and the use of a tree concept that is based on a general tree theory for which there are no special-purpose solvers.

The thesis contains a specification for a conceptualization of a tree with a position that makes it easy to explore and navigate a tree even as it avoids any explicit references to simplify reasoning. The thesis also contains concept enhancements for trees and an implementation layered using trees for a data abstraction for searching (a version of maps). A key contribution is the development of the implementation so that it is amenable for verification with internal assertions such as representation invariants and abstraction relations, operation specifications, loop invariants, and progress metrics, all of which involve the general tree theory.

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to Dr. Murali Sitaraman for his advice and guidance throughout the research and writing of this thesis. My appreciations also goes to the rest of my committee members, Dr. Brian Dean and Dr. Feng Luo for their comments and encouragement.

I would like to thank other members of Clemson and The Ohio State University RESOLVE Software Research Groups, in particular, Dr. Joan Krone, Dr. Bill Ogden, Mathew Pfister and Yu-Shan Sun. The general tree theory and the concepts (that have been refined and) used in this thesis are due to Dr. Joan Krone and Dr. Bill Ogden, who also provided me with feedback on various points of this thesis.

I would like to acknowledge Arusha Technical College (ATC) for providing me a chance to pursue my Master's degree at Clemson University.

And finally, I would like to thank my wife Josephine and my daughter Caitlyn for their encouragement and never ending support for the time I was away.

This research has benefitted in part by a US FFSP and a US NSF grant CCF-1161916.

TABLE OF CONTENTS

Table of Contents (Continued)

LIST OF FIGURES

List of Figures (Continued)

Figure Page

List of Figures (Continued)

List of Figures (Continued)

Figure Page

CHAPTER ONE

INTRODUCTION

Automated Verification

Automation of verification is the fundamental goal of many verification systems in existence today [8]. Among them are, Dafny [11], KeY [2] and, RESOLVE [14]. When automation in verification is ultimately achieved, the only support that programmers need to provide towards verification are the internal assertions such as progress metrics, loop invariants, and other mathematical specifications which describe precisely what the code is required to do. Among many components constituting a verifying compiler, the prover is a key one. The prover has a vital function of discharging verification conditions (VCs) proving which is equivalent to the correctness of a program. For practical reasons and to ensure the correctness of the prover itself, it is important that the prover to be as simple as possible and the VCs supplied to the prover as “obvious” as possible.

Significant progress has been made in the area of decision procedures for different theories and fragments, and these specialized decision procedures have proven to show much promise for discharging VCs that arise in the process of reasoning about programs [3, 12, 16]. However, a major consideration for the decision procedures is that they are effective only when the VC’s are within the scope of the respective decision procedures, most of which restrict the assertions to be of first order. However, to achieve automated verification in general, the challenge is to meet the task of proving VC’s that span multiple theories often involving the use of higher order logic, situations for which it is unlikely that viable decision procedures exist.

While the complexity arising from multiple theories including new ones is unavoidable, automated verification has any hope of becoming viable only if a software component specifications and corresponding implementations are well engineered and the VC's arising from establishing their correctness are "obvious". Being "obvious" implies the correctness of the resulting VCs can be established automatically in a few steps mechanically, without requirement of deep thinking [9]. Given suitable mathematical results and "obvious" VCs, verification can be done through simple deductions done even by humans and automated provers can establish correctness formally through the discovery of a short proof even without the use of special-purpose solvers.

With that being said, how hopeful can we be regarding automated provers? The answer to this question is put forward in the experimentation with two provers, Minimalist Prover (MP) and Z3 done in [4, 7, 15]. A detailed technical description of these provers is out of the scope of this thesis; however, in summary MP design focuses on showing validity of VCs provided a set of previously proven theorems in reusable mathematical units. With well-engineered theories, it is sufficient for this prover to use only instances of reusable mathematical units to construct proofs under the assumption that the assertions lead to VCs that are obvious regardless of how complex the theories are. In their experimentation, Kabani *et al* employed theories describing mathematical strings and numbers. These theories were further used in component specifications with no use of any decision procedures to tackle them. With this approach, as far as the provers are concerned, these created theories imitate the complex theories and so no special solvers are available. The experimentation is continuing with promising results and a suggestion of further exploration of the idea that will lead to automated verification of components specified using new theories.

Thesis Focus and Contribution

The non-trivial General Tree Theory used in this thesis was initially developed by Dr. Bill Ogden, and it contains an additional dimension of complexity compared to most of the theories since it does not already appear standardized in the world of mathematics. Sections of this theory used in this thesis are as shown on Appendix E. If a theory is well engineered, then the specifications and implementations based on that theory can lead to VCs that are relatively “obvious” for verification. While any verification system can be used, this thesis presents a candidate implementation in RESOLVE that can serve as a proof of concept for experimenting with the Minimalist Prover (MP) [4, 7, 15] which is built with an intent of verifying well-engineered programs accompanied with well-designed supporting mathematical units even when the generated VC’s span theories where no suitable decision procedures available.

The central contribution of this thesis is development of a verification-amenable implementation of a concept named Almost_Constant_Function_Template which specifies a map data abstraction. The implementation uses Exploration_Tree_Template, a concept that captures a navigable tree structure while avoiding any explicit reference behavior and need for aliasing. Development of the balanced, binary search tree implementation of the map data abstraction involves specification and implementation of several local operations, along with a host of internal assertions for verification, as detailed in this thesis. The thesis builds on and refines earlier, incomplete versions of the concepts for Exploration_Tree_Template and Almost_Constant_Function_Template conceived by Dr. Joan Krone and Dr. Bill Ogden. An important contribution of this thesis is explanations of these non-trivial concepts with illustrations so that they are accessible to the larger computer science audience. In addition to concept refinements two binary tree extensions for General Tree Theory were added, one to define balancing and another for binary search tree property. These two extensions are shown in Appendix F and Appendix G. Further, enhancements for exploration tree template have been developed and used.

An overview of the artifacts relevant to this thesis are shown in Figure 1. The figure includes additional elements, such as a list-based implementation maps to give a broader overview. The concepts, and theories refined and extended to achieve the development of the balanced, binary search tree implementation of the map data abstraction are the focus of this thesis and they are highlighted. In the coming chapters, these artifacts will be explained in detail.

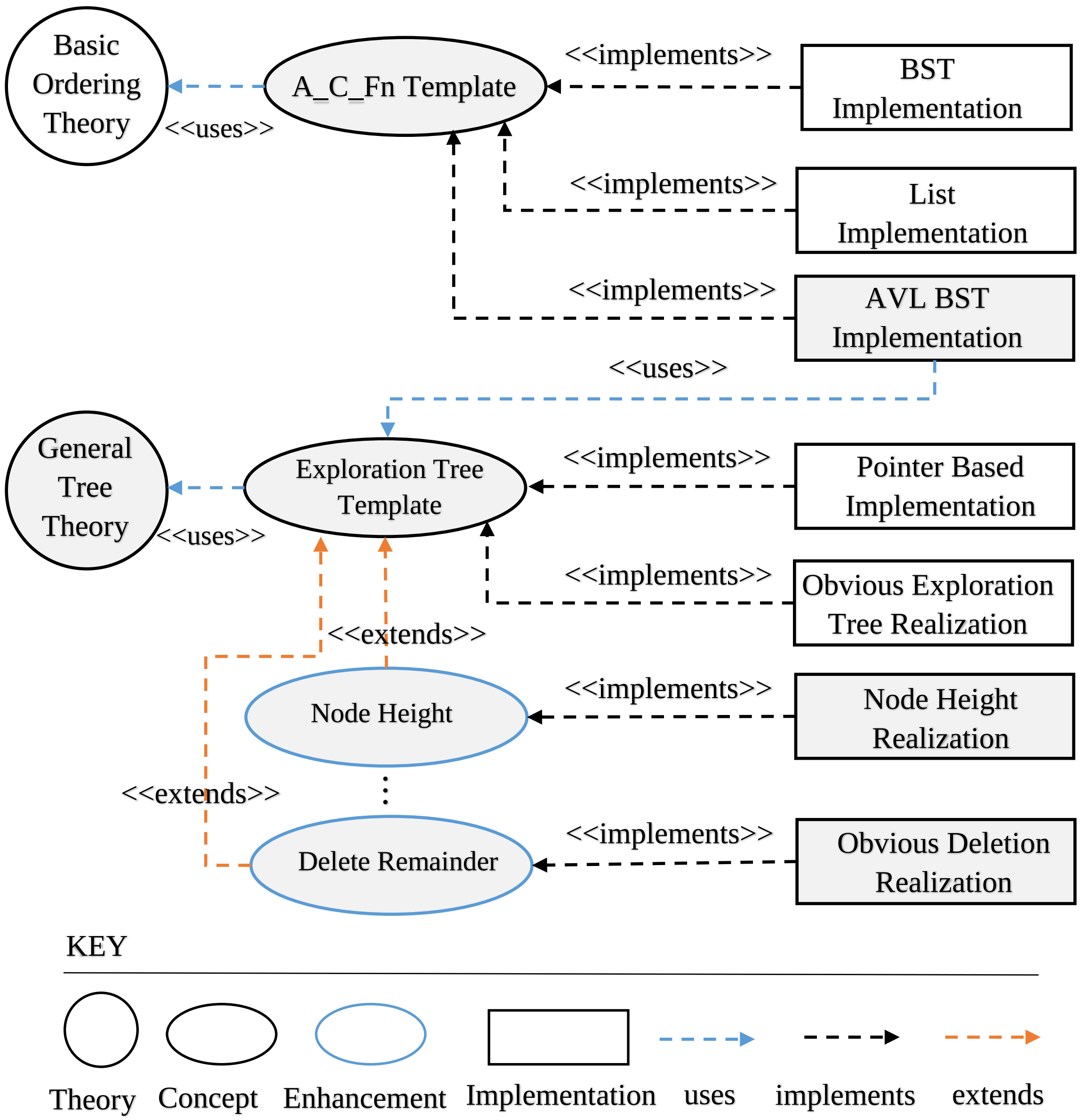


Figure 1: A General Overview of Thesis Focus and Contribution

Organization

The organization of this thesis is in four sections. The first and second sections after the introduction provide detailed explanations on the refined Exploration_Tree_Template, followed by different enhancements of this concept. Next is a discussion of Almost_Constant_Function_Template with a tree-based implementation. The third section is a discussion of verification of a simple enhancement implementation for purposes of illustration. The last section contains a summary and future directions.

CHAPTER TWO

EXPLORATION TREE TEMPLATE AND ENHANCEMENTS

To fulfill the challenge of providing a proof of concept that automated proof of correctness of a complex piece of software based on higher order logic is possible, it is necessary to choose a concept which is based on a non-standard mathematical theory which has been developed with automated proving in mind. For this purpose, the Exploration_Tree_Template is ideal.

The Exploration_Tree_Template is specified with no explicit reference behavior in contrast to how trees are presented in theory and practice in the literature [12]. Since the specification completely hides the underlying pointer-based tree structure, it simplifies reasoning of implementations which are based on these trees.

In this chapter the Exploration_Tree_Template concept is described precisely. To simplify the explanations, special diagrams are used to illustrate different aspects of the template and for brevity, figures used in support of the concept explanations will only show some snippets of the template. A detailed version of the entire template can be found in Appendix A. This chapter also includes some enhancements which contain extensions to the core concept.

An Informal Introduction to Exploration Tree Template

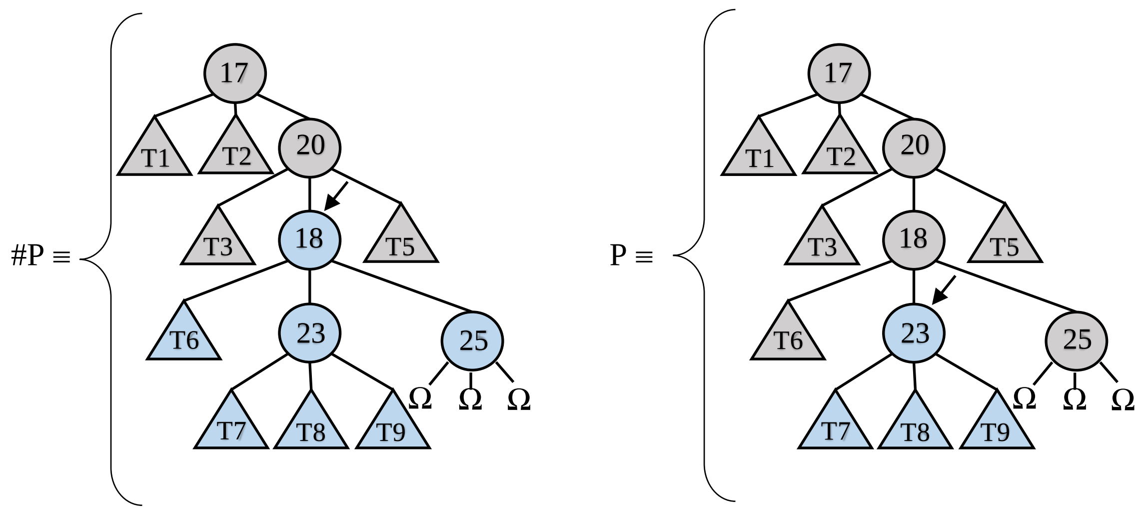
A skeleton of the formal specification for Exploration_Tree_Template is shown in Figure 2. This template is a generic concept (specification) with three parameters that are provided during instantiation. The first parameter required is a node label `(Node_label)` which specifies the node type; the second one `k` is an integer value setting the maximum number of children each node can have in a defined tree, and third is `Initial_Capacity` which state the maximum number of nodes that an instantiated tree can have.

An exploration tree is a tree with a position indicator. Figure 2 also shows that, Exploration_Tree_Template is a family of tree positions `(Tree_Posn)` emphasizing the fact that because of the generic nature of this template, not only is one type exported, but a whole family of types, each with different contents.

```
Concept Exploration_Tree_Template( type Node_Label; evaluates k,
                                    Initial_Capacity: Integer );
      uses Std_Integer_Fac, Std_Boolean_Fac, General_Tree_Theory

      Type Family Tree_Posn ⊆ U_Tr_Pos( k, Node_Label );

      Operation Advance( evaluates dir: Integer; updates P: Tree_Posn );

      Operation Reset( updates P: Tree_Posn );

      Operation At_an_End( restores P: Tree Posn ): Boolean;

      Operation Add_Leaf( alters Labl: Node_Label; updates P: Tree_Posn );

      Operation Remove_Leaf( replaces Leaf_Lab: Node_Label;
                                            updates P: Tree_Posn );

      Operation At_a_Leaf( restores P: Tree_Posn ): Boolean;

      Operation Swap_Label( updates Labl: Node_Label;
                                            updates P: Tree_Posn );

      Operation Swap_Rem_Trees( updates P, Q: Tree_Posn );

      Operation Swap_w_Rem( updates P, Q: Tree_Posn );

      Operation Retreat( updates P: Tree_Posn );

      Operation Path_Length( restores P: Tree_Posn ): Integer;

   end Exploration_Tree_Template;
```

Figure 2: A Skeleton Interface for Exploration Tree Template

The template includes several primary operations that are useful in creating, navigating and modifying trees as shown in Figure 2. The first operation Advance is used in navigation of trees; the movement can be in one of the `k` directions `(dir)` specified during operation call. Starting from one tree position operation Advance can navigate to the next tree position depending on the given direction. Advance modifies the tree position and hence, the use of the parameter mode **updates**. Figure 3(a) below shows a tree position indicated by an arrow known as the position indicator.



(a) **(b)**

Figure 3: (a) An example exploration tree: A tree with position indicator (b) Updated tree position after a call to Advance operation

The tree in Figure 3(a) has the value of `k` equals to `3` giving three possible directions for advancing this tree. For example, if from this current position, Advance operation is called on direction `2`. The position indicator will move into the tree and an updated tree position is shown in Figure 3(b). Retreat operation does the opposite of Advance, once Retreat is called, it updates the tree position by moving the indicator to the previous tree position. Using Figure 3(b), an operation Retreat on this tree position will result into a tree position in Figure 3(a). When a position indicator is advanced to the end of the tree as shown in Figure 4(a), we cannot advance the tree any further and the tree position is said to be at an end. A Boolean operation At_an_End can be used to test if a tree position is at an end, this operation does not make any changes to the tree position, therefore, parameter mode **restores** is used. Figure 4(a) is also an example of a position were an operation Add_Leaf can be called and an extra node will be added into the tree as shown in Figure 4(b). Operation Add_Leaf updates the tree position to include the new node whose label is passed in as parameter during operation call. Because we only need this label to create the new node and nothing after that, parameter mode **alters** is used for this case.

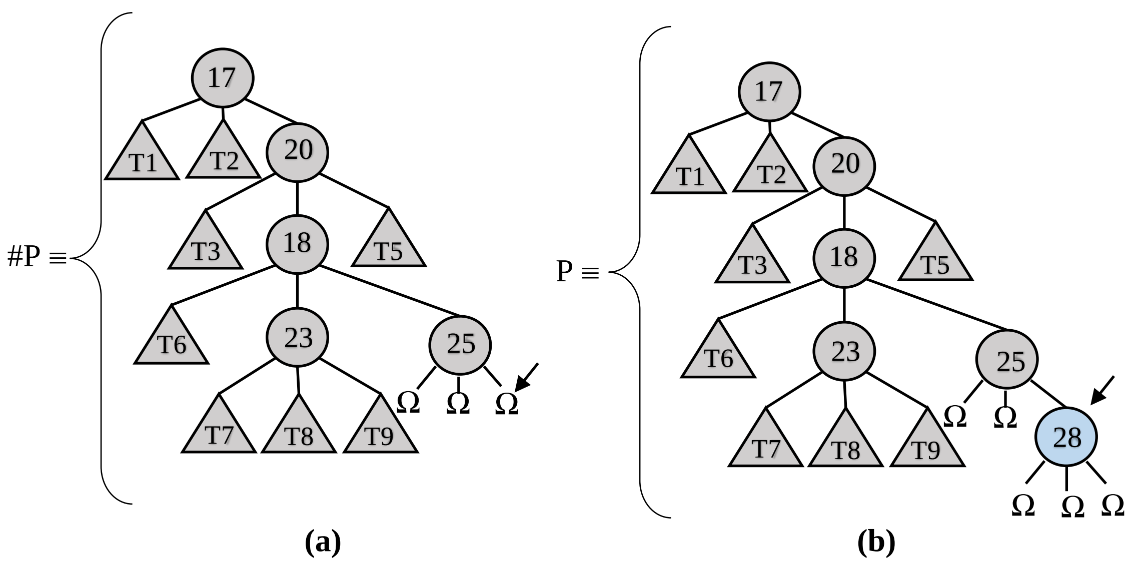


Figure 4: (a) Tree position indicator at an end (b) Updated Tree Position after adding a new leaf

An operation Reset will move the position indicator to the beginning (root node) of the tree from any tree position, this operation can be useful when we want to return a root node. At_a_Leaf is a Boolean operation and will return true when the tree position is at a leaf, at this position the pointer will be at any of the nodes with empty tree children represented by Ω. Figure 4(b) is an example of the tree position being at a leaf.

The fact that Exploration_Tree_Template is a generic concept, its parameters can be of any type and in such case for a reasonable and efficient transfer of these arbitrary entries, swapping is used over copying of reference or values [5]. The efficiency in swapping is in the execution-time where compilers takes constant time exchanging references to even large objects, this implementation of swapping is different from copying where for large objects execution-time needs to account time for copying the objects. Swapping also allows reasoning without introducing aliasing, in contrary to copying which introduces aliasing and so compromising abstract reasoning.

Because of these advantages of swapping in generic components, Swap_Label operation is defined in Exploration_Tree_Template, this operation will be used to transfer arbitrary type label into the tree. The two-way transfer provided by swapping will update both the tree position and the parameter node label. To illustrate this operation, consider Figure 5(a) which shows a tree position and a node label, a call to Swap_Label will update both label and a tree position and the result is shown in Figure 5(b).

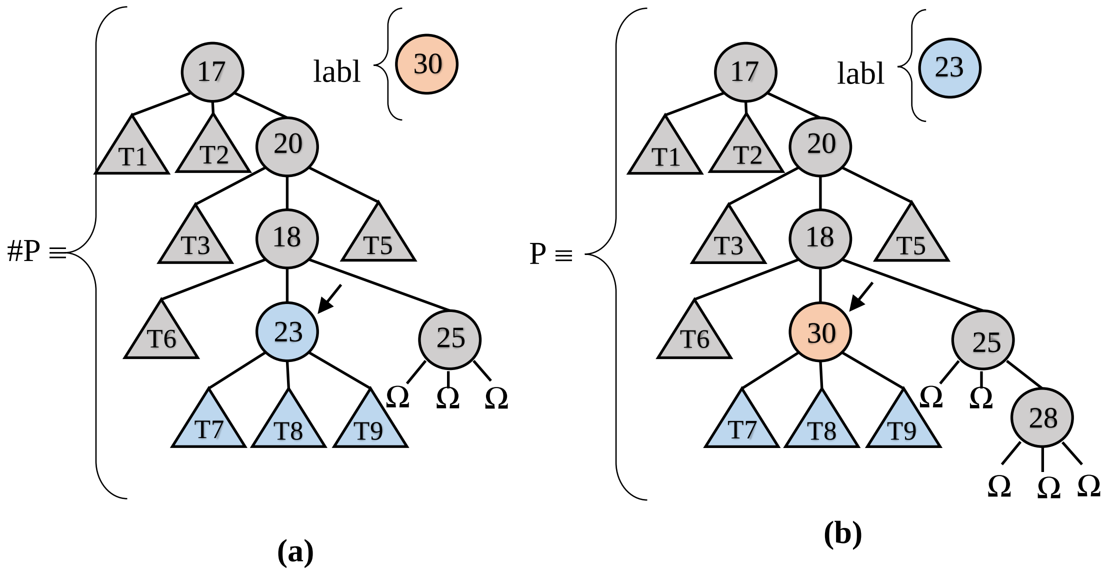


Figure 5: (a)Given tree position and node label (b) Updated tree position and node label after a call to Swap Label

At any tree position, the position indicator divides the tree into two parts, the part before the indicator which is a "Path" and the part after the indicator which is the "Remaining Tree" (Rem_Tr) both Path and Rem_Tr will be formally explained in the next section.

Exploration_Tree_Template can be implemented in a straightforward fashion using classical `k`-link nodes. To be verified formally, it can be implemented using an abstraction of linked locations [10]. All tree operations can be implemented to work in constant time.

A Formal Presentation of Exploration Tree Template

This section explains a formal specification of the Exploration_Tree_Template shown in Figure 6. The specification of this concept uses two facilities Std_Boolean_Fac and Std_Integer_Fac (which bring in Booleans and Integers, since no types are assumed prebuilt in RESOLVE), as well as the General_Tree_Theory. Next is a concept level **requires** clause which state that the value `k` must be greater than or equal to `1` and `Initial_Capacity` is at least `1`, these two requirements will guarantee no tree is created with zero children and zero capacity. A global conceptual variable `Remaining_Cap` is a natural number and get initialized to `Initial_Capacity` in the initialization **ensures** clause, `Initial_Capacity` is provided during instantiation of the template. When nodes are added to the tree, or removed from the tree, `Remaining_Cap` is affected.

The mathematical model for Exploration_Tree_Template is a family of tree positions `(Tree_Posn)`. This family of types is modeled as a subset of all Uniform Tree Positions `(U_Tr_Pos)` defined by `k` children and `Node_Label`. As discussed earlier a `Tree_Posn` has two parts, a "Path" `(Path)` which is a string of "Sites" and a remaining tree `(Rem_Tr)` which is a *k*-tree. The mathematical model is illustrated and explained using example in the upcoming paragraphs.

In Figure 6, Exploration_Tree_Template uses `P` as an exemplar to specify the effects of **initialization** (constructor) and **finalization** (destructor). The effect of initialization is that `P.Path` is `Empty_String(Λ)` and `P.Rem_Tr` is `Empty_Tree(Ω)`.

The effect of **finalization** is that the count of the tree nodes that belonged to the tree object is added back to the existing `Remaining_Cap`.

The following figures will illustrate what is meant by “Site”, `Path` and the `Rem_Tr`. As explained earlier a `Path` is a string of Sites and in every single Site there is a Label, Left Tree String (LTS) and Right Tree String (RTS), LTS and RTS are sometimes called Left Branch String and Right Branch String respectively. To illustrate this, we use Figure 7 which introduces another presentation of a `Tree_Posn` and this time with a detailed breakdown. This is an abstract way of showing a `Path` and `Rem_Tr` of the `Tree_Posn`, and it corresponds directly to the mathematical model in the concept shown in Figure 6. Figure 7 has two Sites, the first Site has a node label 17, a LTS which has two Trees ($\langle T_1, T_2 \rangle$) and an empty RTS. The second Site has a label of 20, one tree in the LTS ($\langle T_3 \rangle$) and another Tree in RTS ($\langle T_5 \rangle$).

```
Concept Exploration_Tree_Template( type Node_Label; evaluates k,
                                        Initial_Capacity: Integer );
      uses Std_Integer_Fac, Std_Boolean_Fac, General_Tree_Theory
                                             with Relativization_Ext;
      requires 1 ≤ k and 0 < Initial_Capacity which_entails k: ℕ>0
                                             and Initial_Capacity: ℕ;
      Var Remaining_Cap: ℕ;
         initialization
            ensures Remaining_Cap = Initial_Capacity;

      Family Tree_Posn ⊆ U_Tr_Pos( k, Node_Label );
         exemplar P;
         initialization
            ensures P.Path = Λ and P.Rem_Tr = Ω;
         finalization
            ensures Remaining_Cap = #Remaining_Cap + N_C (P.Path Ψ
                                                            P.Rem_Tr);

            ⋮
   end Exploration_Tree_Template;
```

Figure 6: A formal specification of Exploration Tree Template

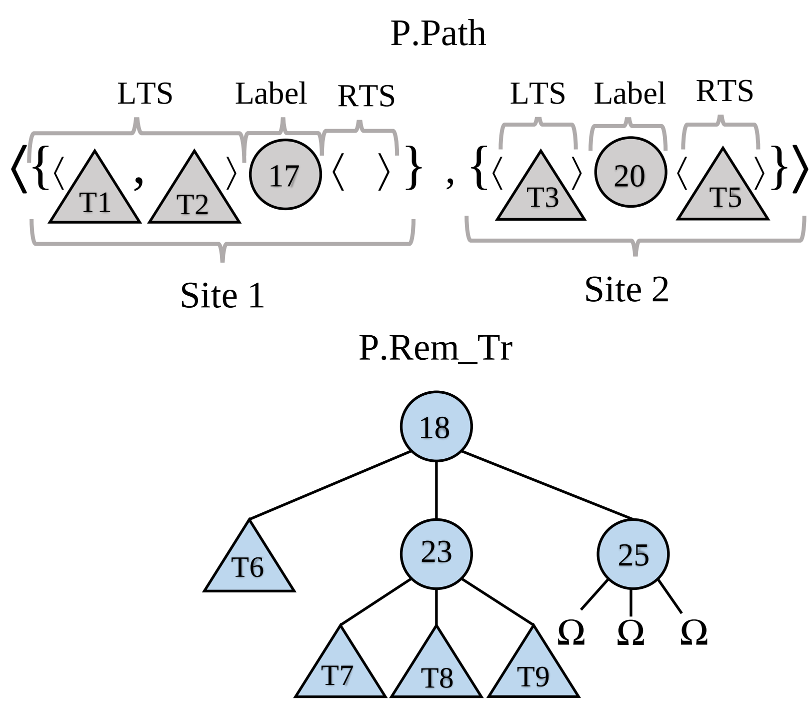


Figure 7: A formalized version of a Tree Position

From a Path and a Rem_Tr of a given Tree_Posn we can form the entire tree. To achieve this, Zip operator (Ψ) defined in the General Tree Theory is used. This operator is used in the specifications of several operations, so we begin with an explanation of this operator. From a Tree_Posn Zip operator takes Sites in the Path and stitch them back to the tree in the Rem_Tr resulting to a tree whose root node will be the label of the last site extracted from the Path. To illustrate this operator, consider a Tree_Posn in Figure 8(a) which has two Sites in the Path and the remaining tree. Zip operator is inductively defined to extract the last added Site first and zip it to the remaining tree leaving one Site in the Path and a resulting tree is shown in Figure 8(b). Next the last Site will be extracted and zipped to the remaining resulting to a whole tree and leaving the Path empty, the result is shown in Figure 8(c) with the root node being the label of the last site.

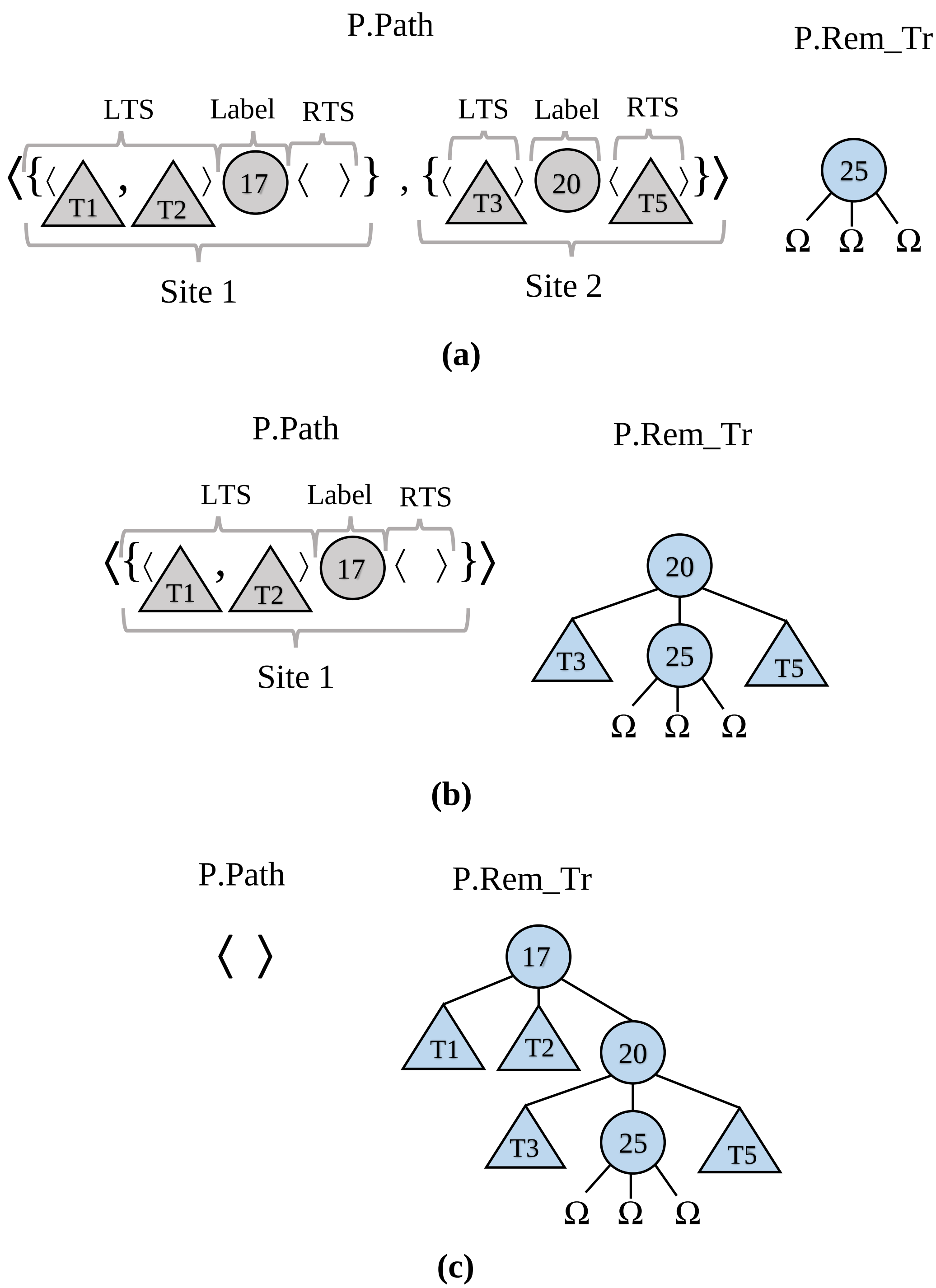


Figure 8: An illustration of Zip Operator

As discussed earlier, Exploration_Tree_Template includes specifications for several primary operations that are useful in creating, navigating and modifying trees. Now a formal version of these operations will be explained. The first primary operation Advance is specified in Figure 9.

```
Operation Advance (evaluates dir: Integer; updates P: Tree_Posn);
    requires P.Rem_Tr ≠ Ω
    which_entails P.Rem_Tr: Tr(Node_Label)~{Ω}and 1 ≤ dir ≤ k;
    ensures P.Rem_Tr = $̸( Prt_btwn(dir ∸ 1, dir,
                                         Rt_Brhs(#P.Rem_Tr)) ) and
    P.Path = #P.Path∘⟨( Rt_Lab(#P.Rem_Tr), Prt_btwn(0,
                                  dir ∸ 1,Rt_Brhs(#P.Rem_Tr)),
                              Prt_btwn(dir, k, Rt_Brhs(#P.Rem_Tr)) )⟩;
```

Figure 9: A formal Specification of Advance operation

Advance operation updates an incoming `Tree_Posn` on a given `dir` if the `Rem_Tr` is not an `Empty_Tree(Ω)` and the given `dir` is a valid value of `k (i.e. 1 ≤ dir ≤ k)`. The subordinate annotation **which_entails** is included in this specification following the requirement that `P.Rem_Tr` is not empty tree to explicitly alert the type checker that it is acceptable to use the incoming value of `P.Rem_Tr`, where a non-empty tree is expected. This annotation is the reason `#P.Rem_Tr` can be used in Rt_Lab and Rt_Brhs in the **ensures** clause without violating type checking. If these requirements specified in the **requires** clause are met, then the **ensures** clause of Advance operation states how `Path` and `Rem_Tr` of a given `Tree_Posn` are updated.

Operation Advance is further described using Figure 10 and Figure 11. Figure 10 is a current `Tree_Posn` and the named positions from `0` to `3` are for the sake of simplifying the formal explanations this operation. If `dir = 3` on the parameter list and the tree positions shown in Figure 10. The post condition in Advance shows that the `Path` will be updated to contain all Sites it had before `(#P.Path)`, concatenated with a new Site defined by the root label of the remaining tree `(Rt_Lab(#P.Rem_Tr))` and the two branches separated by the provided direction `(dir)`. The Left Tree Branch will start from position `0` to `dir ∸ 1 (2)`, where ∸ is natural number subtraction. The Right Tree Branch is between `dir` which is `3` and `k` which is also `3`, explaining why the Right Branch String is the empty string. The `Rem_Tr` will be updated as depicted in Figure 11.

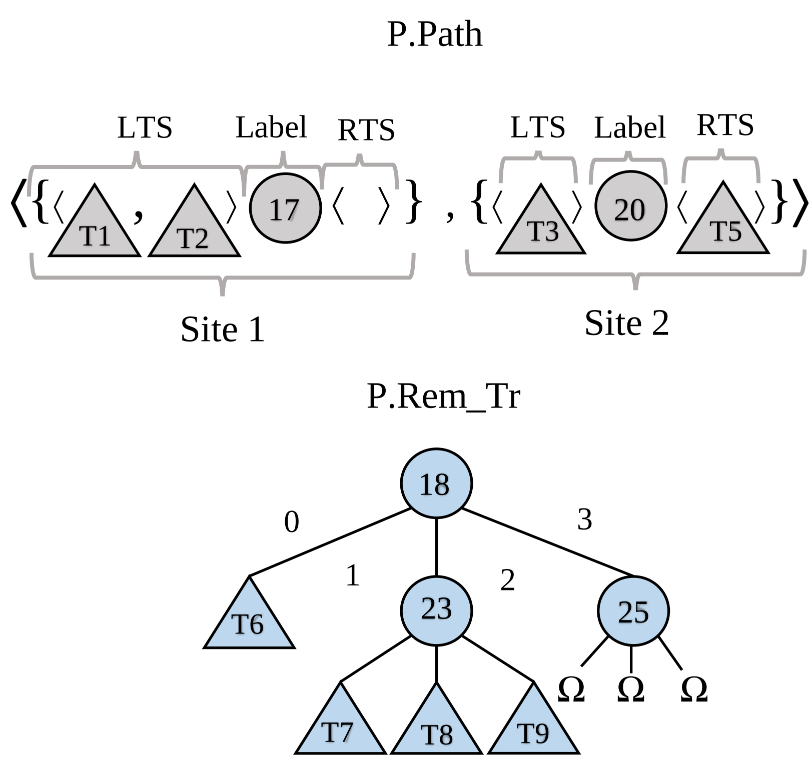


Figure 10: Current Tree Position before calling Advance

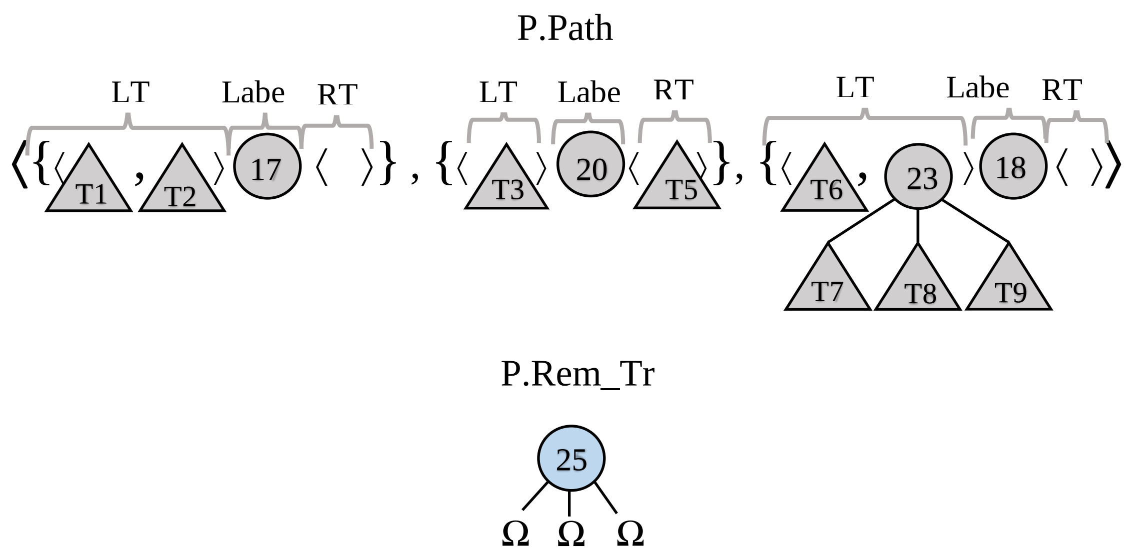
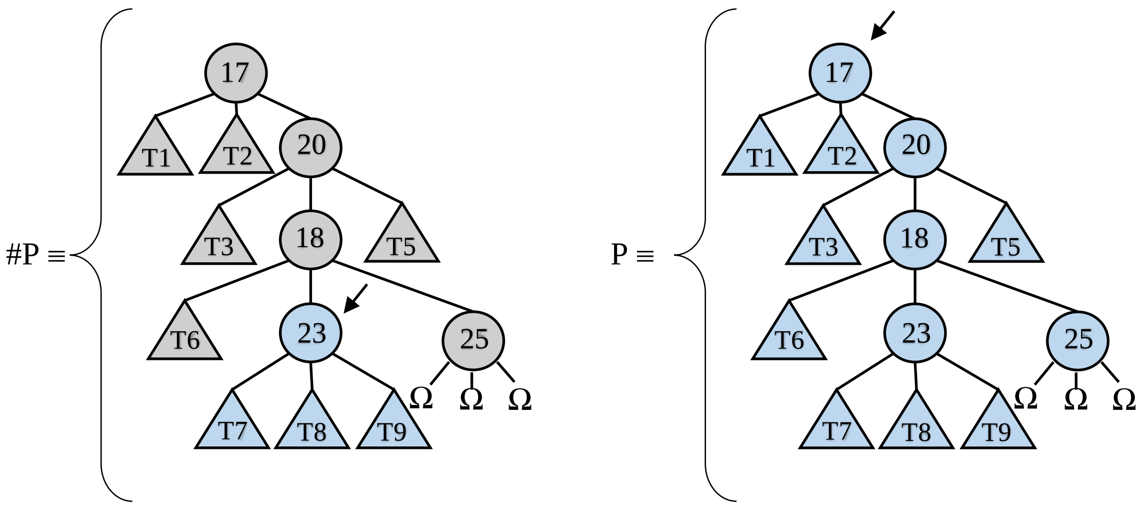


Figure 11: Tree Position after Advancing on direction 3

As it can be observed from Figure 10 and Figure 11, one call to Advance added one site to the existing `Path`. In contrast, the operation Retreat will extract the last added Site and zip it with the `Rem_Tr`. Operation Retreat will be explained in detail later in the chapter. The Reset operation, specified in Figure 12, has an effect of moving the tree position to the top. Reset updates the current `Tree_Posn` by ensuring the `Path` becomes an `Empty_String(Λ)` and the `Rem_Tr` to be the result of zipping together an incoming `Path (#P.Path)` with the incoming `Rem_Tr(#Rem_Tr)`, there is no **requires** clause for this operation. To illustrate Reset operation Figure 13(a) shows a current `Tree_Position` using a position indicator, Figure 13(b) is the result of calling Reset operation, the position indicator will be at the root node where the `Path` is now `Empty_String(Λ)` and the `Rem_Tr` is an entire tree.

```
   ⋮
  Operation Reset( updates P: Tree_Posn );
     ensures P.Path = Λ and P.Rem_Tr = #P.Path Ψ #P.Rem_Tr;

  Operation At_an_End( restores P: Tree Posn ): Boolean;
     ensures At_an_End = ( P.Rem_Tr = Ω )
   ⋮
   ⋮
end Exploration_Tree_Template;
```

Figure 12: Specifications for operations Reset and At an End



(a) **(b)**

Figure 13: (a) Current Tree Position (b) Tree Position indicator at the root after calling Reset

The next operation At_an_End specified in Figure 12 is a Boolean operation which returns true in case a `Tree_Posn` is at the end. A `Tree_Posn` is said to be at an end if and only if the `Rem_Tr` is an `Empty_Tree(Ω)`.

From an Empty_Tree(Ω) one can create a tree by adding one node at a time, to achieve this, operation Add_Leaf is specified in Figure 15, the operation will have an effect of adding a new leaf and decreasing the value of the Remaining_Cap by one whenever it is called. The Remaining_Cap will be zero (0) when we have no room to add any more nodes. Add_Leaf can only be called when the Remaining_Cap is greater than zero, and the Rem_Tr is an Empty_Tree(Ω) as stated in the **requires** clause. At the end of the operation, Add_Leaf has no effect to the current Path and thus, P.Path = #P.Path and the Rem_Tr will be a result of joining (Using Join operator, Jn) a new leaf of an incoming Label with k branches of Empty_Tree(Ω) as stated in the **ensures** clause.

Join operator (Jn) is defined in the General Tree Theory and take in a string of trees and a node label to give back a complete tree. The node label becomes the root node of the resulting tree and each individual tree within the string becomes a child to this root node. Figure 14 illustrate how Jn operator works using a string of trees in Figure 14(a), these trees have the same properties, in this example just empty trees are used. Figure 14(b) is a node label. Join operator will connect all these trees to the node label and form a tree in Figure 14(c) which has the same properties as the individual trees before the join.

A formal illustration of Add_Leaf is shown in Figure 16, in Figure 16(b) is a tree position with a new node added to the remaining tree.

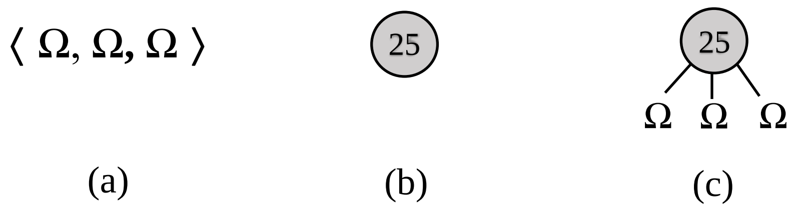


Figure 14: An illustration of a Join operator

```
Operation Add_Leaf( alters Labl: Node_Label; updates P: Tree_Posn );
    affects Remaining_Cap;
    requires P.Rem_Tr = Ω and Remaining_Cap > 0;
    ensures P.Path = #P.Path and
                    P.Rem_Tr = Jn( ⟨Ω⟩^k, #Labl ) and
                            Remaining_Cap = #Remaining_Cap ∸ 1;
```

Figure 15: Specification of Add Leaf Operation

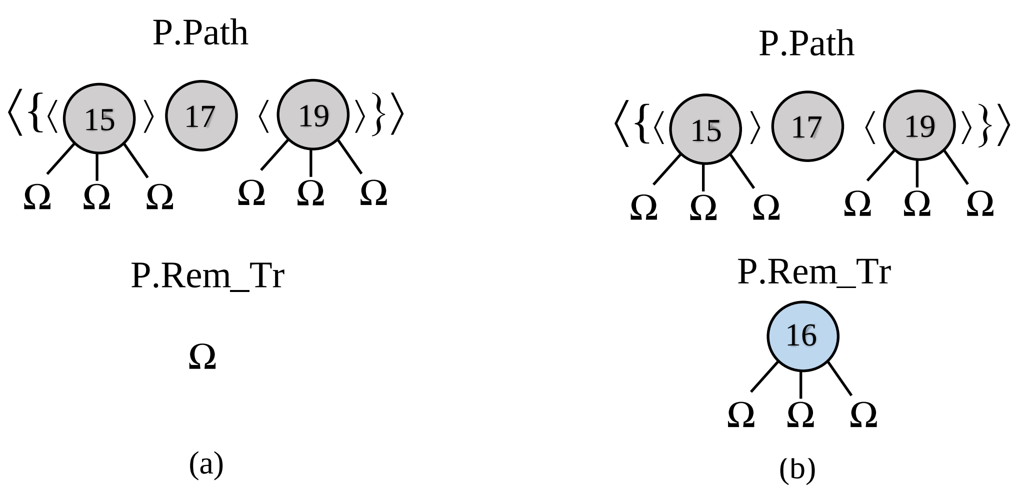


Figure 16: (a) Current Tree Position (b) Updated Tree Position on calling Add Leaf

Operation Remove_Leaf does the opposite of Add_Leaf. This operation will update the given `Tree_Posn` to having a `Rem_Tr` equal to `Empty_Tree(Ω)` and the root label of the removed leaf updates the value of `Leaf_Lab`. The specifications for Remove_Leaf are shown in Figure 17.

```
 ⋮
Operation Remove_Leaf( replace Leaf_Lab: Node_Label;
                                          updates P:Tree_Posn );
      affects Remaining_Cap;
      requires P.Rem_Tr ≠ Ω
      (which_entails P.Rem_Tr: Tr(Node_Label)~{Ω}) and
                                    Rt_Brhs(P.Rem_Tr) = ⟨Ω⟩^k;
      ensures P.Path = #P.Path and P.Rem_Tr = Ω and
                  Leaf_Lab = Rt_Lab(#P.Rem_Tr) and
                        Remaining_Cap = #Remaining_Cap + 1;

Operation At_a_Leaf( restores P: Tree_Posn ): Boolean;
      ensures At_a_Leaf = (P.Rem_Tr ≠ Ω
            (which_entails P.Rem_Tr: Tr(Node_Label)~{Ω}) and
                                    Rt_Brhs(#P.Rem_Tr)=⟨Ω⟩ ^k);
 ⋮
 ⋮
end Exploration_Tree_Template;
```

Figure 17: Specifications for Operation Remove Leaf and At a Leaf

At_a_Leaf is a Boolean operation with specifications shown in Figure 17, the operation return a Boolean value depending on whether a given `Tree_Posn` has a leaf as the `Rem_Tr` or not.

When a specific node label needs to be updated within a given `Tree_Posn`, Exploration_Tree_Template specifies the Swap_Label operation as shown in Figure 18, in the previous section a reason why swapping is used instead of copying was explained. Swap_Label requires `Rem_Tr` not to be an `Empty_Tree(Ω)`, this is stated in the **requires** clause. The **ensures** clause updates both label `(Labl)` and `Tree_Posn`, the outgoing `Labl` will equal the root label `(Rt_Lab)` of the incoming `Rem_Tr` and a new root label will be a join of all branches of the incoming `Rem_Tr` to `(#Labl)`.

```
    ⋮
Operation Swap_Label( updates Labl: Node_Label; updates P: Tree_Posn );
    requires P.Rem_Tr ≠ Ω
                        (which_entails P.Rem_Tr: Tr(Node_Label)~{Ω});
    ensures Labl = Rt_Lab(#P.Rem_Tr) and P.Path = #P.Path and
                        P.Rem_Tr = Jn( Rt_Brhs(#P.Rem_Tr), #Labl);

Operation Swap_Rem_Trees( updates P, Q: Tree_Posn );
    ensures P.Path = #P.Path and Q.Path = #Q.Path and
                                    P.Rem_Tr = #Q.Rem_Tr and
                                          Q.Rem_Tr = #P.Rem_Tr;
Operation Swap_w_Rem( updates P, Q: Tree_Posn );
    ensures P.Path = Λ and P.Rem_Tr = #Q.Rem_Tr and
                                Q.Path = #Q.Path∘#P.Path and
                                         Q.Rem_Tr = #P.Rem_Tr;
Operation Retreat( updates P: Tree_Posn );
    requires P.Path ≠ Λ;
    ensures P.Path = Prt_btwn(0, |#P.Path| ∸ 1, #P.Path) and
    P.Rem_Tr =(Prt_Btwn (|#P.Path| ∸ 1, |#P.Path|, #P.Path)
                                                   Ψ P.Rem_Tr;

Operation Path_Length( restores P: Tree_Posn ): Integer;
    ensures Path_Length = |P.Path|;

Operation Rmng_Capacity(): Integer;
    ensures Rmng_Capacity = ( Remaining_Cap );

end Exploration_Tree_Template;
```

Figure 18: The rest of Operations in Exploration Tree Template

The operations Swap_Rem_Trees and Swap_w_Rem are two operations with very close effect, both operations takes in two known tree positions as parameters and swap their remaining trees. However, Swap_Rem_Trees will have no changes to the paths of both tree positions, while Swap_w_Rem will update both `Tree_Posn` and `Rem_Tr`. Figure 19 illustrate this using two colored tree positions `P` and `Q`, shown in Figure 19(a) are the tree positions before Swap_Rem_Trees is called. Figure 19(b) shows updated tree positions `P` and `Q`. Figure 20(b) illustrates the results of calling operation

Swap_w_Rem on tree positions in Figure 20(a). Notice in Figure 20 also `Path` is updated for both `P` and `Q`.

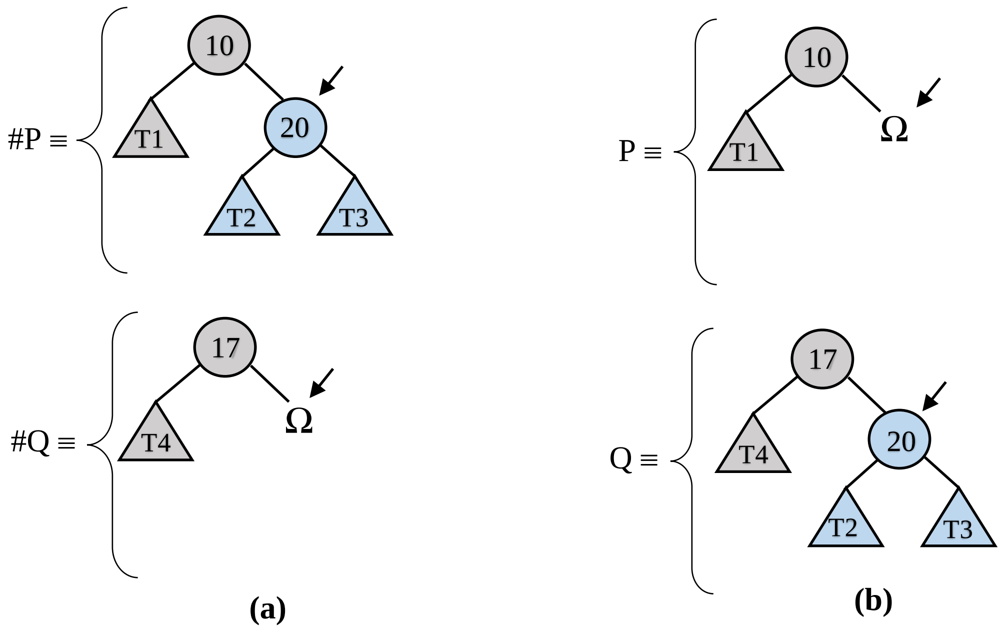


Figure 19: (a) Tree positions P and Q (b) Resulting tree positions P and Q after Swapping the Remaining Trees

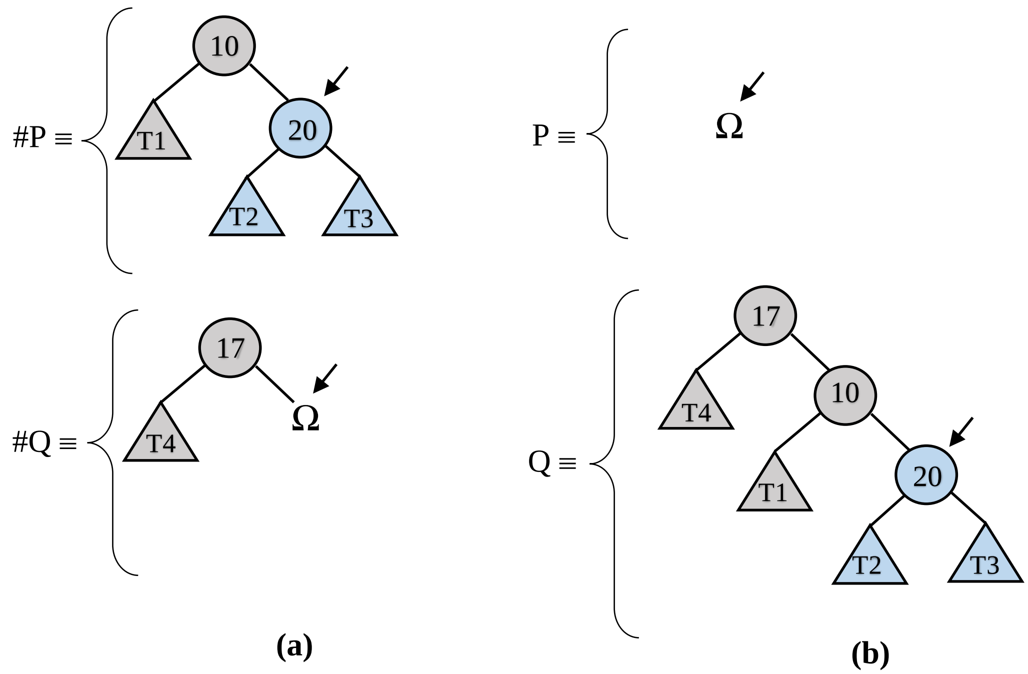


Figure 20 : (a) Tree positions P and Q (b) P and Q updated after Swap_w_Rem

As stated earlier operation Retreat has an opposite effect to Advance. Retreat will remove the last added Site and zip it to the `Rem_Tr` of the `Tree_Posn`. Retreat can only be called when the `Path` of a given `Tree_Posn` is not an `Empty_String (Λ)` as stated in the **requires** clause. The **ensures** clause uses `Prt_Btwn` which is a string operator to extract the last added Site that will be zipped to the `Rem_Tr`.

The last two operations to be specified are Path_Length and Rmng_Capacity. Path_Length operation returns the length of the `Path` and the Rmng_Capacity operation will return the `Remaining_Cap` of the tree when called.

Enhancements to Exploration_Tree_Template

In the discussion above, Exploration_Tree_Template was explained in detail and in it are several primary operations specified. But a close observation will reveal that there may be other operations that can be useful in variety of applications but not specified in this template. Generally, to make the specifications task and realization of data abstraction reasonable only a few primary operations, typically orthogonal and implementable efficiently, are usually specified in the concept. Any other operation that can be implemented using a combination of primary operations and may be useful can be specified as secondary operations. In RESOLVE language, a specification inheritance mechanism is provided to permit an easy extension of these primary operations available in the concept by writing enhancements to concepts.

The enhancements discussed in this subsection are used in the tree-based implementation of the map concept in the next chapter.

The first enhancement to be discussed is Deletion_Capability which describes a Delete_Rem_Tree operation with specifications is shown in Figure 21.

```
Enhancement Deletion_Capability for Exploration_Tree_Template;
   Operation Delete_Rem_Tree (updates P: Tree_Posn)
      affects Remaining_Cap;
      ensures P.Path = # P.Path and P.Rem_Tr = Ω and
   Remaining_Cap = #Remaining_Cap + N_C ( #P.Rem_Tr);
end Deletion_Capability;
```

Figure 21: Specification of Delete Remainder Operation

The operation specifications in Figure 21 guarantees what is in the `Path` before the operation is called remain the same even after the operation call (`P.Path =`

#P.Path) and updates the Rem_Tr to be an empty tree after deleting the remaining tree. Figure 22 demonstrates this using a tree position in (a). After calling Delete_Rem_Tree, everything in the remaining tree will be deleted. The resulting tree position is shown in Figure 22(b).

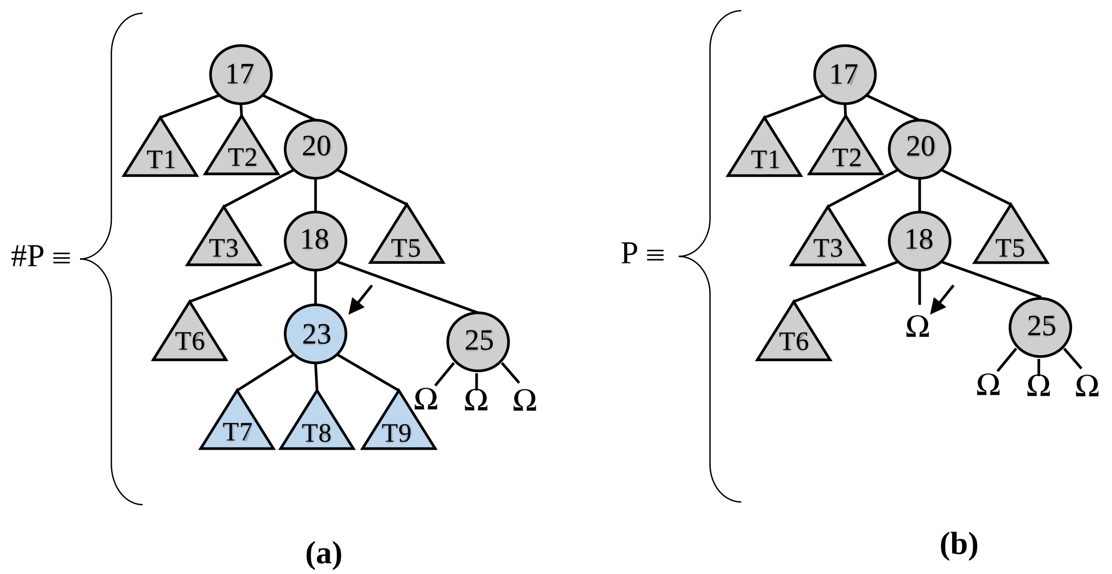


Figure 22: (a) Tree position before deleting the remaining tree (b) Tree position after deleting the remaining tree

```
Realization obvious_Deletion_Realiz for Deletion_Capability
                                  of Exploration_Tree_Template;
   Procedure Delete_Remainder (updates P: Tree_Posn);

      Var Q: Tree_Posn;
      Swap_Rem_Trees (P, Q);

   end Delete_Remainder;
end obvious_Deletion_Realiz;
```

Figure 23: An Implementation of Delete Remainder Operation

The second enhancement achieves a node count and returns the number of nodes in the remaining tree of a given `Tree_Posn`. The operation Rem_Tr_Node_Count, shown in Figure 24, counts the nodes in the remainder part of the `Tree_Posn`. The total number of nodes for the tree position can be found by making the entire tree a Rem_Tr.

```
Enhancement Rem_Tr_Node_Count_Capability for Exploration_Tree_Template;

   Operation Rem_Tr_Node_Count( restores P: Tree_Posn ):Integer;
      ensures Rem_Tr_Node_Count = ( N_C(P.Rem_Tr ));

end Rem_Tr_Node_Count_Capability;
```

Figure 24: Specification of Rem_Tr_Node_Count Operation

Rem_Tr_Node_Count is implemented in Figure 25. The basic idea of this realization is to recursively count nodes starting from a root node of the remainder tree and all its children. To show termination in the recursion and loop, two proper ordinal valued progress metric expressions are defined in the **decreasing** clause. These two metrics will decrease in every recursive call or iteration of the loop. The **maintaining** clause provided must be adequate for verification.

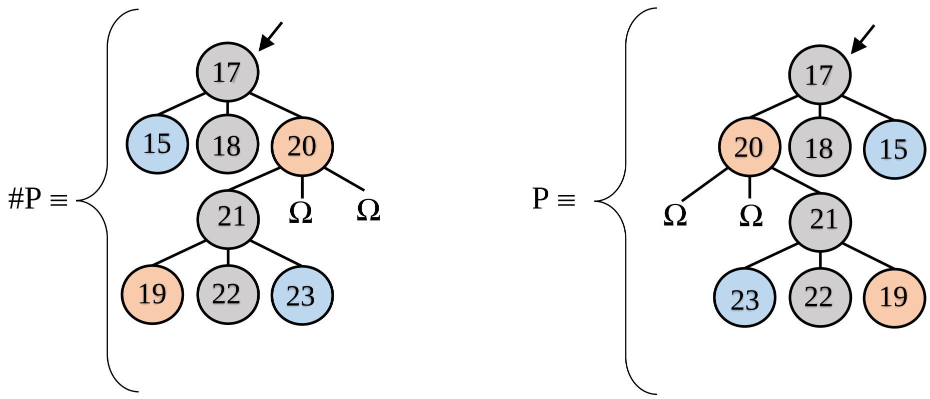
```
Realization Recursive_Node_Count_Realiz for
            Rem_Tr_Node_Code_Capability of Exploration_Tree_Template
   Recursive Procedure Rem_Tr_Node_Count (restores P:Tree_Posn):Integer

      decreasing ht(P.Rem_Tr);

      Var dir,count : Integer;
      If (At_an_End(P)) then
         Rem_Tr_Node_Count := 0 ;
      else
         dir := 1;
         count := 1;
         while (dir <= k)
            maintaining P.Path = #P.Path and P.Rem_Tr = #P.Rem_Tr
               N_C(P.Rem_Tr) = count +
                                          ∑_dir^k N_C(Split_at(dir∸1,P.Rem_Tr ) ;
            decreasing ((k+1) - dir);
         do
            Advance(dir, P);
            count := count + Rem_Tr_Node_Count(P);
            Retreat(P);
            Increment(dir);
         end;
         Rem_Tr_Node_Count := count;
      end;
   end Rem_Tr_Node_Count;
end Recursive_Node_Count_Realiz;
```

Figure 25: Rem_Tr_Node_Count Realization

The third enhancement is the Tree_Reversal_Capability specified in Figure 26. Reversal of a tree about a given root node will swap nodes from outer children going inwards. Figure 27 illustrates tree reversal. The implementation of this enhancement is shown in Figure 28.

```
Enhancement Tree_Reversal_Capability for Exploration_Tree_Template;
   Operation Reverse_Rem_Tr (updates P: Tree_Posn );
      ensures P.Rem_Tr = #P.Rem_Tr^TRev and P.Path = #P.Path;
end Tree_Reversal_Capability;
```

Figure 26: Enhancement specification for Tree_Reversal_Capability



(a) **(b)**

Figure 27: (a) A tree position before reversal (b) updated tree position after reversal

```
Realization Obvious_Reversal_Realiz for Tree_Reversal_Capability
                                    of Exploration_Tree_Template;
   Recursive Procedure Reverse_Rem_Tr (updates P: Tree_Posn);

      decreasing ht(P.Rem_Tr) ;

      Var Q: Tree_Posn;
      Var dir, last: Integer;
      dir := 1;
      last := k;
      If (not At_an_End(P)) then
         While (dir < last)
            maintaining P.Path = #P.Path and #P.Rem_Tr =
            Jn(( 〈Prt_Btwn(0,dir - 1,Rt_Brhs(P.Rem_Tr))o
                  (Prt_Btwn(dir - 1, last, Rt_Brhs(P.Rem_Tr))^Rev o
                  Prt_Btwn(last, k, Rt_Brhs(P.Rem_Tr))〉)^Rev,
                                                Rt_Lab(P.Rem_Tr));
            decreasing (last - dir)
         do
            Advance(dir, P);
            Swap_w_Rem(P,Q);
            Swap_Rem_Trees(P,Q);
            Reverse(P);
            Swap_w_Rem(Q,P);
            Retreat(P);

            Advance(last,P);
            Swap_w_Rem(P,Q);
            Swap_Rem_Trees(P,Q);
            Reverse(P);
            Swap_w_Rem(Q,P);
```

```
            Retreat(P);
            Decrement(last);

            Advance(dir,P);
            Swap_Rem_Trees(P,Q);
            Retreat(P);
            Increase(dir);
         end;
         If(dir = last) then
            Advance(dir,P)
            Reverse(P);
         end;
   end Reverse_Rem_Tr;
end Obvious_Reversal_Realiz;
```

Figure 28: Tree Reversal Realization

The final enhancement to be discussed is Node_Height with specifications shown in Figure 29. Node_Height of a node `x` will return an integer representing the longest path from `x` to an `Empty_Tree`, in the specification, node `x` will always be the root node of the `Rem_Tr` as stated in the **ensures** clause. The realization of this enhancement is shown in Figure 30.

```
   Enhancement Node_Height_Capability for Exploration_Tree_Template;

      Operation Node_Height( restores P: Tree_Posn ): Integer;
         ensures Node_Height = ( ht(P.Rem_Tr ) );

   end Node_Height_Capability;
```

Figure 29: Enhancement specifications for Node Height operation

```
Realization Node_Height_Realiz for Node_Height_Capability
                                        of Exploration_Tree_Template
   Recursive Procedure Node_Height ( restores P: Tree_Posn ): Integer
      decreasing ht(P.Rem_Tr) ;

      Var MaxHeight, NextHeight, dir: Integer;
      MaxHeight := 0;
      NextHeight := 0;
      dir := 1;

      If (At_an_End(P)) then
         Node_Height:= 0 ;
      else
         while ( dir < = k ) then
               maintaining P.Path = #P.Path and
                  P.Rem_Tr = #P.Rem_Tr and
                  MaxHeight =
                      Max(  ht(Split_at(d÷1,P.Rem_Tr).RT);
                    d=1 to dir-1
               decreasing (k - dir);
         do
               Advance(dir, P);
               NextHeight := Node_Height(P);
               If (MaxHeight < NextHeight) then
                  MaxHeight := NextHeight;
               end;
               Retreat(P);
               Increment(dir);
         end;
         Node_Height:= 1 + MaxHeight;
      end;
   end Node_Height;
end Node_Height_Realiz;
```

Figure 30: Realization of the operation Node_Height

CHAPTER THREE

A GENERAL, MAP CONCEPT SPECIFICATION AND A TREE-BASED REALIZATION

Searching for information is one of the main topics of interest in computing and a map data abstraction encapsulates this idea. The abstraction allows information to be associated with key values in such a way that it is possible to search, retrieve, delete, or modify information associated with a key value efficiently. This chapter first presents a detailed explanation of Almost_Constant_Function_Template that captures this data abstraction. Later in the chapter, a balanced binary search tree based map implementation will be explained where an Almost_Constant_Function_Template is used as an interface. The concept includes operations to navigate through the keys in an orderly fashion. For brevity, most of the figures used to support the explanations will just use sections of the concept; A detailed version of the concept is found on Appendix B.

An Informal Introduction to Almost Constant Function Template

The Almost_Constant_Function_Template is the specification of a generic data abstraction for searching and Figure 31 shows an informal specification of this template. The generic nature of this template is defined by the type of both `Index` and `Range_Value` provided during instantiation. The type family `A_C_Fn` is modeled as a total function where indices are mapped to range values. In the template, a default value `C` is taken as a parameter so that the positions of the function with no explicit assigned value will be mapped to this default value. To illustrate this model, consider Figure 32 which shows a mapping of integers to real numbers. Initially all indices will be mapped to the default value C and as non-default range values (deviations) are associated with index values and added into the function they deviate from the default values.

Currently the example function in Figure 32 has three deviations, 2.1, 1.2 and 2.3, and we can insert, remove or swap values in the function. To achieve this, an operation Swap_Value is defined. This operation uses its three parameters to achieve all three actions with the same operation. For example, to insert a new value, Swap_Value parameter V will have the new value to be inserted to the function at a specified index i which is currently mapped to a default value C. To remove an existing value, Swap_Value will have the default value C passed in as V to an index i which is currently mapped to a deviation. Swapping happens when a new value is to be inserted to an index that is not mapped to a default value.

Navigating the function can be achieved in the order of indices that the client define (in Figure 31 the index i is defined to precede j) by three operations, First_Int_Index, Next_Int_Index and Would_Be_Last. Fist_Int_Index it gives the first interesting index in the function and that is the first index not mapped to a default value, in Figure 32 this would be 2. From 2 we can move to the next interesting index using Next_Int_Index operation, if we loop this operation by getting the next index the entire function can be navigated until the last index. To know if an index is the last one and all interesting key values have been navigated, a Boolean operation Would_Be_Last is used. The ability to navigate (in order) is necessary to copy a map or to print a map, for example.

```
Concept Almost_Constant_Function_Template( type Index, Range_Value;
   def const C: Range_Value; evaluates Dev_Ct_Max: Integer;
                            def const (i: Index) ⊴ (j: Index): B )

      Family A_C_Fn ⊆ (Index→Range_Value);

      Operation Swap_Value( updates V: Range_Value; updates F: A_C_Fn;
                                                  restores i: Index );
      Operation First_Int_Index( replaces i: Index; restores F: A_C_Fn);

      Operation Next_Int_Index( restores i: Index; restores F: A_C_Fn;
                                                replaces r: Index );
      Operation Would_Be_Last( restores i: Index; restores F: A_C_Fn ):
                                                               Boolean;
      Operation Max_Deviation_Ct(): Integer;

      Operation Deviation_Count_of( restores F: A_C_Fn ): Integer;

      Operation Make_Constant( clears F: A_C_Fn );

end Almost_Constant_Function_Template;
```

Figure 31: A Skeleton Interface for Almost Constant Function Template

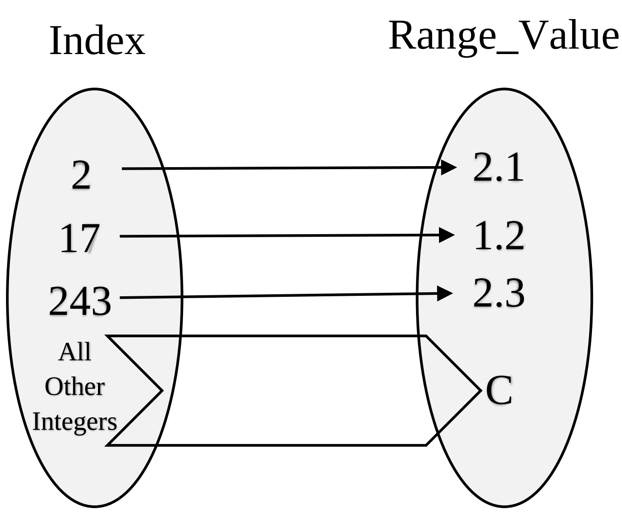


Figure 32: An example “almost constant” map from Integer to Real

To know how many deviations are currently in the function, operation Deviation_Count_of is used. Max_Deviation_Ct will provide the maximum number of deviations you can have in a function.

A Formal Specification of Almost Constant Function Template

A formal specification of Almost_Constat_Function_Template is shown in Figure 33. To instantiate this concept a client should provide the type of both `Index` and `Range_Value`. `Dev_Ct_Max` which is an integer and provided during instantiation will set the maximum number of deviations the function can have; this value is constrained by the specified concept level **requires** clause which state that the `Dev_Ct_Max` is at least `1`. The concept imports as a parameter an ordering of indices that will allow the client to use the operations provided in the concept to navigate per order of these indices. The concept level **requires** clause specifies this ordering of indices to be of total ordering using the mathematical predicate `Is_Total_Ordering(⊴)`.

The mathematical modeling of an A_C_Fn is a function from `Index` to `Range_Value`. Using `F` as an exemplar for A_C_Fn, a Deviation Count of `F(Deviation_Count(F))` state how many indices in `F` are not mapped to the default value `C`; This count is constrained to be less than or equal to the `Dev_Ct_Max` as stated in the **constraint** clause. For every function `F` constructed, the **initialization** clause will map every index to a default value `C`.

```
Concept Almost_Constant_Function_Template(type Index,Range_Value;
   def const C: Range_Value; evaluates Dev_Ct_Max: Integer;
                                def const (i: Index) ⊴ (j: Index): B );
                                         *Deviation Count Maximum *)
      uses Std_Integer_Fac, Std_Boolean_Fac,Basic_Ordering_Theory;
      requires 1 ≤ Dev_Ct_Max and Is_Total_Ordering(⊴ );

   Family A_C_Fn ⊆ (Index→Range_Value);(* Almost Constant Function *)
      exemplar F;
      Def Const Deviation_Count( F: A_C_Fn ): ℕ =
                                  ( ‖{ i: Index | F(i) ≠ C }‖ );
      constraint
         Deviation_Count( F ) ≤ Dev_Ct_Max;
      initialization
         ensures F = λ i: Index.( C );

   Oper Swap_Value( updates V: Range_Value; updates F: A_C_Fn;
                                                   restores i: Index );
      requires  Deviation_Count(F) < Dev_Ct_Max or F(i) ≠ C or V = C;
      ensures F(i) = #V and V = #F(i) and
                        ∀ j: Index, if j ≠ i then F(j) = #F(j);
      ⋮
      ⋮

end Almost_Constant_Function_Template;
```

Figure 33: A Formal Specification of Almost Constant Function Template

Formally, Swap_Value operation is specified as shown in Figure 33. Its specification includes several **requires** clauses which are disjunctions: The first one is `Deviation_Count(F) < Dev_Ct_Max` which requires a function to have space before inserting a new value. The second requirement is `F(i) ≠ C`, and this requirement comes into picture when a new `Range_Value` is intended to replace existing `Range_Value`. The last one is `V = C`, this requirement covers a case when a default value `C` is passed in as an incoming `Range_Value` and is synonymous to resetting an existing value to a default value. The **ensures** clause for this operation essentially swaps whatever is in the function at an index i (i.e. F(i)) to V and V to F(i) and everything else in the function is unchanged.

The next four definitions shown in Figure 34 are helper definitions locally defined and intended to make the rest of the operations easier to specify. The first definition is for the predicate “less than” that is true if and only if when given two indices i and j, index i strictly precedes j i.e., when $i \trianglelefteq j$ and $i \neq j$. The second definition Are_Devs_after tells us if there are any deviations after the current index i. Is_1st_Dev_after it tells what is the next index after the given index i that is not mapped to default value C. The last definition Is_1st_Dev tells if everything before i are mapped to C, implying i is the first deviation.

First_Int_Index is formally defined using Is_1st_Dev to give back the first index of the function whose value is not mapped to C. The **requires** clause of First_Int_Index restrict this operation to be called when there are no deviations within the function. Operation Next_Int_Index uses the definition Are_Devs_after to specify the **requires** clause, Are_Devs_after has to be true to call the operation. If these requirements are met, Next_Int_Index uses Is_1st_Dev_after in **ensures** clause to give back the next index after i. As discussed in the previous section, with these two operations, a client can traverse the entire function, looking for the next interesting index until the last index. To know the last index a Boolean operation Would_Be_Last is available. It specifies the last index to be the one where no more deviations will exist after that and when it is reached all key values associated with non-default range values have been navigated.

```
⋮

Def Const (i: Index) ⊲ (j: Index): B = ( i ⊴ j and i ≠ j );

Def Const Are_Devs_after( i: Index, F: A_C_Fn ): B =
                        ( ∃ k: Index ∍ i ⊲ k and F(k) ≠ C );

Def Const Is_1st_Dev_after( i, k: Index, F: A_C_Fn ): B =
            ( i ⊲ k and F(k) ≠ C and ∀ j: Index, if i ⊲ j ⊲ k, then F(j) = C );

Def Const Is_1st_Dev( k: Index, F: A_C_Fn ): B =
                      ( F(k) ≠ C and ∀ j: Index, if j ⊲ k then F(j) = C );

Operation First_Int_Index( replaces i: Index; restores F: A_C_Fn );
   requires 1 ≤ Deviation_Count (F);
   ensures Is_1st_Dev( i, F );

Operation Next_Int_Index( restores i: Index; restores F: A_C_Fn;
                                             replaces r: Index );
   requires Are_Devs_after( i, F );
   ensures Is_1st_Dev_after( i, r, F);

Operation Would_Be_Last( restores i: Index; restores F: A_C_Fn ):
                                                          Boolean;
   ensures Would_Be_Last = ( ¬ Are_Devs_after( i, F ) );

Operation Max_Deviation_Ct(): Integer;
   ensures Max_Deviation_Ct = ( Dev_Ct_Max );

Operation Deviation_Count_of( restores F: A_C_Fn ): Integer;
   ensures Deviation_Count_of = ( Deviation_Count(F) );

Oper Make_Constant( clears F: A_C_Fn );

end Almost_Constant_Function_Template;
```

Figure 34: A snippet showing specifications for Almost_Constant_Function_Template

AVL Balanced Binary Search Tree-Based Map Implementation

This section presents a balanced binary search tree based map implementation. The idea is to use the generic Exploration_Tree_Template and instantiate it to be a binary tree by supplying the value `k` as `2`. However, to exploit the natural ordering of Binary Search Tree (BST) additional constraints are provided in the realization, one that guarantees that the binary tree maintains binary search tree (BST) property and another that assures that the tree is balanced for fast performance.

Realization Parameter Operations

The BST_Realiz for Almost_Constant_Function_Template implements all the operations specified in the interface and to make the implementation both modular and efficient, the realization includes several imported and locally defined operations and definitions which are not part of the concept.

Since the Index and Range_Value types are supplied by the user and may be non-trivial, no operations on these types—not even assignment for copying and equality checking—may be assumed to exist automatically. Users must provide suitable parameters depending on the actual Index and Range_Value types. These operations that need to be supplied by the users include ones needed for the ordering of indices, copying an index, assigning new default value and one to check if a given value is a default value. Since the type of `Index` and `Range_Value` are supplied as parameters when the template is instantiated, all these operations are also provided as arguments. The four operations are defined in the parenthesis as the realization parameters are In_Order, Replica, New_Dflt_RV and Is_Dflt_RV as shown in Figure 35. For brevity, Figure 35 and other figures used in this section will only show sections of BST_Realiz and a detailed version of it is given in Appendix C.

```
Realization BST_Realiz (                   (* Binary Search Tree *)
                  Operation In_Order (restores i, j: Index): Boolean;
                     ensures In_Order = ( i ⊴ j );
                  Operation Replica(restores i: Index): Index;
                     ensures Replica = ( i );
                  Operation New_Dflt_RV(): Range_Value;
                     ensures New_Dflt_RV = ( C );
                                          (* New Default Range Value *)
                  Operation Is_Dflt_RV(V:Range_Value): Boolean;
                     ensures Is_Dflt_RV = ( V = C );
                                          (* Is Default Range Value *)
                                ) for Almost_Constant_Function_Template;
         uses Exploration_Tree_Template;

   Operation Are_Equal(restores i, j: Index): Boolean;
      ensures Are_Equal = ( i = j );
      procedure
      Are_Equal := In_Order(i, j) and In_Order(j, i);
   end Are_Equal;

   Operation Precedes(restores i, j: Index): Boolean;
      ensures Precedes = ( i ⊲  j );
      procedure
      Precedes := In_Order(i, j) and not In_Order(j, i);
   end Precedes;
   ⋮

   ⋮

end BST_Realiz;
```

Figure 35: Binary Search Tree Realization

Key Value Pair as a Record Structure

In Figure 36, a local Facility is described by instantiating an Exploration_Tree_Template realized by Obv_Exploration_Tree_Realiz. The goal is to supply appropriate arguments to create a tree structure that will be useful in implementing maps. One of the parameters is `Node_Label`. Having maps being represented by a key and value pair, a record structure is created of **Type** `IRV_Pair` with two fields, `id` for the `Index` and `V` for `Range_Value`. Therefore, every single `IRV_Pair` will have both `id` and `V` which will serve as a `Node_Label`. The second parameter define the number of children needed for the tree created and for this case `2` is supplied for binary tree. Lastly, a `Dev_Ct_Max` is provided as the `Initial_Capacity` of the tree. This declaration also includes three enhancements to Exploration_Tree_Template that will be useful in several implementations of different operations. Following this Facility declaration are two local definitions, Is_Dflt_C_Free and a predicate represented by the symbol ◄ which will be explained later.

```
Realization BST_Realiz (
    ⋮
    ⋮

   Type IRV_Pair = Record            (* Index Range Value Pair *)
                   id : Index;
                   V: Range_Value;
                  end;

   Facility Tree_Fac is Exploration_Tree_Template (IRV_Pair, 2,
                                                           Dev_Ct_Max)
                        realized by Obv_Exploration_Tree_Realiz
                    enhanced by Node_Count_Capability
                        realized by Obv_Node_Count_Realiz
                    enhanced by Deletion_Capability
                        realized by Obvious_Deletion_Realiz
                    enhanced by Node_Height
                        realized by Obv_Node_Height_Capability_Realiz;

   Definition Is_Dflt_C_Free ( T: Tr(IRV_Pair) ): B =
                              ( ∀ p: Occ_Set( T.Path Ψ T.Rem_Tr ),
    (* Is Default Constant Free *)                              p.V ≠ C);

   Definition Is_Antitransitive( ρ: (D: Set)⊠D→B ) =
            ( ∀ x, y, z: D, if ¬ x ρ y and ¬ y ρ z, then ¬ x ρ z );

   Definition (p: IRV_Pair) ◄ (q: IRV_Pair): B = ( p.id ⊲ q.id );
    (* Is Pair Less Than *)
      Corollary 1: Is_Transitive(◄) and Is_Asymmetric(◄)and
                                          Is_Antitransitive(◄);
    ⋮
end BST_Realiz;
```

Figure 36: Binary Search Tree Realization

Conventions and Correspondence

Figure 36 defines a **record** of **Type** `A_C_Fn` which has two fields, `TP` which is a `Tree_Posn` and a `Last_Id` which is an index in the tree that is the maximum of all the indices in the tree. The **convention** and **correspondence** are a part of this record. The use of these assertions in verification of the implementation are discussed elsewhere [6].

To simplify expressions of the **convention** and **correspondence** assertions in this realization, the mathematical definitions Fn_Sub_Gr (Function Subgraph), Dom_Set (Domain Set) and Rpd_Fn (Represented Function) are specified. Fn_Sub_Gr is a power set of power set of IRV_Pair, and it defines a unique existence of an index (id) and a value (V) for a given power set of IRV_Pair.From this definition, it follows that an occurrence set of a binary search tree which has IRV_Pair as nodes is a Fn_Sub_Gr as stated in the corollary. Dom_Set is a power set of indices and for an index `i` in IRV_Pair. The corollaries state that, there exists a unique IRV_Pair with `i`, and there will be only one mapping of that index to Range_Value, unless the index is not in the Dom_Set in which case it will be mapped to `C`. Definitions Fn_Sub_Gr and Dom_Set are used to define Rpd_Fn which is a function that takes indices and maps those which are in the tree to explicit values and those which are not to a default value `C`. Rpd_Fn captures the almost constant function that is represented in a tree structure.

The **convention** assertion also known as representation invariant will keep the implementation of the operations consistent by providing conditions that may be assumed true at the beginning of every external operation, and must be shown to be at the end of each operation leaving the representation still satisfying the **convention**. In Figure 37, the **convention** contains a predicate Is Left Right Conformal with (Is_L_R_Cfml_w) which uses the predicate ◄ defined in Figure 36. ◄ is a Boolean predicate that returns true when the left index is less than the right index. Is_L_R_Cfml_w describes the BST property of the tree representation and it is formally defined in the extension Left_Right_Conformality_Ext for General Tree Theory illustrated on Appendix F. For performance Is_Balanced predicate is used in the **convention** and will be explained in details at the end of this chapter. Another predicate is Is _Dflt_C_Free which is defined in Figure 36 and it guarantees that, every operation implemented will not leave a default value `C` stored within the structure. The other part of the **convention** describes an index `Last_Id` to be in the occurrence set and any other index the set will have is less than `Last_Id`. The subordinate annotation **which_entails** is included in this specification to explicitly assure the type checker that the Occ_Set is a Fn_Sb_Gr.

```
Realization BST_Realiz (

      ⋮
      ⋮

   Def. Fn_Sub_Gr:℘(℘(IRV_Pair)) =  (* Function SubGraph *)
            {S:℘(IRV_Pair)|∀ p, q: S, if p.id = q.id,
                                                    then p.V = q.V };
      Corollary 1: ∀ T: U_Tr_Pos(2, IRV_Pair), if Is_L_R_Cfml_w (◄, T),
                                                  then Occ_Set(T): Fn_Sub_Gr;

   Def. Dom_Set( S:℘(IRV_Pair) ):℘(Index) =
                  { i: Index|∃ p: S ∍ i = p.id };  (* Domain Set *)
         Corollary 1: ∀ S: Fn_Sub_Gr, ∀ i: Dom_Set(S), ∃! p: S ∍ i = p.id;
         Corollary 2: ∀ S: Fn_Sub_Gr, ∃! F: Index→Range_Value ∍
                  ∀ p: S, F(p.id) = p.V and ∀ i: (Index~Dom_Set(S)), F(i) = C;

   Implicit Def. Rpd_Fn( S: Fn_Sub_Gr ): Index→Range_Value  is
                 ∀ p: S, Rpd_Fn(S)(p.id) = p.V and
                       ∀ i: (Index~Dom_Set(S)), Rpd_Fn(S)(i) = C;
                 (* Represented Function *)

   Type A_C_Fn = Record
                  TP: Tree_Fac.Tree_Posn; (* Tree Position *)
                  Last_Id : Index;        (* Last Index *)
                 end;

         convention Is_L_R_Cfml_w(◄, F.TP.Path Ψ F.TP.Rem_Tr )
                    which_entails
                    Occ_Set( F.TP.Path Ψ F.TP.Rem Tr ): Fn_Sub_Gr and
                    Is_Balanced (F.TP) and Is_Dflt_C_Free (F.TP) and
                    ∀ p: Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr),
                                             p.id ⊴ F.Last_Id and
                    if Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) ≠ Ω,
                       then ∃ q: Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) ∍
                                             q.id = F.Last_Id;
         correspondence Conc.F =
                       Rpd_Fn( Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) );

       ⋮
       ⋮

end BST_Realiz;
```

Figure 37: Binary Search Tree Realization

The **correspondence** is an abstraction function between the realization representation view and the specification abstract view. The **correspondence** provides a mapping between all values in the realization representation that satisfy the **convention** to the values in the concept. This mapping must be well founded and this fact is established by the proof on the obligations generated by the VC generator for the **correspondence**. In Figure 37, the **correspondence** defines a value in the conceptual function `F (`**`Conc.`**`F)` to correspond a Rpd_Fn of all indices and values in the occurrence set. Occurrence set is a set of all nodes in the realization representation (tree) and is defined in the General Tree Theory to accept a tree and return a set of all nodes within the tree.

Figure 38 summarizes the relationship between the conceptual space and the representation space through **correspondence** using an example function. In the conceptual space an Almost Constant Function example is used within the constraints in this space. On the other hand, is the same function in the tree representation space and satisfy the **convention** which has all the constraints in this space. The two spaces are related through an abstraction function which maps every concrete value that satisfies the convention in the implementation to an abstract value that satisfies the constraints specified in the concept.

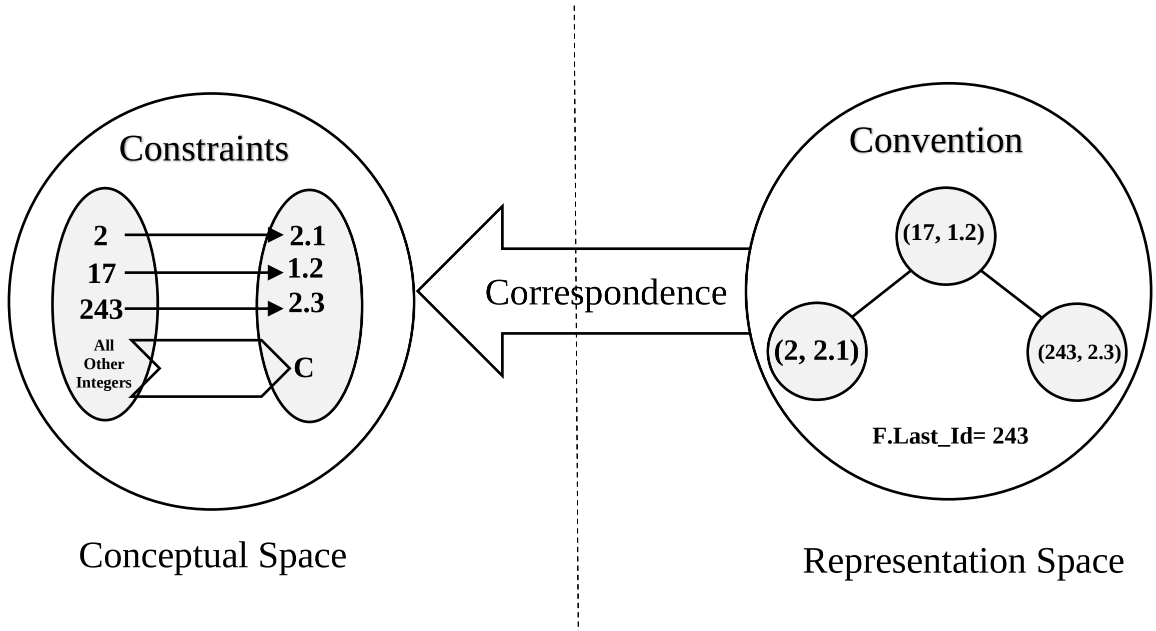


Figure 38: Map implementation

Implementation of Almost Constant Function Operations
Using Locally defined operations.

The implementation contains a variety of local operations to modularize the code further.

The first two local operations Are_Equal and Precedes in Figure 35 use the operation In_Order to define equality and "less than" for the two given indices passed in as parameters to these operations.

Figure 39 shows a local operation Current_Id which returns an index for the root node of the `Rem_Tr` in a `Tree_Posn`. To copy the generic index value, the imported operation Replica is used in the realization of Current_Id.

In Figure 40, a local operation Shift_to_Index_in_Rem_of is specified and implemented. This operation serves as a helper function for operation Shift_to_Index in Figure 43. A Boolean parameter Is_Present is used in the specification and set to true when an index `i` is at some node in the tree and false otherwise. The subordinate annotation **which_entails** is also included in this specification to explicitly assure the type checker that the part of the tree stated is not empty and therefore, it is a legitimate argument in the subsequent use in Rt_Lab (Root Label).

```
Realization BST_Realiz (

     ⋮
     ⋮
   Operation Current_Id(restores F: A_C_Fn ): Index;   (*Current Index*)
      requires F.TP.Rem_Tr ≠ Ω;
      ensures Current_Id = (Rt_Lab (F.TP.Rem_Tr).id );
     procedure
      Var P: IRV_Pair;
      Swap_Label (P, F.TP);
      Current_Id := Replica (P.id );
      Swap_Label (P, F.TP);
   end Current_Id;

      ⋮
      ⋮

end BST_Realiz;
```

Figure 39: Operation Current_Id to return an Index of the root node of Rem_Tr

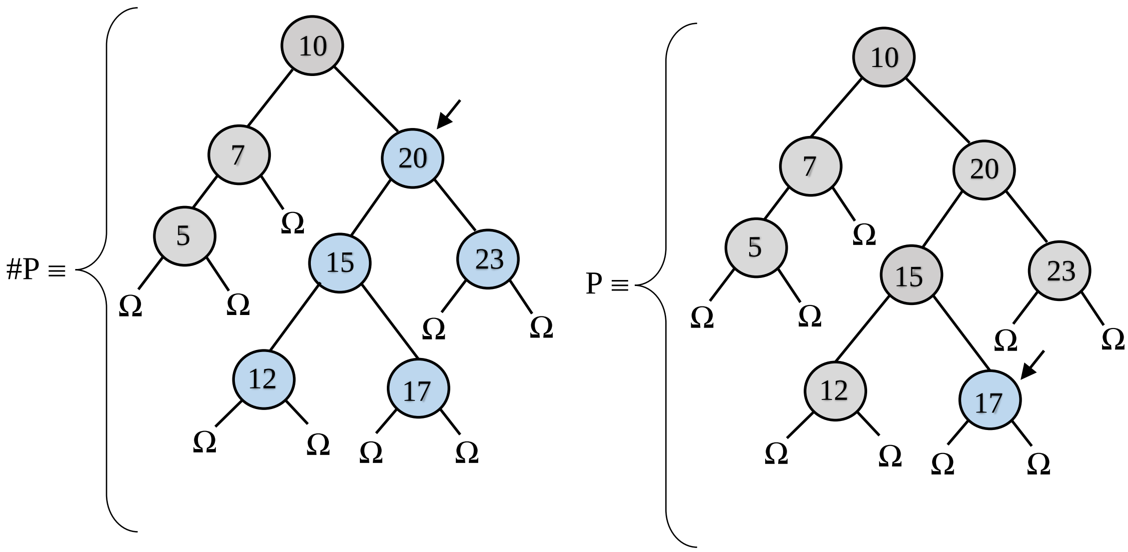
```
Realization BST_Realiz (

     ⋮
     ⋮

   Operation Shift_to_Index_in_Rem_of( updates F: A_C_Fn;
                       restores i: Index; replaces Is_Present: Boolean );
      requires Is_L_R_Cfml_w( ◄, F.TP.Rem_Tr );
      ensures F.TP.Path Ψ F.TP.Rem_Tr = #F.TP.Path Ψ #F.TP.Rem_Tr and
      #F.TP.Path Is_Prefix F.TP.Path and F.Last_Id = #F.Last_Id and
      if i ∊ Dom_Set( Occ_Set(#F.TP.Rem_Tr) ), then Is_Present and
         F.TP.Rem_Tr ≠ Ω ( which_entails F.TP.Rem_Tr:(Tr(IRV_Pair)~{Ω}) )
                                   and Rt_Lab(F.TP.Rem_Tr).id = i and
         if i ∉ Dom_Set( Occ_Set(#F.TP.Rem_Tr) ),
            then ¬ Is_Present and F.TP.Rem_Tr = Ω and
      Is_L_R_Cfml_w( ◄, prt_btwn(|#F.TP.Path|, |F.TP.Path|, F.TP.Path) Ψ
                                                   Jn(⟨Ω⟩², (i, C)) );

   recursive procedure Shift_to_Index_in_Rem_of( updates F: A_C_Fn;
                       restores i: Index; replaces Is_Present: Boolean );

      decreasing ht(F.TP.Rem_Tr);

      If (Are_Equal(i, Current_Id(F))) then
         Is_Present := True();
      else
         If (not At_an_End(F.TP)) then
            If (Precedes(i, Current_Id(F)) then
               Advance (1, F.TP);
            else
               Advance (2, F.TP);
            end;
            Shift_to_Index_in_Rem_of(F, i, present);
         else
            Is_Present := False();
         end;
      end;
   end Shift_to_Index_in_Rem_of;

      ⋮
      ⋮

end BST_Realiz;
```

Figure 40: Binary Search Tree Realization

In the specifications for Shift_to_Index_in_Rem_of, the **ensures** clause of the operations assures that no changes are made to the tree contents, A conjunction `F.Last_Id =`

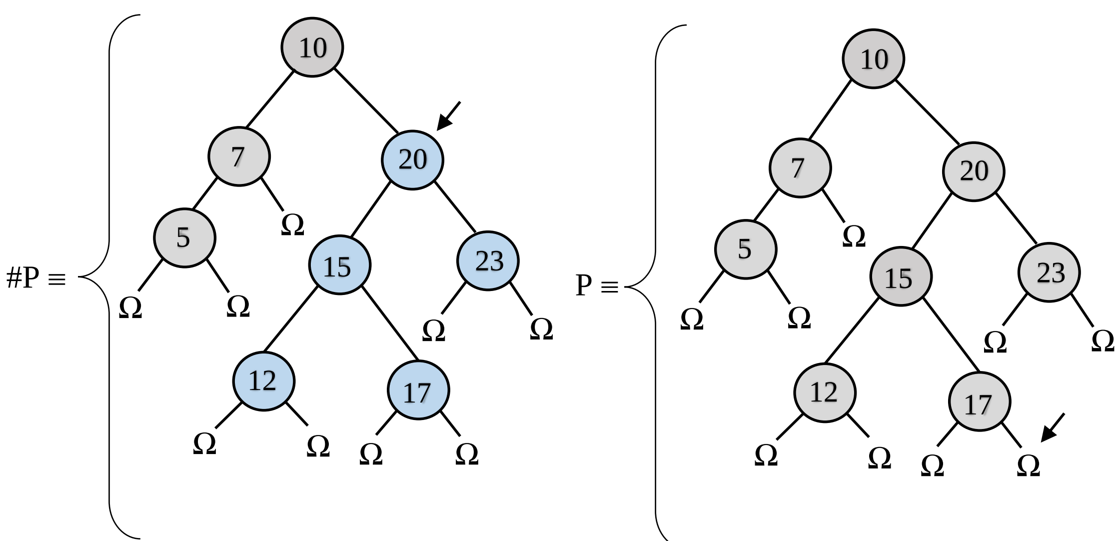
#F.Last_Id guarantees that F.Last_Id is unchanged. The **ensures** clause also addresses a case when an index i is present in the tree and in this case a Tree_Posn will be updated in such a way the root node of the Rem_Tr will have an id equal to the index i specified as input parameter. The last part of the **ensures** clause is the case when an index i is not present in the tree, and in this situation, we expect after the entire search for an index i, the search will stop with Rem_Tr of the Tree_Posn being Empty_Tree and at the same time to stop at a position that in case we were to add that non-existing index i then it will still satisfy the BST property.

The operation Shift_to_Index uses Shift_to_Index_in_Rem_of. In the implementation, a local check before resetting a tree is performed, this will help in the cases where resetting is unnecessary and so improving efficiency. To illustrate the effect of this operation, consider a Tree_Posn in Figure 41(a) which is currently at an index 20. If we shift to an index 17, the resulting Tree_Posn is shown in Figure 41(b). Figure 42 shows a case when we shift to an index not present, for example, if we shift to index 18.



(a) **(b)**

Figure 41: (a) Tree position at index 20 (b) the resulting tree position at index 17



(a) **(b)**

Figure 42: Tree position at index 20 (b) Resulting tree position at index 18 which is not present in the tree

```
Realization BST_Realiz (
     ⋮
     ⋮

  Operation Shift_to_Index ( updates F: A_C_Fn;
                    restores i: Index; replaces Is_Present: Boolean );
      requires Is_L_R_Cfml_w( ◄, F.TP.Rem_Tr );
      ensures F.TP.Path Ψ F.TP.Rem_Tr = #F.TP.Path Ψ #F.TP.Rem_Tr and
      #F.TP.Path Is_Prefix F.TP.Path and F.Last_Id = #F.Last_Id and
      if i ∈ Dom_Set( Occ_Set(#F.TP.Rem_Tr) ), then Is_Present and
         F.TP.Rem_Tr ≠ Ω (which_entails F.TP.Rem_Tr:(Tr(IRV_Pair)~{Ω}))
                                        and Rt_Lab(F.TP.Rem_Tr).id = i and
         if i ∉ Dom_Set( Occ_Set(#F.TP.Rem_Tr) ),
            then ¬ Is_Present and F.TP.Rem_Tr = Ω and
      Is_L_R_Cfml_w( ◄, prt_btwn(|#F.TP.Path|, |F.TP.Path|, F.TP.Path) Ψ
                                                     Jn(⟨Ω⟩², (i, C)) );

   procedure Shift_to_Index ( updates F: A_C_Fn; restores i: Index;
                                         replaces Is_Present: Boolean );

      If (Path_Length(F.TP) ≥ 1 and Precedes(i, Current_Id(F)) then
         Reset(F.TP);
      end;
      Shift_to_Index_in_Rem_of (F, i, Is_Present);

   end Shift_to_Index;
      ⋮
      ⋮

end BST_Realiz;
```

Figure 43: Binary Search Tree Realization

Shift_to_Index_in_Rem_of uses recursion in binary search to find an index `i` of the desired value. The base cases are when the search ends up with a root node `id` on the `Rem_Tr` equal to `i` as a case in Figure 41(b) or end up with an `Empty_Tree(Ω)` at the exact position `i` was to be in if it were present as is the case in Figure 42(b).

Figure 44 specifies another local operation Shift_to_First. This operation walks through the left spine of the binary search tree and stops at the first index; this will be the first node in the in-order traversal of the tree. To make sure this operation walks in the correct spine that leads to the first node, a requirement is set that the Path should be Empty_String(Λ) and the Rem_Tr not an Empty_Tree(Ω). The **ensures** clause guarantees that the contents of the tree and the last index (Last_Id) are unchanged after the operation and that the root node id of the outgoing Rem_Tr is the first deviation of the given tree. The recursive implementation of this operation just Advances to the left of the tree until it finds the first node.

```
Realization BST_Realiz (

      ⋮
      ⋮

   Operation Shift_to_First (updates F: A_C_Fn )
        requires F.TP.Path = Λ and F.TP.Rem_Tr ≠ Ω;
        ensures F.TP.Path Ψ F.TP.Rem_Tr = #F.TP.Path Ψ #F.TP.Rem_Tr and
                  F.Last_Id = #F.Last_Id and
                  F.TP.Rem_Tr ≠ Ω
                        (which_entails F.TP.Rem_Tr: (Tr(IRV_Pair)~{Ω})
                  and (Rt_Lab( F.TP.Rem_Tr )).id = i and
                        Is_1st_Dev ( Rt_Lab( F.TP.Rem_Tr ).id, F.TP );

   Recursive Procedure Shift_to_First ( updates F:A_C_Fn );
      decreasing ht(F.TP.Rem_Tr );

      If (At_an_End (F.TP)) then
         Retreat (F.TP);
      else
         Advance (1, F.TP);
         Shift_to_First (F);
      end;
   end Shift_to_First;

      ⋮
      ⋮

end BST_Realiz;
```

Figure 44: Shift to First operation in BST Realization

The next operation Delete_Rt_Node specified in Figure 45 is a local operation and it is used in procedure Swap_Value. Delete_Rt_Node will remove a node from a tree and its specifications shows that the operation will affect the `Remaining_Cap` any time it is called. The pre-condition to this operation requires that the `Rem_Tr` not be `Empty_Tree(Ω)`. The other requirements are that the tree must be a search tree and balanced. The **ensures** clause guarantees no modification to the `Path`, and because in the end, the operation gets rid the root of the incoming `Rem_Tr`, then the root node of the incoming `Rem_Tr` will no longer be a member of the occurrence set. The operation also must ensure that the resulting tree is still a search tree and in case of imbalance that may be caused by deletion that it will not lead to a difference in height being greater than 2. Lastly, the value of `Last_Id` will still be the last index of the updated `Tree_Posn`.

```
Realization BST_Realiz (
     ⋮
     ⋮
   Operation Delete_Rt_Node ( updates F: A_C_Fn);
      affects Remaining_Cap;
      requires F.TP.Rem_Tr ≠ Ω and
                  Is_L_R_Cfml_w(◄, F.TP.Rem_Tr Ψ F.TP.Path ) and
                            Is_Balanced (F.TP.Path Ψ F.TP. Rem_Tr);
      ensures F.TP.Path = #F.TP.Path and
            Occ_Set (F.TP. Rem_Tr) = Occ_Set (#F.TP.Rem_Tr) -
                                          {Rt_Lab(#F.TP.Rem_Tr)} and
            Is_L_R_Cfml_w (◄, F.TP.Rem_Tr Ψ F.TP.Path ) and
            Remaining_Cap = #Remaining_Cap +1 and
         If (F.TP.Rem_Tr ≠ Ω
                 (which_entails F.TP.Rem_Tr: (Tr(IRV_Pair)~{Ω}))then
            0 ≤ |ht(Split_at(0, F.TP.Rem_Tr).RT) -
                             ht(Split_at(1, F.TP.Rem_Tr).RT)| ≤ 2 and
            (∀p: Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr), p.id ⊴ F.Last_Id
         and if Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) ≠ Ω, then
            ∃q: Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) ∍ q.id = F.Last_Id;
   Procedure Delete_Rt_Node (updates F: A_C_Fn);
      Var L, R : A_C_Fn;
      Var P : IRV_Pair;
      Var Is_Last_Id: Boolean;
      Is_Last_Id := False();

      If (Are_Equal((Rt_Lab(F.TP.Rem_Tr)).id, F.Last_Id)) then
         Is_Last_Id := True();
      end;

      If (At_a_Leaf(F)) then
         Remove_Leaf(P, F.TP);
      else
         Advance(1, F.TP);
         If(At_an_End(F.TP)) then
            Retreat(F.TP);
```

```
   Advance(2, F.TP);
   Swap_Rem_Trees(R.TP, F.TP);
   Retreat(F.TP);
   Remove_Leaf(P, F.TP);
   Swap_Rem_Trees(R.TP, F.TP);
else
   Retreat(F.TP);
   Advance(2, F.TP);
   If(At_an_End(F.TP)) then
         Retreat(F.TP);
         Advance(1, F.TP);
         Swap_Rem_Trees(L.TP, F.TP);
         Retreat(F.TP);
         Remove_Leaf(P, F.TP);
         Swap_Rem_Trees(L.TP, F.TP);
   else
         Retreat(F.TP);
         Advance(1, F.TP);
         Swap_Rem_Trees(L.TP, F.TP);
         Retreat (F.TP);
         Advance(2, F.TP);
         Swap_Rem_Trees(R.TP, F.TP);
         Retreat(F.TP);
         Remove_Leaf(P, F.TP);
         Shift_to_First(R);
         Swap_Rem_Trees(R.TP, F.TP);
         Reset (R.TP);
         Advance (1, F.TP);
         Swap_Rem_Trees(L.TP, F.TP)
         Retreat (F.TP);
         Advance(2. F.TP);
         If (At_an_End(F.TP)) then
         Swap_Rem_Trees(R.TP, F.TP);
      else
         Advance(2, F.TP);
         Swap_Rem_Trees(R.TP, F.TP);
         Retreat(F.TP);
      end;
         Retreat(F.TP);
   end;
end;
If (Is_Last_Id and At_an_End(F)) then
   Retreat(F.TP);
   F.Last_Id := Current_Id(F);
   Advance(2, F.TP);
else
   If (Is_Last_Id) then
      F.Last_Id := Current_Id(F);
   end;
end;
```

```
        end;
    end Delete_Rt_Node;
        ⋮
        ⋮
end BST_Realiz;
```

Figure 45: Procedure Delete Root Node in BST Realization

The implementation of Delete_Rt_Node operation considers three cases that a node to be deleted x can be in before it gets deleted. The first case is when x is a leaf, this is a trivial case where Remove_Leaf will just be called on x. The second case is when x has either no right or no left child. In this case the implementation takes two steps to delete x and reconstruct the tree, first removing the existing child of x which includes everything rooted at this child, making node x a leaf and so reverting back to the first case. At this point Remove_Leaf can be called on x and the only remaining task will be to reconnect what used to be a child of x to be the parent of x. It is easy to observe that whichever scenario in the second case is true, reconstruction of the tree will still maintain the BST property. The third case is a non-trivial one where a node x has both children. This is a case illustrated in Figure 46. For this case, several steps are now involved in making sure the respective node is deleted and BST property is maintained. First it is necessary to make the node x a leaf. In the implementation, two tree positions (left, L and right, R) are created for this task. By swapping the right tree branch with R and left tree branch with L node x becomes a leaf which can now be deleted by just calling Remove_Leaf. However, the tricky part falls into the reconstruction part of the tree after getting rid of the intended node. A helpful note on this case is to observe that in L we have every node that was less than x and on the R, we have all nodes that were greater x, this provides two possible ways to reconstruct the tree that still maintain the BST property, first by finding the maximum node in L to takes place of the deleted node `x` or by finding the minimum node in R to take place of the deleted node `x`. In the implementation shown in Figure 45 the latter case is used.

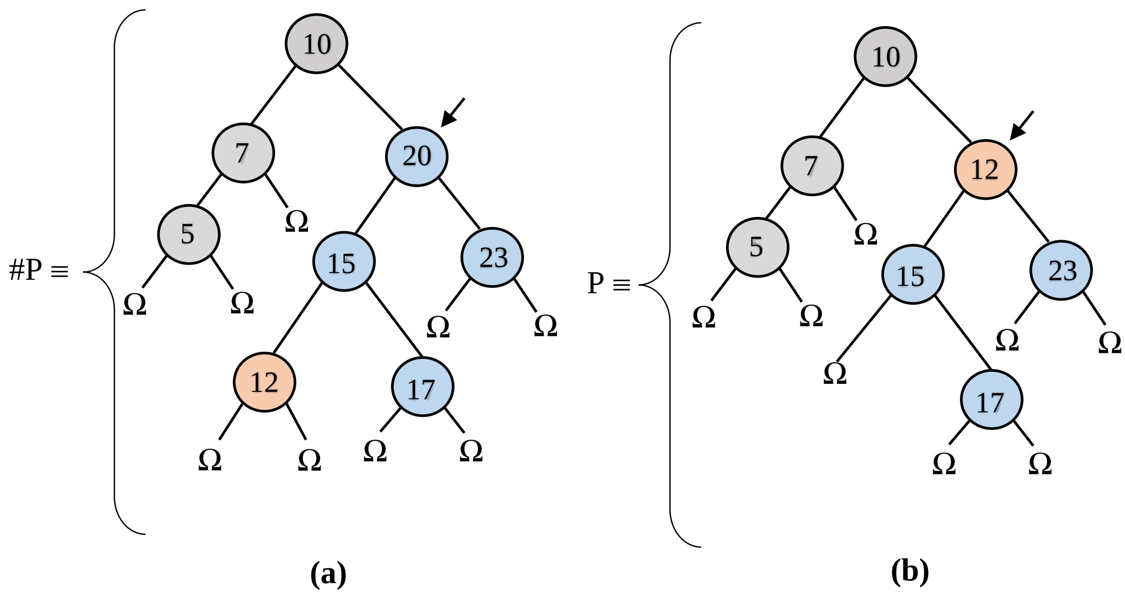


Figure 46: (a) Node to be deleted with both children (b) The result after deletion

The Swap_Value operation can be used to insert a new `Range_Value` into a map, remove an existing `Range_Value` or swap the existing `Range_Value` with a new one at a given index. Figure 47 gives an implementation of this operation. The implementation starts off by shifting to the specified node using a local operation Shift_to_Index. The two results of a Boolean valued variable `present` will branch the implementation in two cases. The first case is when `present` is true and the incoming value is not a default `Range_Value`, this leads to a swap between the incoming `Range_Value` and the one existing at index `i`. If `present` is true but the incoming `Range_Value` is a default value, then the existing `Range_Value` will have to be deleted and Delete_Rt_Node operation is called at this point. When `present` is false and the incoming `Range_Value` is not a default range value, Swap_Value inserts that new value into the map at the specified index `i`.

```
Realization BST_Realiz (

     ⋮

   Procedure Swap_Value(updates V: Range_Value;
                                updates F: A_C_Fn; restores i: Index);
      Var P: IRV_Pair;
      Var present: Boolean;

      P.id := Replica( i );

      Shift_to_Index ( F, i, present );

      If present then
         If not Is_Dflt_RV( V ) then
            P.V :=: V;
            Swap_Label( P, F.TP );
            V :=: P.V;
         else
            Delete_Rt_Node(F);
            V :=: P.V;
            Adjust(F);
         end;
      else
         If not Is_Dflt_RV( V ) then
            P.V :=: V;
            If (Node_Count(F.TP) = 0) then
               F.Last_Id := Replica(P.id);
            else
               If (not In_Order (F.Last_Id, P.id) then
                  F.Last_Id := Replica(P.id);
               end;
            end;
            Add_Leaf ( P, F.TP );
            V := New_Dflt_RV ();
            Adjust(F);
         end;
      end;
   end Swap_Value;

      ⋮

end BST_Realiz;
```

Figure 47: An implementation of operation Swap Value

The operation First_Int_Index will provide the first interesting index of the tree by updating a given `Tree_Posn` to have the first index as the root node of the `Rem_Tr`. The implementation of this operation is shown in Figure 48 and uses a locally defined operation Shift_to_First, in the end the parameter `i` is replaced by the first index.

```
Realization BST_Realiz (

     ⋮
   Procedure First_Int_Index ( replaces i: Index; restores F: A_C_Fn );

      Reset(F.TP )
      Shift_to_First( F );
      i := Current_Id( F );

   end First_Int_Index;
      ⋮

end BST_Realiz;
```

Figure 48: An implementation of operation First Interesting Index

Figure 49 shows an implementation of the operation Next_Int_Index which on a given index `i` will provide the next index after `i` in an in-order traversal of the tree. This implementation considers the fact that the next index after `i` in the in-order traversal of the tree may lie in the right tree branch of the node `i`. Therefore, starting on a `Tree_Posn` with root node `id` of the `Rem_Tr` equals to `i`, Advance (2, F.TP) will navigate the tree to the right tree branch of the node `i`. If the right tree branch is `Empty_Tree(Ω)`, the next index should be in the ancestors of node `i`. Otherwise, next index is expected to be the minimum node on the left tree branch of `Rem_Tr` root node. However, if the left tree branch is `Empty_Tree(Ω)`, then the root node of the `Rem_Tr` is the next index after `i`.

```
Realization BST_Realiz (

     ⋮
     ⋮

   Procedure Next_Int_Index (restores i: Index;
                              restores F: A_C_Fn; replace r: Index );
      Var P: IRV_Pair;
      Var present: Boolean;

      Shift_to_Index (i, F, present);

      Advance (2, F.TP);
      If (At_an_End (F.TP)) then
         Retreat(F.TP);
         While (Precedes (Current_Id(F), i) or Are_Equal(Current_Id(F),i))
            maintaining F.Path Ψ F.Rem_Tr =
            ((Prt_btwn(0, |#F.Path| ∸ 1, #F.Path)) o
            Prt_Btwn (|#F.Path| ∸ 1, |#F.Path|, #F.Path) ) Ψ #F.Rem_Tr ;
            decreasing | F.TP.Path |;
         do
            Retreat(F.TP);
         end;
         r := Current_Id(F)
      else
         Advance(1, F.TP);
         If (At_an_End (F.TP)) then
            Retreat (F.TP);
            r := Current_Id(F)
         else
            Shift_To_First(F);
            r := Current_Id(F);
         end;
      end;
   end Next_Int_Index;

      ⋮
      ⋮

end BST_Realiz;
```

Figure 49: Specification and implementation of operation Next_Int_Index in BST_Realiz

```
Realization BST_Realiz (
     ⋮
     ⋮

   Procedure Would_Be_Last (restores i: Index; restores F: A_C_Fn):
                                                         Boolean;

      If ( Are_Equal (F.Last_Id , i) ) then
         Would_Be_Last := True();
      else
         Would_Be_Last := False();
      end;

   end Would_Be_Last;

   Procedure Max_Deviation_Ct(): Integer;

      Max_Deviation_Ct := Dev_Ct_Max;

   end Max_Deviation_Ct;

   Procedure Deviation_Count_of ( restores F: A_C_Fn ): Integer;

      Deviation_Count_of := Node_Count ( F.TP );

   end Deviation_Count_of;

   Procedure Make_Constant ( clears F: A_C_Fn );

      Reset( F.TP );
      Delete_Remainder( F.TP );

   end Make_Constant;

end BST_Realiz;
```

Figure 50: A snippet showing BST_Realiz

The Boolean operation Would_Be_Last is implemented as shown in Figure 50, for a specified index `i`, Would_Be_Last will return true if `i` is the last index in the in-order traversal of the tree. The implementation compares the incoming provided index with the `Last_Id`.

The last three operations in Figure 50 are somewhat easy and direct to understand, Max_Deviation_Ct() will return the maximum number of deviations in a given map, this is implemented by just equating Max_Deviation_Ct to `Dev_Ct_Max` provided during instantiation of the template. The second operation, Deviation_Count_of is implemented by a Node_Count of the given `Tree_Posn`. Make_Constant is implemented by use of Delete_Remainder enhancement where the entire is deleted.

AVL Binary Search Tree Balancing

Using a BST in this implementation makes it possible to achieve an worst-case complexity of O(log Dev_Count_of(F)) or better for the map operations. However, in the worst case the performance can be as poor as on a linked list if the BST is not well maintained during insertion and deletion of a nodes. Consider a case when a sorted sequence of keys is inserted into a BST. If there is no mechanism to readjust the tree height as the elements are inserted, the final structure will be a linked list and searching can have a linear worst-case performance O(Dev_Count_of (F)). To solve this problem balancing is necessary in BST which will promise a logarithmic worst-case performance in all operations. In this implementation, a predicate Is_Balanced is added to the **convention** to guarantee that the external operations keep the representation balanced.

Balancing will achieve proper branching of the BST and it does this by re-balancing every time there is a change in the tree whether by inserting or deleting a node. There are several balancing techniques that exists in theory and practice. For this implementation, a worst-case mechanism AVL trees is used. AVL trees are height-balanced trees and named after two inventors Adel'son-Vel'skii G and E.M. Landis [1]. The basic idea of AVL tree balancing mechanism is to guarantee that for every node in the tree, the difference in height for the left sub-tree and right sub-tree is at most ±1. To maintain this balance factor, special operations called rotations will be required to re-adjust the tree nodes whenever a balance factor is violated at a node. Two types of rotations used: Right rotation which is specified and implemented in Figure 52 and left rotation which is specified and implemented in Figure 54. The operation Right_Rotate_Rem_Tr requires the `Rem_Tr` to be left-heavy as stated in the **requires** clause.

This specification also uses a Split_at function which is defined in the General Tree Theory. Split_at will produce a Site and a Remaining Tree from a tree depending on the splitting position provided. Figure 51 illustrates this function. On an example tree (T) in Figure 51(a), Split_at (0, T) will result into a Site and `Rem_Tr` shown in Figure 51(b). Therefore, `Split_at(0, F.TP.Rem_Tr).RT ≠ Ω` simply states that the left subtree of the root node is not empty tree.

At the end of rotation, the **ensures** clause first guarantees that no changes are made to the `Last_Id` and `Path`, however, the `Rem_Tr` will be updated and the specifications in the **ensures** clause uses `Jn` operator and Split_at function to define the resulting `Rem_Tr` after rotation. Left_Rotate_Rem_Tr is the mirror image of Right_Rotate_Rem_Tr and requires the `Rem_Tr` to be right heavy, after left rotation the remaining tree is either left heavy or with subtrees which have same height.

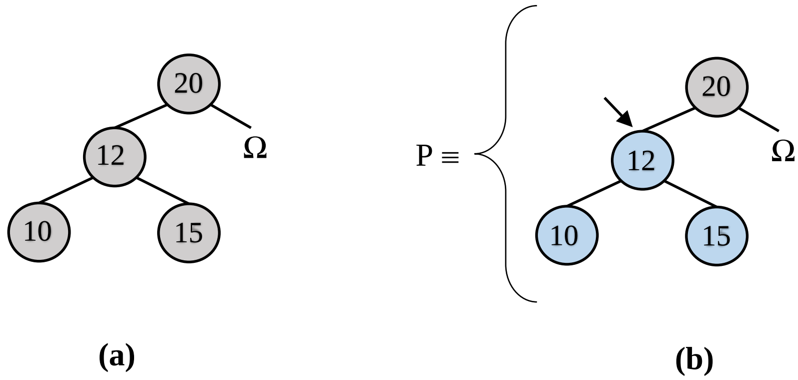


Figure 51: (a) Given 2-Tree T (b) Resulting Site and Remaining Tree after Split_at (0, T)

```
Realization BST_Realiz (

      ⋮
   Operation Right_Rotate_Rem_Tr(updates F: A_C_Fn);
      requires F.TP.Rem_Tr ≠ Ω
            (which_entails F.TP.Rem_Tr: U_Tr(2, IRV_Pair)~{Ω}) and
               Split_at(0, F.TP.Rem_Tr).RT ≠ Ω
            (which_entails Split_at(0, F.TP.Rem_Tr).RT:
                                                    U_Tr(2, IRV_Pair)~{Ω});
      ensures F.Last_Id = #F.Last_Id and
            F.TP.Path = #F.TP.Path and F.TP.Rem_Tr =
            Jn( ⟨ Split_at(0, Split_at(0, #F.TP.Rem_Tr).RT).RT,
              Jn(⟨Split_at(1, Split_at(0, #F.TP.Rem_Tr).RT).RT,
               Split_at(1, #F.TP.Rem_Tr).RT⟩, Rt_Lab(#F.TP.Rem_Tr)) ⟩,
                                  Rt_Lab(Split_at(0, #F.TP.Rem_Tr).RT) );
     Procedure
         Var New_Rem_Tr: Tree_Fac.Tree_Posn;
         Advance (1, F.TP);
         Swap_Rem_Trees (New_Rem_Tr, F.TP);
         Advance (2, New_Rem_Tr);
         Swap_Rem_Trees (New_Rem_Tr, F.TP);
         Retreat (F.TP);
         Swap_Rem_Trees(New_Rem_Tr, F.TP);
         Retreat (New_Rem_Tr);
         Swap_Rem_Trees(New_Rem_Tr, F.TP);
   end Right_Rotate_Rem_Tr ;

      ⋮

end BST_Realiz;
```

Figure 52: A snippet showing operation Right_Rotate_Rem_Tr in BST_Realiz

Both rotations in general are achieved by deterministic number of steps as demonstrated in Figure 53. In this figure, (a) is a tree position with left heavy `Rem_Tr`, A right rotation at this tree position will result into a right heavy `Rem_Tr` shown in Figure 53(b). Alternatively, if we left rotate a tree position in Figure 53(b), it will result into a tree position shown in Figure 53(a). The implementation of these operations uses operation Advance to get to right section of the tree, a temporary variable `T` to hold that section, and a Swap_Rem_Trees operation for movement.

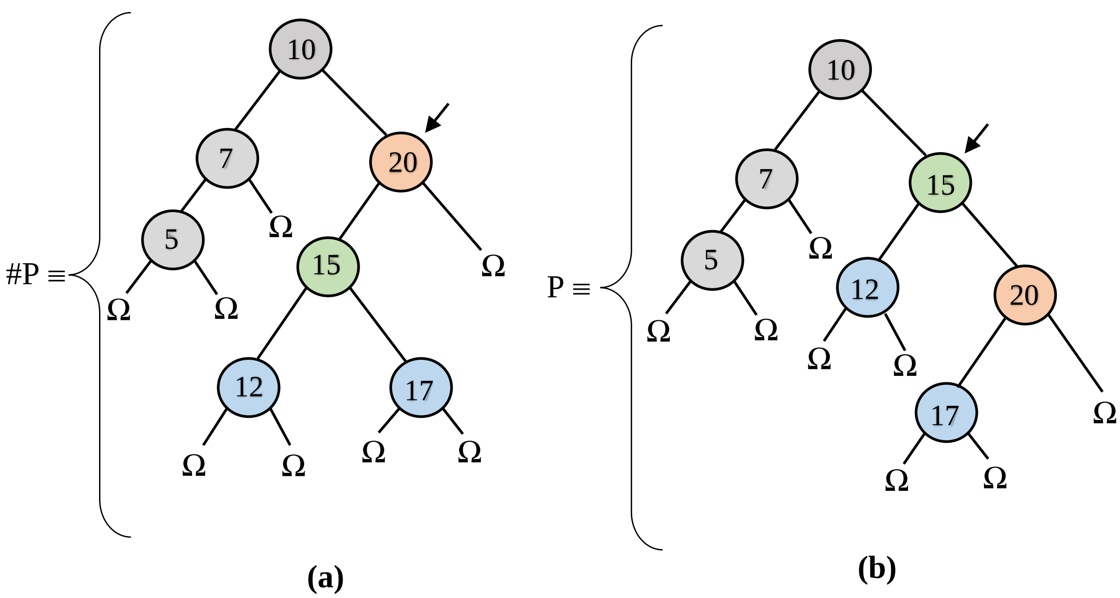


Figure 53: An illustration of Right Rotation and Left Rotation: (a) left heavy (b) Right heavy

```
Realization BST_Realiz (

     ⋮
     ⋮
   Operation Left_Rotate_Rem_Tr (updates F : A_C_Fn);
      requires F.TP.Rem ≠ Ω
            (which_entails F.TP.Rem: U_Tr(2, IRV_Pair)~{Ω}) and
            Split_at(1, F.TP.Rem_Tr).RT ≠ Ω
            (which_entails Split_at(1, F.TP.Rem_Tr).RT:
                                    U_Tr(2, IRV_Pair)~{Ω});
      ensures F.Last_Id = #F.Last_Id and F.TP.Path = #F.TP.Path and
            F.TP.Rem_Tr =
            Jn(⟨Jn(⟨Split_at(0, #F.TP.Rem_Tr ).RT,
               Split_at(0, Split_at(1, #F.TP.Rem_Tr).RT).RT⟩,
                  Rt_Lab(#F.TP.Rem_Tr)),
                        Split_at(1, Split_at(1, #F.TP.Rem_Tr).RT).RT⟩,
                                  Rt_Lab(Split_at(1, #F.TP.Rem_Tr).RT));

   procedure

      Var New_Rem_Tr: Tree_Fac.Tree_Posn;
      Advance (2, F.TP);
      Swap_Rem_Trees (New_Rem_Tr, F.TP);
      Advance (1, New_Rem_Tr);
      Swap_Rem_Trees (New_Rem_Tr, F.TP);
      Retreat (F.TP);
      Swap_Rem_Trees (New_Rem_Tr, F.TP);
      Retreat (New_Rem_Tr);
      Swap_Rem_Trees (New_Rem_Tr, F.TP);

   end Left_Rotate_Rem_Tr;
      ⋮
      ⋮

end BST_Realiz;
```

Figure 54: Specification and implementation of operation Left Rotate in BST Realization

To restore balance of an AVL tree there are several cases to be considered depending on whether the balance violating node is Left-Left heavy, Left-Right heavy, Right-Right heavy or Right-Left heavy. These four cases will also determine the type and number of rotations needed to re-balance the tree. Shown in Figure 59 is operation Adjust which considers the above four cases to reestablish balance of an AVL tree. To simplify implementation of Adjust operation, local operations LT _Height and RT_Height are defined and used with a sole purpose of finding heights of left subtree and right subtree respectively.

```
Operation LT_Height (restores F: A_C_Fn): Integer
   ensures LT_Height = ht (Split_at(1, F.TP.Rem_Tr).RT);
Procedure

   If(At_an_end(F.TP)) then
      LT_Height := 0;
   else
      Advance (1, F.TP);
      LT_Height := Node_Height (F.TP);
      Retreat (F.TP)
   end;
end;

Operation RT_Height (restores F: A_C_Fn) : Integer
   ensures RT_Height = ht(Split_at(2, F.TP.Rem_Tr).RT);
Procedure

   If(At_an_end(F.TP)) then
      RT_Height := 0;
   else
      Advance (2, F.TP);
      RT_Height := Node_Height (F.TP);
      Retreat (F.TP);
   end;
end;
```

Figure 55: Operations LT_Height and RT_Height used in Adjust operation

The implementation of Adjust operation in Figure 59 use the result of the difference between height of the left subtree `(LTHeight)` and height of the right subtree `(RTHeight)` to determine if the respective node maintains the AVL tree balancing. If this difference is less than -1 or greater than 1 re-balancing is required. The entire process of re-balancing needs to identify which case from among the four cases discussed earlier does the balance violation fall into. This classification will require the two values `LTHeight` and `RTHeight`. The two heights are compared and whichever is greater than the other determines which side of the tree is heavier. The implementation is set to eliminate one case after the other. Once the exact case is identified, it will govern the type and number rotations needed to restore the balance.

Two cases LR-Heavy and RL-Heavy mentioned above will require double rotation to achieve balance. The map implementation defines local operation Elevate_Right_Middle and Elevate_Left_Middle to achieve balance in those cases without double rotation. The specification and implementation of Elevate_Right_Middle and Elevate_Left_Middle are shown in Figure 57 and Figure 58, respectively. These specifications are the mirror image of each other.

In Figure 57, the specifications show that operation Elevate_Right_Middle requires the remaining tree not to be empty tree. Split_at function is used to explicitly define which case of a tree this operation can be called. The case identified with the Split_at function is Left – Right Heavy `(Split_at(1, Split_at(0, F.TP.Rem_Tr).RT).RT ≠ Ω)`.

The ensures clause of this operation will guarantee no changes made to the `Last_Id` and `Path`, however, the `Rem_Tr` will be updated as specified using `Jn` operator and Split_at function to represent the updated `Rem_Tr` after Elevate_Right_Middle. As shown in Figure 59, the specification of the operation Adjust requires that the tree satisfies the BST property even before the operation is called and that `Rem_Tr` is `Empty_Tree(Ω)`. After the operation Adjust is called, the **ensures** clause guarantees that the content of the tree and the `Last_Id` are not changed, and that the tree is balanced and still maintains the BST property.

To illustrate the operation Adjust, consider an imbalanced BST tree with a tree position in Figure 56(a). Based on the cases discussion above, this is a Left-Left heavy which will need a single right rotation to restore balance. The resulting tree position is shown in Figure 56(b). The next case shown in Figure 60 is a Left-Right heavy balance violation which would require double rotations in case Right and Left rotations were to be used, in this implementation Elevate_Left_Middle is used and Figure 60(a) shows a balanced case.

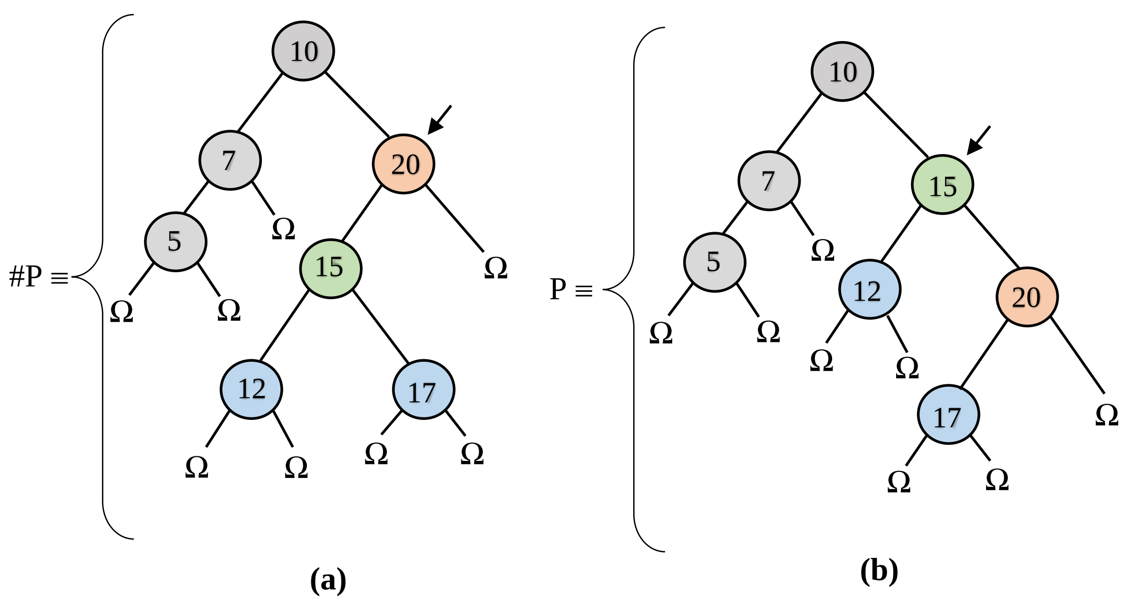


Figure 56: Demonstration of operation Adjust, left-left heavy case: (a) Imbalance tree position (b) balanced tree position after right rotation

```
Realization BST_Realiz (

     ⋮
     ⋮

   Operation Elevate_Right_Middle(updates F: A_C_Fn);
         requires F.TP.Rem ≠ Ω
            (which_entails F.TP.Rem_Tr: U_Tr(2, IRV_Pair)~{Ω}) and
               Split_at(1, F.TP.Rem_Tr).RT ≠ Ω
            (which_entails
               Split_at(1, F.TP.Rem_Tr).RT: U_Tr(2, IRV_Pair)~{Ω}) and
               Split_at(0, Split_at(1, F.TP.Rem_Tr).RT).RT ≠ Ω
            which_entails
     Split_at(0, Split_at(1, F.TP.Rem_Tr).RT).RT: U_Tr(2, IRV_Pair)~{Ω};

         ensures F.Last_Id = #F.Last_Id and F.TP.Path = #F.TP.Path and
               F.TP.Rem_Tr =
               Jn(⟨Jn(⟨ Split_at(0, #F.TP.Rem).RT),
                        Split_at(1,Split_at(0,Split_at(0,#F.TP.Rem)
                              .RT).RT).RT ⟩,Rt_Lab(#F.TP.Rem_Tr)),
                  Jn(⟨Split_at(1, Split_at(0, Split_at(1,#F.TP.Rem_Tr)
                              .RT).RT).RT, Split_at(1, (Split_at(1,
                                                #F.TP.Rem_Tr).RT).RT⟩,
                                Rt_Lab(Split_at(1,#F.TP.Rem_Tr).RT ) ⟩,
                 Rt_Lab(Split_at(0, Split_at(1, #F.TP.Rem_Tr).RT).RT));

   procedure
         Var New_Rem_Tr: Tree_Fac.Tree_Posn;
         Advance (2, F.TP);
         Advance (1, F.TP);
         Swap_Rem_Trees (F.TP, New_Rem_Tr);
         Advance (2, New_Rem_Tr);
         Swap_Rem_Trees (New_Rem_Tr, F.TP);
         Retreat (F.TP);
         Swap_Rem_Trees (F.TP, New_Rem_Tr);
         Retreat (New_Rem_Tr);
         Advance (1, New_Rem_Tr);
         Swap_Rem_Trees (New_Rem_Tr, F.TP);
         Retreat (F.TP);
         Swap_Rem_Trees (F.TP, New_Rem_Tr);
         Retreat (New_Rem_Tr);
         Swap_Rem_Trees(New_Rem_Tr, F.TP);
   end Elevate_Left_Middle;
      ⋮

end BST_Realiz;
```

Figure 57: Operation Elevate Right Middle for balancing

```
Realization BST_Realiz (

     ⋮
     ⋮

   Operation Elevate_Left_Middle(updates F: A_C_Fn);
         requires F.TP.Rem ≠ Ω
            (which_entails F.TP.Rem_Tr: U_Tr(2, IRV_Pair)~{Ω}) and
               Split_at(0, F.TP.Rem_Tr).RT ≠ Ω
            (which_entails
               Split_at(0, F.TP.Rem_Tr).RT: U_Tr(2, IRV_Pair)~{Ω}) and
               Split_at(1, Split_at(0, F.TP.Rem_Tr).RT).RT ≠ Ω
            which_entails
     Split_at(1, Split_at(0, F.TP.Rem_Tr).RT).RT: U_Tr(2, IRV_Pair)~{Ω};

         ensures F.Last_Id = #F.Last_Id and F.TP.Path = #F.TP.Path and
               F.TP.Rem_Tr =
               Jn(⟨Jn(⟨ Split_at(0, Split_at(0, #F.TP.Rem_Tr).RT).RT,
                    Split_at(0, Split_at(1,Split_at(0,#F.TP.Rem_Tr)
                        .RT).RT).RT ⟩,Rt_Lab(Split_at(0, #F.TP.Rem_Tr))),
                  Jn( ⟨ Split_at(1, Split_at(1, Split_at(0,#F.TP.Rem_Tr)
                            .RT).RT).RT, Split_at(1, #F.TP.Rem_Tr).RT ⟩,
                                                Rt_Lab(#F.TP.Rem_Tr))⟩,
                 Rt_Lab(Split_at(1, Split_at(0, #F.TP.Rem_Tr).RT).RT) );

   Procedure

         Var New_Rem_Tr: Tree_Fac.Tree_Posn;
         Advance (1, F.TP);
         Advance (2, F.TP);
         Swap_Rem_Trees (F.TP, New_Rem_Tr);
         Advance (1, New_Rem_Tr);
         Swap_Rem_Trees (New_Rem_Tr, F.TP);
         Retreat (F.TP);
         Swap_Rem_Trees (F.TP, New_Rem_Tr);
         Retreat (New_Rem_Tr);
         Advance (2, New_Rem_Tr);
         Swap_Rem_Trees (New_Rem_Tr, F.TP);
         Retreat (F.TP);
         Swap_Rem_Trees (F.TP, New_Rem_Tr);
         Retreat (New_Rem_Tr);
         Swap_Rem_Trees(New_Rem_Tr, F.TP);

   end Elevate_Left_Middle;

      ⋮

end BST_Realiz;
```

Figure 58: Operation Elevate Left Middle for balancing

```
Operation Adjust (updates F: A_C_Fn)
   requires Is_L_R_Cfml_w(◄, F.TP.Rem_Tr Ψ F.TP.Path ) and
            F.TP.Rem_Tr ≠ Ω
   ensures F.TP.Path Ψ F.TP. Rem_Tr = #F.TP.Path Ψ #F.TP.Rem_Tr and
              F.Last_Id = #F.Last_Id and
                     Is_L_R_Cfml_w(◄, F.TP.Rem_Tr Ψ F.TP.Path ) and
                           Is_Balanced (F.TP.Path Ψ F.TP. Rem_Tr);

Recursive Procedure Adjust (updates F: A_C_Fn);
   decreasing ht(F.TP.Rem_Tr);

   Var balance: Integer
   balance := LT_Height (F) - RT_Height (F);

   If (balance > 1) then
      Advance (1, F.TP);

      If (LT_Height (F) >= RT_Height (F)) then
         Retreat(F.TP);
         Right_Rotate_Rem_Tr (F);
      else
         Retreat (F.TP);
         Elevate_Left_Middle(F);
      end;
   else
      If (balance < −1) then
         Advance (2, F.TP);

         If (RT_Height (F) >= LT_Height (F)) then
            Retreat(F.TP);
            Left_Rotate_Rem_Tr (F);
         else
            Retreat (F.TP);
            Elevate_Right_Middle(F);
         end;
      end;
   end;
   If (Path_Length(F.TP) /= 0) then
      Retreat (F.TP);
      Adjust (F);
   end;
end Adjust;
   ⋮

end BST_Realiz;
```

Figure 59: Implementation of operation Adjust

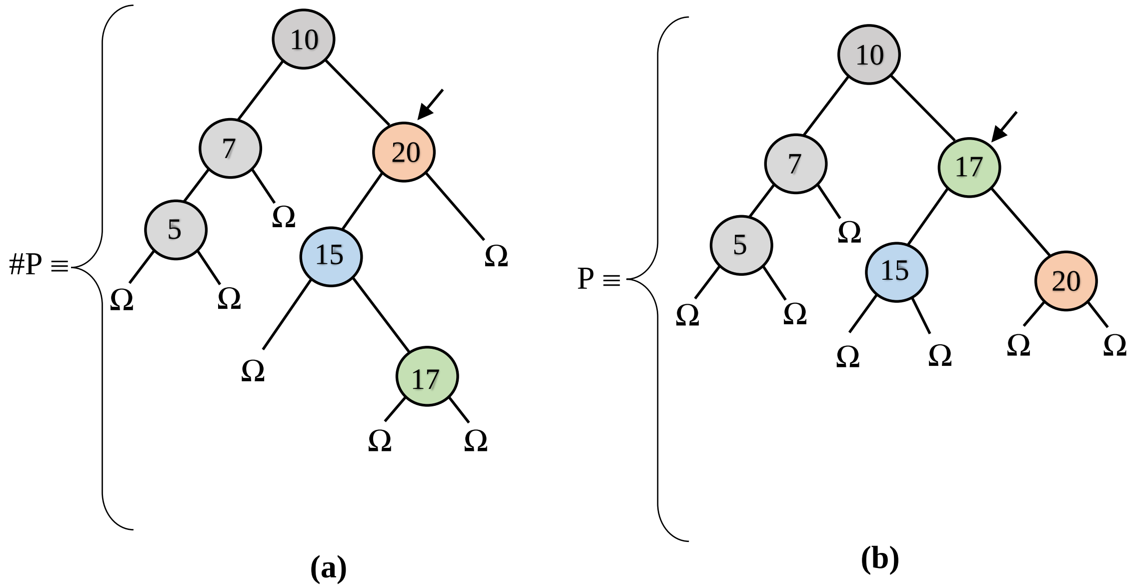


Figure 60: Demonstration on Left-Right Heavy imbalance: (a) Left-Right Heavy Rem_Tr (b) Balanced result after Elevate Left Middle

CHAPTER FOUR

VERIFICATION

As stated earlier, a thesis objective is to present a challenge verification problem of an implementation involving multiple theories and the use of the tree concept which is based on the non-trivial general tree theory for which there are no special-purpose solvers. This chapter presents the work that is in progress concerning verification of the enhancements and map implementation developed under this research.

Generation of Verification Conditions (VCs)

The purpose of this section is merely to illustrate the verification process using the simplest possible example. VCs for the Delete_Remainder enhancement are discussed here. As a part of the verifying compiler, the VC Generator will accept the implementation together with specifications and apply respective proof rules to mechanically form VCs, proving all of which is equivalent to the correctness of the program [6].

For the generation of VCs for Delete_Remainder, a minimal set of the specifications and theories just needed for this enhancement were input and three VCs were generated for correctness of Delete_Remainder. The first one is shown in Figure 61. Each VC has a goal and given(s). In the first VC, the goal is to prove `P'.Path = P.Path` and with the givens it can be observed that the proof is “obvious”. In this case given 1 is sufficient to prove the goal.

```
VC 0_1
Ensures Clause of Delete_Remainder:
Obvious_Deletion_Realiz.rb(4:11)

Goal(s):

(P'.Path = P.Path)

Given(s):

1. (P'.Path = P.Path)
2. (P'.Rem_Tr = Q.Rem_Tr)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (Q'.Path = Q.Path)
5. (Q.Path = Empty_String)
6. (Q.Rem_Tr = Empty_Tree)
```

Figure 61: Fist VC for ensures clause of Delete Remainder

The second VC is shown next in Figure 62. This VC has a goal of `P'.Rem_Tr = Empty_Tree` and it is provable using givens 2 and 6.

```
VC 0_2
Ensures Clause of Delete_Remainder:
Obvious_Deletion_Realiz.rb(4:11)

Goal(s):

(P'.Rem_Tr = Empty_Tree)

Given(s):

1. (P'.Path = P.Path)
2. (P'.Rem_Tr = Q.Rem_Tr)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (Q'.Path = Q.Path)
5. (Q.Path = Empty_String)
6. (Q.Rem_Tr = Empty_Tree)
```

Figure 62: Second VC for ensures clause of Delete Remainder

The third and final VC concerns `Remaining_Cap` and it is shown in Figure 63. The goal and givens are straightforward. As it can be observed for all the VCs generated for this simplest example, correctness can be established by a simple automated prover without deep thinking [9]. While it is difficult to claim this would be the case for all VCs generated for the non-trivial map implementation, that is the opportunity and challenge presented by this thesis. A more detailed output of the VC generation process is shown in Appendix D.

```
VC 0_3

Ensures Clause of Delete_Remainder:
Obvious_Deletion_Realiz.rb(4:11)

Goal(s):

((Remaining_Cap + N_C(Zip_Op(Q'.Path, Q'.Rem_Tr))) =
(Remaining_Cap + N_C(P.Rem_Tr)))

Given(s):

1. (Q'.Path = Q.Path)
2. (P'.Rem_Tr = Q.Rem_Tr)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (P'.Path = P.Path)
5. (Q.Path = Empty_String)
6. (Q.Rem_Tr = Empty_Tree)
```

Figure 63: Third VC for ensures clause of Delete Remainder

CHAPTER FIVE

SUMMARY AND FUTURE DIRECTIONS

The primary goal of this thesis is to present an opportunity and a challenge for automated verification. Using a non-trivial tree theory, exploration tree template and almost constant function concepts, several enhancements to the tree concept and a map implementation based on trees have been developed. The implementation is annotated to make it amenable to verification, in the process illustrating what is necessary for software engineers to learn to develop verified components. While the effort is considerable, once developed and verified, the cost will be amortized over the lifetime uses of the component.

While this thesis has led to different enhancements and implementations that can test the progress we have towards automated verification, it is also the beginning phase of a host of directions that are worthy of exploration and improvement. First and foremost are the improvements that can be made to map implementation which is currently too long because few operation enhancements are currently available for exploration tree. An immediate direction is the creation of suitable enhancements for various tree operations that are currently locally defined within the implementation. This improvement will simplify the code and verification process.

Another future work that can improve this thesis is more mathematical development that would make the assertions simpler for automated systems to manipulate (e.g., avoidance of quantifiers in the few places where they are used).

A direction that is worthy of exploration is the type of balancing mechanism that can be used. This thesis has presented AVL trees which is a worst-case balancing mechanism. But for research and experimentation, efficient implementations based on other ideas such as splay trees (amortized mechanism) and randomly-balanced BSTs (randomized mechanism) can be developed with suitable annotations. Performance annotations of all implementations is another useful direction.

The general tree theory being one of the complex theories presents a challenge in coming up with an effective way to describe it. In this thesis, a lot of work has been done to use illustrations to make these theories useable in classrooms. It may be instructive to teach the concepts presented here at varying levels of formality to various audiences and evaluate their suitability.

APPENDICES

Appendix A

Exploration Tree Template

```
Concept Exploration_Tree_Template( type Node_Label; eval k, Initial_Capacity: Integer );
        uses Std_Integer_Fac, Std_Boolean_Fac, General_Tree_Theory with Relativization_Ext;
        requires 1 ≤ k and 0 < Initial_Capacity which_entails k: ℕ>0 and Initial_Capacity: ℕ;
        Var Remaining_Cap: ℕ;
                initialization
                        ensures Remaining_Cap = Initial_Capacity;

        Family Tree_Posn ⊆ U_Tr_Pos( k, Node_Label );
                exemplar P;
                initialization
                        ensures P.Path = Λ and P.Rem_Tr = Ω;
                finalization
                        ensures Remaining_Cap = #Remaining_Cap + N_C (P.Path Ψ P.Rem_Tr);

        Oper Advance( eval dir: Integer; upd P: Tree_Posn );
                requires P.Rem_Tr ≠ Ω
                which_entails P.Rem_Tr: Tr(Node_Label)~{Ω}and 1 ≤ dir ≤ k;
                ensures P.Rem_Tr = ≸( Prt_btwn(dir ∸ 1, dir, Rt_Brhs(#P.Rem_Tr)) ) and
                P.Path = #P.Path∘⟨( Rt_Lab(#P.Rem_Tr), Prt_btwn(0, dir ∸ 1,
                Rt_Brhs(#P.Rem_Tr)),Prt_btwn(dir, k, Rt_Brhs(#P.Rem_Tr)) )⟩;

        Oper Reset( upd P: Tree_Posn );
                ensures P.Path = Λ and P.Rem_Tr = #P.Path Ψ #P.Rem_Tr;

        Oper At_an_End( rest P: Tree Posn ): Boolean;
                ensures At_an_End = ( P.Rem_Tr = Ω );

        Oper Add_Leaf( alt Labl: Node_Label; upd P: Tree_Posn );
                affects Remaining_Cap;
                requires P.Rem_Tr = Ω and Remaining_Cap > 0;
                ensures P.Path = #P.Path and P.Rem_Tr = Jn( ⟨Ω⟩^k, #Labl ) and
                                                Remaining_Cap = #Remaining_Cap ∸1;

        Oper Remove_Leaf( rpl Leaf_Lab: Node_Label; upd P: Tree_Posn );
                affects Remaining_Cap;
                requires P.Rem_Tr ≠ Ω (which_entails P.Rem_Tr: Tr(Node_Label)~{Ω})
                                                        and Rt_Brhs(P.Rem_Tr) = ⟨Ω⟩^k;
                ensures P.Path = #P.Path and P.Rem_Tr = Ω and Leaf_Lab = Rt_Lab(#P.Rem_Tr)
                                                and Remaining_Cap = #Remaining_Cap + 1;
```

```
Oper At_a_Leaf( rest P: Tree_Posn ): Boolean;
        ensures At_a_Leaf = ((which_entails P.Rem_Tr: Tr(Node_Label)~{Ω})
                                and Rt_Brhs(#P.Rem_Tr)=⟨Ω⟩ ^k);

Oper Swap_Label( upd Labl: Node_Label; upd P: Tree_Posn );
        requires P.Rem_Tr ≠ Ω (which_entails P.Rem_Tr: Tr(Node_Label)~{Ω});
        ensures Labl = Rt_Lab(#P.Rem_Tr) and P.Path = #P.Path and
                P.Rem_Tr = Jn( Rt_Brhs(#P.Rem_Tr), #Labl );

Oper Swap_Rem_Trees( upd P, Q: Tree_Posn );
        ensures P.Path = #P.Path and Q.Path = #Q.Path and P.Rem_Tr = #Q.Rem_Tr and
                Q.Rem_Tr = #P.Rem_Tr;

Oper Swap_w_Rem( upd P, Q: Tree_Posn );
        ensures P.Path = Λ and P.Rem_Tr = #Q.Rem_Tr
                and Q.Path = #Q.Path∘#P.Path and Q.Rem_Tr = #P.Rem_Tr;

Oper Retreat( upd P: Tree_Posn );
        requires P.Path ≠ Λ;
        ensures P.Path = Prt_btwn(0, |#P.Path| ∸ 1, #P.Path) and P.Rem_Tr =(
                Prt_Btwn (|#P.Path| ∸ 1, |#P.Path|, #P.Path) Ψ #P.Rem_Tr;

Oper Path_Length( rest P: Tree_Posn ): Integer;
        ensures Path_Length = |P.Path|;

Oper Rmng_Capacity(): Integer;
        ensures Rmng_Capacity = ( Remaining_Cap );

end Exploration_Tree_Template;
```

Appendix B

Almost Constant Function Template

```
Concept Almost_Constant_Function_Template( type Index, Range_Value;
        def const C: Range_Value; eval Dev_Ct_Max: Integer; def const (i: Index) ⊴ (j: Index): 𝔹 );
                                                  (∗Deviation Count Maximum ∗)
        uses Std_Integer_Fac, Std_Boolean_Fac, Basic_Ordering_Theory;
                requires 1 ≤ Dev_Ct_Max and Is_Total_Ordering(⊴ );

        Family A_C_Fn ⊆ (Index→Range_Value);          (∗ Almost Constant Function ∗)
                exemplar F;
                Def Const Deviation_Count( F: A_C_Fn ): ℕ = ( ‖{ i: Index | F(i) ≠ C }‖ );
                constraint
                        Deviation_Count( F ) ≤ Dev_Ct_Max;
                initialization
                        ensures F = λ i: Index.( C );

        Oper Swap_Value( upd V: Range_Value; upd F: A_C_Fn; rest i: Index );
                requires  Deviation_Count(F) < Dev_Ct_Max or F(i) ≠ C or V = C;
                ensures F(i) = #V and V = #F(i) and ∀ j: Index, if j ≠ i then F(j) = #F(j);

        Def Const (i: Index) ⊲ (j: Index): 𝔹 = ( i ⊴ j and i ≠ j );

        Def Const Are_Devs_after( i: Index, F: A_C_Fn ): 𝔹 = ( ∃ k: Index ∋ i ⊲ k and F(k) ≠ C );
                (∗Are Deviations after ∗)

        Def Const Is_1st_Dev_after( i, k: Index, F: A_C_Fn ): 𝔹 = ( i ⊲ k and F(k) ≠ C and
         (∗ Is 1st Deviation after ∗)                    ∀ j: Index, if i ⊲ j ⊲ k, then F(j) = C );

        Def Const Is_1st_Dev( k: Index, F: A_C_Fn ): 𝔹 = ( F(k) ≠ C and ∀ j: Index, if j ⊲ k,
        (∗ Is 1st Deviation ∗)                                                  then F(j) = C );

        Oper First_Int_Index( rpl i: Index; rest F: A_C_Fn );          (∗ First Interesting Index ∗)
                requires 1 ≤ Deviation_Count (F);
                ensures Is_1st_Dev( i, F );
```

```
        Oper Next_Int_Index( rest i: Index; rest F: A_C_Fn; rpl r: Index );
                                                    (* Next Interesting Index *)
                requires Are_Devs_after( i, F );
                ensures Is_1st_Dev_after( i, r, F);

        Oper Would_Be_Last( rest i: Index; rest F: A_C_Fn ): Boolean;
                ensures Would_Be_Last = ( ¬ Are_Devs_after( i, F ) );

        Oper Max_Deviation_Ct(): Integer;           (* Maximum Deviation Count *)
                ensures Max_Deviation_Ct = ( Dev_Ct_Max );

        Oper Deviation_Count_of( rest F: A_C_Fn ): Integer;
                ensures Deviation_Count_of = ( Deviation_Count(F) );

        Oper Make_Constant( clr F: A_C_Fn );

end Almost_Constant_Function_Template;
```

Appendix C

Map Implementation

```
Realization BST_Realiz (                                    (∗ Binary Search Tree ∗)
                        Operation In_Order (restores i, j: Index): Boolean;
                            ensures In_Order = ( i ⊴ j );
                        Operation Replica(restores i: Index): Index;
                            ensures Replica = ( i );
                        Operation New_Dflt_RV(): Range_Value;
                            ensures New_Dflt_RV = ( C ); (∗ New Default Range Value ∗)
                        Operation Is_Dflt_RV(V:Range_Value): Boolean;
                            ensures Is_Dflt_RV = ( V = C ); (∗ Is Default Range Value ∗)
                                          ) for Almost_Constant_Function_Template;
          uses Exploration_Tree_Template;

   Operation Are_Equal(restores i, j: Index): Boolean;
      ensures Are_Equal = ( i = j );
     procedure
      Are_Equal := In_Order(i, j) and In_Order(j, i);
   end Are_Equal;

   Operation Precedes(restores i, j: Index): Boolean;
      ensures Precedes = ( i ⊲  j );
     procedure
      Precedes := In_Order(i, j) and not In_Order(j, i);
   end Precedes;

  Type IRV_Pair = Record                                  (∗ Index Range Value Pair ∗)
               id : Index;
               V: Range_Value;
                  end;

  Facility Tree_Fac is Exploration_Tree_Template (IRV_Pair, 2, Dev_Ct_Max)
                          realized by Obv_Exploration_Tree_Realiz
                         enhanced by Node_Count_Capability
                          realized by Obv_Node_Count_Realiz
                         enhanced by Deletion_Capability
                          realized by Obvious_Deletion_Realiz
                         enhanced by Node_Height
                          realized by Obv_Node_Height_Capability_Realiz;

   Definition Is_Dflt_C_Free ( T: Tr(IRV_Pair) ): 𝔹 = ( ∀ p: Occ_Set( T.Path Ψ T.Rem_Tr ),
                              (∗ Is Default Constant Free ∗)
   Definition Is_Antitransitive( ρ: (D: Set)⊠D→𝔹 ) = ( ∀ x, y, z: D, if ¬ x ρ y and ¬ y ρ z,
                                                                        then ¬ x ρ z );
```

```
Definition (p: IRV_Pair) ◄ (q: IRV_Pair): 𝔹 = ( p.id ⊲ q.id );             (∗ Is Pair Less Than ∗)
    Corollary 1: Is_Transitive(◄) and Is_Asymmetric(◄) and Is_Antitransitive(◄);

Def. Fn_Sub_Gr: ℘(℘(IRV_Pair)) = { S: ℘(IRV_Pair) | ∀ p, q: S, if p.id = q.id,
        (∗ Function SubGraph ∗)                                        then p.V = q.V };
    Corollary 1: ∀ T: U_Tr_Pos(2, IRV_Pair), if Is_L_R_Cfml_w (◄, T ),
                                                          then Occ_Set(T): Fn_Sub_Gr;

Def. Dom_Set( S: ℘(IRV_Pair) ): ℘(Index) = { i: Index | ∃ p: S ∋ i = p.id };  (∗ Domain Set ∗)
    Corollary 1: ∀ S: Fn_Sub_Gr, ∀ i: Dom_Set(S), ∃! p: S ∋ i = p.id;
    Corollary 2: ∀ S: Fn_Sub_Gr, ∃! F: Index→Range_Value ∋
                          ∀ p: S, F(p.id) = p.V and ∀ i: (Index~Dom_Set(S)), F(i) = C;

Implicit Def. Rpd_Fn( S: Fn_Sub_Gr ): Index→Range_Value  is
    ∀ p: S, Rpd_Fn(S)(p.id) = p.V and ∀ i: (Index~Dom_Set(S)), Rpd_Fn(S)(i) = C;
                                                         (∗ Represented Function ∗)

Type A_C_Fn = Record
                    TP: Tree_Fac.Tree_Posn;  (∗ Tree Position ∗)
                    Last_Id : Index;         (∗ Last Index ∗)
                 end;
    convention Is_L_R_Cfml_w(◄, F.TP.Path Ψ F.TP.Rem_Tr )
               which_entails
               Occ_Set( F.TP.Path Ψ F.TP.Rem Tr ): Fn_Sub_Gr and
               Is_Balanced (F.TP) and Is_Dflt_C_Free (F.TP) and
               ∀ p: Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr), p.id ⊴ F.Last_Id and
                   if Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) ≠ Ω,
                     then ∃ q: Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) ∋  q.id = F.Last_Id;
    correspondence Conc.F = Rpd_Fn( Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) );

Operation Current_Id(restores F: A_C_Fn ): Index;                  (∗ Current Index ∗)
    requires F.TP.Rem_Tr ≠ Ω;
    ensures Current_Id = ( Rt_Lab (F.TP.Rem_Tr).id );
 procedure
    Var P: IRV_Pair;
    Swap_Label (P, F.TP);
    Current_Id := Replica (P.id );
    Swap_Label (P, F.TP);
end Current_Id;
```

```
Operation Shift_to_Index_in_Rem_of( updates F: A_C_Fn; restores i: Index;
                                                        replaces Is_Present: Boolean );
        requires Is_L_R_Cfml_w( ◄, F.TP.Rem_Tr );
        ensures F.TP.Path Ψ F.TP.Rem_Tr = #F.TP.Path Ψ #F.TP.Rem_Tr and
            #F.TP.Path Is_Prefix F.TP.Path and F.Last_Id = #F.Last_Id and
            if i ∈ Dom_Set( Occ_Set(#F.TP.Rem_Tr) ), then Is_Present and
                F.TP.Rem_Tr ≠ Ω ( which_entails F.TP.Rem_Tr: (Tr(IRV_Pair)~{Ω}) ) and
                Rt_Lab(F.TP.Rem_Tr).id = i and
            if i ∉ Dom_Set( Occ_Set(#F.TP.Rem_Tr) ),
                then ¬ Is_Present and F.TP.Rem_Tr = Ω and
            Is_L_R_Cfml_w( ◄, prt_btwn(|#F.TP.Path|, |F.TP.Path|, F.TP.Path) Ψ Jn(⟨Ω⟩², (i, C)) );

    recursive procedure
        decreasing ht(F.TP.Rem_Tr);

        If (Are_Equal(i, Current_Id(F))) then
        Is_Present := True();
         else
            If (not At_an_End(F.TP)) then
                If (Precedes(i, Current_Id(F)) then
                   Advance (1, F.TP);
                 else
                   Advance (2, F.TP);
                end;
                Shift_to_Index_in_Rem_of(F, i, Is_Present);
             else
                Is_Present := False();
            end;
        end;
    end Shift_to_Index_in_Rem_of;

Operation Shift_to_Index ( updates F: A_C_Fn; restores i: Index;
                                                        replaces Is_Present: Boolean );
    requires Is_L_R_Cfml_w( ◄, F.TP.Rem_Tr );
    ensures F.TP.Path Ψ F.TP.Rem_Tr = #F.TP.Path Ψ #F.TP.Rem_Tr and
            #F.TP.Path Is_Prefix F.TP.Path and F.Last_Id = #F.Last_Id and
            if i ∈ Dom_Set( Occ_Set(#F.TP.Rem_Tr) ), then Is_Present and
                F.TP.Rem_Tr ≠ Ω ( which_entails F.TP.Rem_Tr: (Tr(IRV_Pair)~{Ω}) ) and
                Rt_Lab(F.TP.Rem_Tr).id = i and
            if i ∉ Dom_Set( Occ_Set(#F.TP.Rem_Tr) ),
                then ¬ Is_Present and F.TP.Rem_Tr = Ω and
            Is_L_R_Cfml_w( ◄, prt_btwn(|#F.TP.Path|, |F.TP.Path|, F.TP.Path) Ψ Jn(⟨Ω⟩², (i, C)) );
```

```
procedure Shift_to_Index ( updates F: A_C_Fn; restores i: Index;
                                                            replaces Is_Present: Boolean );

    If (Path_Length(F.TP) ≥ 1 and Precedes(i, Current_Id(F)) then
        Reset(F.TP);
    end;
    Shift_to_Index_in_Rem_of (F, i, Is_Present);

end Shift_to_Index;


Operation Shift_to_First (updates F: A_C_Fn )
   requires F.TP.Path = Λ and F.TP.Rem_Tr ≠ Ω;
   ensures F.TP.Path Ψ F.TP.Rem_Tr = #F.TP.Path Ψ #F.TP.Rem_Tr and
                 F.TP.Rem_Tr ≠ Ω (which_entails F.TP.Rem_Tr: (Tr(IRV_Pair)~{Ω}) and
                 (Rt_Lab( F.TP.Rem_Tr )).id = i and
                         Is_1st_Dev ( Rt_Lab( F.TP.Rem_Tr ).id, F.TP );


Recursive Procedure Shift_to_First ( updates F:A_C_Fn );
    decreasing ht(F.TP.Rem_Tr );

    If (At_an_End (F.TP)) then
        Retreat (F.TP);
    else
        Advance (1, F.TP);
        Shift_to_First (F);
    end;
end Shift_to_First;


Operation Right_Rotate_Rem_Tr(updates F: A_C_Fn);
    requires F.TP.Rem_Tr ≠ Ω (which_entails F.TP.Rem_Tr: U_Tr(2, IRV_Pair)~{Ω})
                    and Split_at(0, F.TP.Rem_Tr).RT ≠ Ω
                          (which_entails Split_at(0, F.TP.Rem_Tr).RT: U_Tr(2, IRV_Pair)~{Ω});
    ensures F.Last_Id = #F.Last_Id and F.TP.Path = #F.TP.Path and F.TP.Rem_Tr =
            Jn( 〈 Split_at(0, Split_at(0, #F.TP.Rem_Tr).RT).RT,
                  Jn(〈Split_at(1, Split_at(0, #F.TP.Rem_Tr).RT).RT,
                                  Split_at(1, #F.TP.Rem_Tr).RT〉, Rt_Lab(#F.TP.Rem_Tr)) 〉,
                                                     Rt_Lab(Split_at(0, #F.TP.Rem_Tr).RT) );
```

```
    procedure
        Var New_Rem_Tr: Tree_Fac.Tree_Posn;
        Advance (1, F.TP);
        Swap_Rem_Trees (New_Rem_Tr, F.TP);
        Advance (2, New_Rem_Tr);
        Swap_Rem_Trees (New_Rem_Tr, F.TP);
        Retreat (F.TP);
        Swap_Rem_Trees(New_Rem_Tr, F.TP);
        Retreat (New_Rem_Tr);
        Swap_Rem_Trees(New_Rem_Tr, F.TP);
    end Right_Rotate_Rem_Tr ;

Operation Left_Rotate_Rem_Tr (updates F : A_C_Fn);
    requires F.TP.Rem ≠ Ω (which_entails F.TP.Rem: U_Tr(2, IRV_Pair)~{Ω}) and
            Split_at(1, F.TP.Rem_Tr).RT ≠ Ω
                        (which_entails Split_at(1, F.TP.Rem_Tr).RT: U_Tr(2, IRV_Pair)~{Ω});
    ensures F.Last_Id = #F.Last_Id and F.TP.Path = #F.TP.Path and F.TP.Rem_Tr =
            Jn(⟨Jn(⟨Split_at(0, #F.TP.Rem_Tr ).RT,
                    Split_at(0, Split_at(1, #F.TP.Rem_Tr).RT).RT⟩, Rt_Lab(#F.TP.Rem_Tr)),
                                    Split_at(1, Split_at(1, #F.TP.Rem_Tr).RT).RT⟩,
                                        Rt_Lab(Split_at(1, #F.TP.Rem_Tr).RT));

procedure

    Var New_Rem_Tr: Tree_Fac.Tree_Posn;
    Advance (2, F.TP);
    Swap_Rem_Trees (New_Rem_Tr, F.TP);
    Advance (1, New_Rem_Tr);
    Swap_Rem_Trees (New_Rem_Tr, F.TP);
    Retreat (F.TP);
    Swap_Rem_Trees (New_Rem_Tr, F.TP);
    Retreat (New_Rem_Tr);
    Swap_Rem_Trees (New_Rem_Tr, F.TP);

end Left_Rotate_Rem_Tr;
```

```
Operation Elevate_Left_Middle(updates F: A_C_Fn);
    requires F.TP.Rem ≠ Ω
            (which_entails F.TP.Rem_Tr: U_Tr(2, IRV_Pair)~{Ω}) and
               Split_at(0, F.TP.Rem_Tr).RT ≠ Ω
            (which_entails Split_at(0, F.TP.Rem_Tr).RT: U_Tr(2, IRV_Pair)~{Ω}) and
               Split_at(1, Split_at(0, F.TP.Rem_Tr).RT).RT ≠ Ω which_entails
               Split_at(1, Split_at(0, F.TP.Rem_Tr).RT).RT: U_Tr(2, IRV_Pair)~{Ω};

    ensures F.Last_Id = #F.Last_Id and F.TP.Path = #F.TP.Path and F.TP.Rem_Tr =
        Jn( 〈 Jn( 〈 Split_at(0, Split_at(0, #F.TP.Rem_Tr).RT).RT,
                  Split_at(0, Split_at(1, Split_at(0, #F.TP.Rem_Tr).RT).RT).RT 〉,
                                   Rt_Lab(Split_at(0, #F.TP.Rem_Tr)) ),
            Jn( 〈 Split_at(1, Split_at(1, Split_at(0, #F.TP.Rem_Tr).RT).RT).RT,
                  Split_at(1, #F.TP.Rem_Tr).RT 〉, Rt_Lab(#F.TP.Rem_Tr) ) 〉,
                         Rt_Lab(Split_at(1, Split_at(0, #F.TP.Rem_Tr).RT).RT) );
  procedure
    Var New_Rem_Tr: Tree_Fac.Tree_Posn;
    Advance (1, F.TP);
    Advance (2, F.TP);
    Swap_Rem_Trees (F.TP, New_Rem_Tr);
    Advance (1, New_Rem_Tr);
    Swap_Rem_Trees (New_Rem_Tr, F.TP);
    Retreat (F.TP);
    Swap_Rem_Trees (F.TP, New_Rem_Tr);
    Retreat (New_Rem_Tr);
    Advance (2, New_Rem_Tr);
    Swap_Rem_Trees (New_Rem_Tr, F.TP);
    Retreat (F.TP);
    Swap_Rem_Trees (F.TP, New_Rem_Tr);
    Retreat (New_Rem_Tr);
    Swap_Rem_Trees(New_Rem_Tr, F.TP);
end Elevate_Left_Middle;
```

```
Operation Elevate_Right_Middle(updates F: A_C_Fn);
    requires F.TP.Rem ≠ Ω (which_entails F.TP.Rem_Tr: U_Tr(2, IRV_Pair)~{Ω}) and
            Split_at(1, F.TP.Rem_Tr).RT ≠ Ω
                (which_entails Split_at(1, F.TP.Rem_Tr).RT: U_Tr(2, IRV_Pair)~{Ω}) and
            Split_at(0, Split_at(1, F.TP.Rem_Tr).RT).RT ≠ Ω which_entails
            Split_at(0, Split_at(1, F.TP.Rem_Tr).RT).RT: U_Tr(2, IRV_Pair)~{Ω};
    ensures F.Last_Id = #F.Last_Id and F.TP.Path = #F.TP.Path and F.TP.Rem_Tr =
            Jn( 〈 Jn( 〈 Split_at(0, #F.TP.Rem).RT),
                Split_at(1, Split_at(0, Split_at(0, #F.TP.Rem).RT).RT).RT 〉,
                                                    Rt_Lab(#F.TP.Rem_Tr) ),
                  Jn( 〈 Split_at(1, Split_at(0, Split_at(1, #F.TP.Rem_Tr).RT).RT).RT,
                                         Split_at(1, (Split_at(1, #F.TP.Rem_Tr).RT).RT 〉,
                                                    Rt_Lab(Split_at(1, #F.TP.Rem_Tr).RT ) 〉,
                                         Rt_Lab(Split_at(0, Split_at(1, #F.TP.Rem_Tr).RT).RT) );
     procedure
        Var New_Rem_Tr: Tree_Fac.Tree_Posn;
        Advance (2, F.TP);
        Advance (1, F.TP);
        Swap_Rem_Trees (F.TP, New_Rem_Tr);
        Advance (2, New_Rem_Tr);
        Swap_Rem_Trees (New_Rem_Tr, F.TP);
        Retreat (F.TP);
        Swap_Rem_Trees (F.TP, New_Rem_Tr);
        Retreat (New_Rem_Tr);
        Advance (1, New_Rem_Tr);
        Swap_Rem_Trees (New_Rem_Tr, F.TP);
        Retreat (F.TP);
        Swap_Rem_Trees (F.TP, New_Rem_Tr);
        Retreat (New_Rem_Tr);
        Swap_Rem_Trees(New_Rem_Tr, F.TP);
    end Elevate_Left_Middle;
```

```
Operation LT_Height (restores F: A_C_Fn): Integer
    ensures LT_Height = ht (Split_at(0, F.TP.Rem_Tr).RT);
procedure

    If(At_an_end(F.TP)) then
        LT_Height := 0;
    else
        Advance (1, F.TP);
        LT_Height := Node_Height (F.TP);
        Retreat (F.TP)
    end;
end;

Operation RT_Height (restores F: A_C_Fn): Integer
    ensures RT_Height = ht(Split_at(1, F.TP.Rem_Tr).RT);
procedure

    If(At_an_end(F.TP)) then
        RT_Height := 0;
    else
        Advance (2, F.TP);
        RT_Height := Node_Height (F.TP);
        Retreat (F.TP);
    end;
end;

Operation Adjust (updates F: A_C_Fn)
    requires Is_L_R_Cfml_w(◄, F.TP.Rem_Tr Ψ F.TP.Path ) and F.TP.Rem_Tr ≠ Ω;
    ensures F.TP.Path Ψ F.TP.Rem_Tr = #F.TP.Path Ψ #F.TP. Rem_Tr and
            F.Last_Id = #F.Last_Id and
                Is_L_R_Cfml_w(◄, F.TP.Rem_Tr Ψ F.TP.Path ) and
                    Is_Balanced (F.TP.Path Ψ F.TP. Rem_Tr);

Recursive Procedure Adjust (updates F: A_C_Fn);
    decreasing ht(F.TP.Rem_Tr);

    Var balance: Integer

    balance := LT_Height (F) – RT_Height (F);


    If (balance > 1) then
        Advance (1, F.TP);

        If (LT_Height (F) >= RT_Height (F)) then
            Retreat(F.TP);
```

```
                Right_Rotate (F);
            else
                Retreat (F.TP);
                Elevate_Left_Middle(F);
            end;
        else
            If (balance < − 1) then
                Advance (2, F.TP);

                If (RT_Height (F) >= LT_Height (F)) then
                    Retreat(F.TP);
                    Left_Rotate (F);
                else
                    Retreat (F.TP);
                    Elevate_Right_Middle(F);
                end;
            end;
        end;
        If (Path_Length(F.TP) /= 0) then
            Retreat (F.TP);
            Adjust (F);
        end;
    end Adjust;

Operation Delete_Rt_Node ( updates F: A_C_Fn);
    affects Remaining_Cap;
    requires F.TP.Rem_Tr ≠ Ω)
                        Is_L_R_Cfml_w(◄, F.TP.Rem_Tr Ψ F.TP.Path ) and
                                    Is_Balanced (F.TP.Path Ψ F.TP. Rem_Tr);
    ensures F.TP.Path = #F.TP.Path and
        Occ_Set (F.TP. Rem_Tr) = Occ_Set (#F.TP.Rem_Tr) ~ {Rt_Lab(#F.TP.Rem_Tr)} and
        Is_L_R_Cfml_w (◄, F.TP.Rem_Tr Ψ F.TP.Path ) and
          Remaining_Cap = #Remaining_Cap +1 and If (F.TP.Rem_Tr /= Ω) then
            0 ≤ |ht(Split_at(0, F.TP.Rem_Tr).RT) − ht(Split_at(1, F.TP.Rem_Tr).RT)| ≤ 2 and
                ∀ p: Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr), p.id ⊴ F.Last_Id and
                 if Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) ≠ Ω,
                        then ∃ q: Occ_Set(F.TP.Path Ψ F.TP.Rem_Tr) ∍ q.id = F.Last_Id;

Procedure Delete_Rt_Node (updates F: A_C_Fn);
    Var L, R : A_C_Fn;
    Var P : IRV_Pair;
    Var Is_Last_Id: Boolean;
    Is_Last_Id := False();

    If (Are_Equal((Rt_Lab(F.TP.Rem_Tr)).id, F.Last_Id)) then
        Is_Last_Id := True();
    end;
```

```
If (At_a_Leaf(F)) then
    Remove_Leaf(P, F.TP);
else
    Advance(1, F.TP);
    If(At_an_End(F.TP)) then
        Retreat(F.TP);
        Advance(2, F.TP);
        Swap_Rem_Trees(R.TP, F.TP);
        Retreat(F.TP);
        Remove_Leaf(P, F.TP);
        Swap_Rem_Trees(R.TP, F.TP);
    else
        Retreat(F.TP);
        Advance(2, F.TP);
        If(At_an_End(F.TP)) then
            Retreat(F.TP);
            Advance(1, F.TP);
            Swap_Rem_Trees(L.TP, F.TP);
            Retreat(F.TP);
            Remove_Leaf(P, F.TP);
            Swap_Rem_Trees(L.TP, F.TP);
        else
            Retreat(F.TP);
            Advance(1, F.TP);
            Swap_Rem_Trees(L.TP, F.TP);
            Retreat (F.TP);
            Advance(2, F.TP);
            Swap_Rem_Trees(R.TP, F.TP);
            Retreat(F.TP);
            Remove_Leaf(P, F.TP);
            Shift_to_First(R);
            Swap_Rem_Trees(R.TP, F.TP);
            Reset (R.TP);
            Advance (1, F.TP);
            Swap_Rem_Trees(L.TP, F.TP)
            Retreat (F.TP);
            Advance(2. F.TP);
            If (At_an_End(F.TP)) then
                Swap_Rem_Trees(R.TP, F.TP);
            else
                Advance(2, F.TP);
                Swap_Rem_Trees(R.TP, F.TP);
                Retreat(F.TP);
            end;
            Retreat(F.TP);
        end;
    end;
```

```
    If (Is_Last_Id and At_an_End(F)) then
            Retreat(F.TP);
            F.Last_Id := Current_Id(F);
            Advance(2, F.TP);
        else
            If (Is_Last_Id) then
                F.Last_Id := Current_Id(F);
            end;
        end;
    end;
end Delete_Rt_Node;

    Procedure Swap_Value(updates V: Range_Value; updates F: A_C_Fn; restores i: Index);
        Var P: IRV_Pair;
        Var present: Boolean;

        P.id := Replica( i );

        Shift_to_Index ( i, F, present );

        If present then
            If not Is_Dflt_RV( V ) then
                P.V :=: V;
                Swap_Label( P, F.TP );
                V :=: P.V;
            else
                Delete_Rt_Node(F);
                V :=: P.V;
                Adjust(F);
            end;
        else
            If not Is_Dflt_RV( V ) then
                P.V :=: V;
                If (Node_Count(F.TP) = 0) then
                    F.Last_Id := Replica(P.id);
                else
                    If (not In_Order (F.Last_Id, P.id) then
                        F.Last_Id := Replica(P.id);
                    end;
                end;
                Add_Leaf ( P, F.TP );
                V := New_Dflt_RV ();
                Adjust(F);
            end;
        end;
    end Swap_Value;
```

```
Procedure First_Int_Index ( replaces i: Index; restores F: A_C_Fn );

    Reset(F.TP )
    Shift_to_First( F );
    i := Current_Id( F );

end First_Int_Index;

Procedure Next_Int_Index (restores i: Index; restores F: A_C_Fn; replace r: Index );
    Var P: IRV_Pair;
    Var present: Boolean;

    Shift_to_Index (i, F, present);

    Advance (2, F.TP);
    If (At_an_End (F.TP)) then
        Retreat(F.TP);
        While (Precedes (Current_Id(F), i) or Are_Equal(Current_Id(F), i))
            maintaining F.Path Ψ F.Rem_Tr =
            ((Prt_btwn(0, |#F.Path| ∸ 1, #F.Path)) o
                            Prt_Btwn (|#F.Path| ∸ 1, |#F.Path|, #F.Path) ) Ψ #F.Rem_Tr ;
            decreasing | F.TP.Path |;
        do
            Retreat(F.TP);
        end;
        r := Current_Id(F)
    else
        Advance(1, F.TP);
        If (At_an_End (F.TP)) then
            Retreat (F.TP);
            r := Current_Id(F)
        else
            Shift_To_First(F);
            r := Current_Id(F);
        end;
    end;
end Next_Int_Index;

Procedure Would_Be_Last (restores i: Index; restores F: A_C_Fn): Boolean;

    If ( Are_Equal (F.Last_Id , i) ) then
        Would_Be_Last := True();
    else
        Would_Be_Last := False();
    end;

end Would_Be_Last;
```

```
    Procedure Max_Deviation_Ct(): Integer;

        Max_Deviation_Ct := Dev_Ct_Max;

    end Max_Deviation_Ct;

    Procedure Deviation_Count_of ( restores F: A_C_Fn ): Integer;

        Deviation_Count_of := Node_Count ( F.TP );

    end Deviation_Count_of;

    Procedure Make_Constant ( clears F: A_C_Fn );

        Reset( F.TP );
        Delete_Remainder( F.TP );

    end Make_Constant;

end BST_Realiz;
```

Appendix D

VC Generation for Delete Remainder

```
VCs for Obvious_Deletion_Realiz.rb generated Tue Apr 11 13:50:55 EDT 2017

================================ VC(s): ================================

VC 0_1

Ensures Clause of Delete_Remainder: Obvious_Deletion_Realiz.rb(4:11)

Goal(s):

(P'.Path = P.Path)

Given(s):

1. (P'.Path = P.Path)
2. (P'.Rem_Tr = Q.Rem_Tr)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (Q'.Path = Q.Path)
5. (Q.Path = Empty_String)
6. (Q.Rem_Tr = Empty_Tree)

VC 0_2

Ensures Clause of Delete_Remainder: Obvious_Deletion_Realiz.rb(4:11)

Goal(s):

(P'.Rem_Tr = Empty_Tree)

Given(s):

1. (P'.Path = P.Path)
2. (P'.Rem_Tr = Q.Rem_Tr)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (Q'.Path = Q.Path)
5. (Q.Path = Empty_String)
6. (Q.Rem_Tr = Empty_Tree)
```

VC 0_3

Ensures Clause of Delete_Remainder: Obvious_Deletion_Realiz.rb(4:11)

Goal(s):

((Remaining_Cap + N_C(Zip_Op(Q'.Path, Q'.Rem_Tr))) = (Remaining_Cap + N_C(P.Rem_Tr)))

Given(s):

1. (Q'.Path = Q.Path)
2. (P'.Rem_Tr = Q.Rem_Tr)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (P'.Path = P.Path)
5. (Q.Path = Empty_String)
6. (Q.Rem_Tr = Empty_Tree)

========================VC Generation Details ==========================

Enhancement Realization Name: Obvious_Deletion_Realiz
Enhancement Name: Deletion_Capability
Concept Name: Exploration_Tree_Template

==

Appendix E

General Tree Theory Developed by Dr. Bill Ogden

Note: Because of the size of this theory, only few sections referred in thesis are included.

Precis General_Tree_Theory;
uses General_String_Theory **with** Relativization_Ext, Basic_Multiset_Theory;

Definition Is_Tree_Former(Tr: $\mathcal{Cls}$, Ω: Tr, Jn: Str(Tr)×$\mathbb{El}$→(Tr ~ {Ω})): $\mathcal{B}$ = (
Pty 1: ∀ α, β: Str(Tr), ∀ x, y: $\mathbb{El}$, **if** Jn(α, x) = Jn(β, y), **then** α = β **and** x = y;
Pty 2: ∀ C: $\wp$(Tr),
if (i) Ω ∈ C **and**
(ii) ∀ α: Str(C), ∀ x: $\mathbb{El}$, Jn(α, x) ∈ C,
then C = Tr
);
(∗ Tree, Empty Tree, Join, Is_Tree_Former: ∗)

Corollary 1: ∀ Tr: $\mathcal{Cls}$, ∀ Ω: Tr, ∀ Jn: Str(Tr)×$\mathbb{El}$→(Tr ~ {Ω}),
if Is_Tree_Former(Tr, Ω, Jn), **then** ∀ U, V: $\mathcal{Cls}$, ∀ p: U×Str(Tr)×$\mathbb{El}$→U,
∀ b: U→V, ∀ s: U×Str(V)×Str(Tr)×$\mathbb{El}$→V, ∃! f: U×Tr→V ∍ ∀ α: Str(Tr),
∀ u: U, ∀ x: $\mathbb{El}$, f(u, Ω) = b(u) **and** f(u, Jn(α, x)) = s(u, f[p(u, α, x), [α]], α, x);
(∗ Inductive definability, permutation, basis, successor, function∗)

Corollary 2: ∀ Tr_1, Tr_2: $\mathcal{Cls}$, ∀ Ω_1: Tr_1, ∀ Ω_2: Tr_2, ∀ Jn_1: Str(Tr_1)×$\mathbb{El}$→(Tr_1 ~ {Ω_1}),
∀ Jn_2: Str(Tr_2)×$\mathbb{El}$→$_D$(Tr_2~ {Ω_2}), **if** Is_Tree_Former(Tr_1, Ω_1, Jn_1) **and**
Is_Tree_Former(Tr_2, Ω_2, Jn_2), **then** ∃! h: Tr_1→Tr_2 ∍ h(Ω_1) = Ω_2 **and**
∀ α: Str(Tr_1), ∀ x: $\mathbb{El}$, h(Jn_1(α, x)) = Jn_2(h(α), x) **and** Is_Bijective(h);
(∗ Isomorphism of instances ∗)

Corollary 3: ∃ Tr: $\mathbb{Cls}$, ∃ Ω: Tr, ∃ Jn: Str(Tr)×$\mathbb{El}$→(Tr ~ {Ω}) ∍
Is_Tree_Former(Tr, Ω, Jn);
(∗ Satisfiability ∗)

Categorical Definition for ($\mathbb{Tr}$: $\mathcal{Cls}$, Ω: $\mathbb{Tr}$, Jn: Str($\mathbb{Tr}$)×$\mathbb{El}$→($\mathbb{Tr}$ ~ {Ω})) **is**
Is_Tree_Former($\mathbb{Tr}$, Ω, Jn);
Corollary 1: Is_Surjective(Jn);

Implicit Definition Indcd_Fn(U, V: $\mathcal{Cls}$, b: U→V, s: U×Str(V)×Str($\mathbb{Tr}$)×$\mathbb{El}$→V,
p: U×Str($\mathbb{Tr}$)×$\mathbb{El}$→U): U×$\mathbb{Tr}$→V **is**
∀ u: U, Indcd_Fn(U, V, b, s, p) (u, Ω) = b(u) **and** ∀ α: Str($\mathbb{Tr}$), ∀ x: $\mathbb{El}$,
Indcd_Fn(U, V, b, s, p)(u, Jn(α, x)) = s(u, Indcd_Fn(U, V, b, s, p)[p(u, α, x), [α]], α, x);

Inductive Def. on T: $\mathbb{Tr}$ **of** N_C(T): ℕ **is** (∗ Node Count ∗)
(i) N_C(Ω) = 0;
(ii) N_C (Jn(α, x)) = suc($Ag_{(+, 0)}$(N_C[[α]]));
Corollary 1: ∀ T: $\mathbb{Tr}$, N_C(T) = 0 **iff** T = Ω;

Inductive Def. on T: $\mathbb{Tr}$ **of** ht(T): $\mathbb{N}$ **is** (∗ <u>height</u> ∗)

(i) $ht(\Omega) = 0$;

(ii) $ht(Jn(\alpha, x)) = suc(Ag_{(Max, 0)}(ht[[\alpha]]))$;

Corollary 1: $\forall$ T: $\mathbb{Tr}$, ht(T) = 0 **iff** T = Ω;

Corollary 2: $\forall$ T: $\mathbb{Tr}$, ht(T) ≤ N_C(T);

Def. Is_Leaf(T: $\mathbb{Tr}$): $\mathbb{B}$ = (∃ x: $\mathbb{El}$, ∃ α: Str({Ω}) ∍ T = Jn(α, x));

Corollary 1: $\forall$ T: $\mathbb{Tr}$, **if** Is_Leaf(T), **then** N_C(T) = ht(T) = 1;

Inductive Def. on T: $\mathbb{Tr}$ **of** Occ_Set(T: $\mathbb{Tr}$): $\mathbb{Set}$ **is** (∗ <u>Occ</u>urrence <u>Set</u> ∗)

(i) Occ_Set(Ω) = ∅;

(ii) $Occ\_Set(Jn(\alpha, x)) = Ag_{(\cup,\emptyset)}(Occ\_Set[[\alpha]]) \cup \{x\}$;

Corollary 1: $\forall$ T: $\mathbb{Tr}$, ||Occ_Set(T)||: $\mathbb{N}$;

Corollary 2: $\forall$ T: $\mathbb{Tr}$, ||Occ_Set(T)|| ≤ N_C(T);

Inductive Def. on T: $\mathbb{Tr}$ **of** $(T)^{TRev}$: Tr(Γ) **is** (∗ <u>T</u>ree <u>Rev</u>ersal ∗)

(i) $\Omega^{TRev} = \Omega$;

(ii) $Jn(\alpha, x)^{TRev} = Jn(([[\alpha]]^{TRev})^{Rev}, x)$;

Corollary 1: $\forall$ T: $\mathbb{Tr}$, $(T^{TRev})^{TRev} = T$;

Corollary 2: $\forall$ T: $\mathbb{Tr}$, $N\_C(T^{TRev}) = N\_C(T)$;

Corollary 3: $\forall$ T: $\mathbb{Tr}$, $ht(T^{TRev}) = ht(T)$;

Corollary 4: $\forall$ T: $\mathbb{Tr}$, $L\_C(T^{TRev}) = L\_C(T)$;

Corollary 5: $\forall$ T: $\mathbb{Tr}$, $Occ\_Tly(T^{TRev}) = Occ\_Tly(T)$;

Implicit Defs. Rt_Lab(T: $\mathbb{Tr}$~{Ω}): $\mathbb{El}$ **and**

Rt_Brhs(T: $\mathbb{Tr}$~{Ω}): Str($\mathbb{Tr}$) **is** (∗ <u>Roo</u>t <u>Lab</u>el and <u>Br</u>anc<u>h</u>e<u>s</u> ∗)

Jn(Rt_Brhs(T), Rt_Lab(T)) = T;

Corollary 1: $\forall$ x: $\mathbb{El}$, $\forall$ α: Str($\mathbb{Tr}$), Rt_Lab(Jn(α, x)) = x **and** Rt_Brhs(Jn(α, x)) = α;

Def. $\mathbb{Site}$ = **Cart_Prod**

Lab: $\mathbb{El}$;

LTS, RTS: Str($\mathbb{Tr}$) (∗ <u>L</u>eft <u>T</u>ree <u>S</u>tring, <u>R</u>ight <u>T</u>ree <u>S</u>tring ∗)

end;

Implicit Def. $(S: \mathbb{Site})^{SRev}$: $\mathbb{Site}$ **is** (∗ <u>S</u>ite <u>Rev</u>ersal ∗)

S^{SRev}.Lab = S.Lab **and** $S^{SRev}.LTS = ([[S.RTS]]^{TRev})^{Rev}$ **and** $S^{SRev}.RTS = ([[S.LTS]]^{TRev})^{Rev}$;

Corollary 1: $\forall$ S: $\mathbb{Site}$, $(S^{SRev})^{SRev} = S$;

Def. $\mathbb{Tr\_Pos}$ = **Cart_Prod** (∗ <u>Tr</u>ee <u>Pos</u>ition ∗)

Path: Str($\mathbb{Site}$);

Rem_Tr: $\mathbb{Tr}$ (∗ <u>Rem</u>ainder <u>Tr</u>ee ∗)

end;

Implicit Def. $(P: \mathbb{Tr\_Pos})^{PRev}$: $\mathbb{Tr\_Pos}$ **is** (∗ <u>P</u>osition <u>Rev</u>ersal ∗)

$P^{PRev}.Path = [[P.Path]]^{SRev}$ **and** $P^{PRev}.Rem\_TR = P.Rem\_TR^{TRev}$;

Corollary 1: $\forall$ P: $\mathbb{Tr\_Pos}$, $(P^{PRev})^{PRev} = P$;

Inductive Def. on ρ: Str(Site) **of** (ρ)Ψ(T: Tr): Tr **is** (∗ zip operator ∗)

(i) Λ Ψ T = T;

(ii) ext(ρ, S) Ψ T = ρ Ψ Jn(S.LTS∘⟨T⟩∘S.RTS, S.Lab);

Corollary 1: ∀ ρ, σ: Str(Site), ∀ T: Tr, (ρ∘σ)ΨT = ρΨ(σΨT);

Corollary 2: ∀ P: Tr_Pos, $(\text{P.Path } \Psi \text{ P.Rem\_Tr})^{\text{TRev}} = \text{P}^{\text{PRev}}\text{.Path } \Psi \text{ P}^{\text{PRev}}\text{.Rem\_Tr}$;

Corollary 3: ∀ P: Tr_Pos, |P.Path| + ht(P.Rem_Tr) ≤ ht(P.Path Ψ P.Rem_Tr);

Corollary 4: ∀ R, S: Site, ∀ T, U: Tr, **if** ⟨R⟩ Ψ T = ⟨S⟩ Ψ U **and**
(|R.LTS| = |S.LTS| **or** |R.RTS| = |S.RTS|), **then** R = S **and** T = U;

Def. (T: Tr) Is_Subtree (U: Tr): B = (∃ ρ: Str(Site) ∋ ρΨT = U);

Corollary 1: Is_Partial_Ordering(Is_Subtree);

Corollary 2: ∀ T, U: Tr, **if** T Is_Subtree U, **then** N_C(T) ≤ N_C(U);

Corollary 3: ∀ T, U: Tr, **if** T Is_Subtree U, **then** ht(T) ≤ ht(U);

Corollary 4: ∀ T, U: Tr, **if** T Is_Subtree U, **then** L_C(T) ≤ L_C(U);

Corollary 5: ∀ T, U: Tr, **if** T Is_Subtree U, **then** Occ_Set(T) ⊆ Occ_Set(U);

Corollary 6: ∀ T, U: Tr, **if** T Is_Subtree U, **then** Occ_Tly(T) ⊆ Occ_Tly(U);

Corollary 7: ∀ T, U: Tr, **if** T Is_Subtree U, **then** T^{TRev} Is_Subtree U^{TRev};

Implicit Def. Split_at(i: ℕ, T: Tr~{Ω}): **Cart_Prod** St: Site, RT: Tr **end** **is**

(∗ produces a Site and a Remainder Tree ∗)

⟨Split_at(i, T).St⟩ Ψ Split_at(i, T).RT = T **and**
|Split_at(i, T).St.LTS| = min(i, |Split_at(i, T).St.LTS| + |Split_at(i, T).St.RTS|);

Corollary 1: ∀ S: Site, ∀ T: Tr, Split_at(|S.LTS|, ⟨S⟩ Ψ T).St = S **and** ∀ i: ℕ,
Split_at(i, ⟨S⟩ Ψ T).RT = T;

Corollary 2: ∀ i: ℕ, ∀ T: Tr~{Ω}, Split_at(i, T).RT Is_Subtree T;

Corollary 3: ∀ i: ℕ, ∀ T: Tr~{Ω}, ht(Split_at(i, T).RT) < ht(T);

Def. (P: Tr_Pos) $\equiv_T$ (Q: Tr_Pos): B = (P.Path Ψ P.Rem_Tr = Q.Path Ψ Q.Rem_Tr); (∗ are tree equivalent ∗)

Corollary 1: Is_Equivalence($\equiv_T$);

Corollary 2: ∀ T: Tr, $\underset{\substack{\text{P: Tr\_Pos} \ni \\ \text{P} \equiv_T (\Lambda, \text{T})}}{\text{Max}}\left(|\text{P.Path}|\right) \leq \text{ht(T)} \leq \underset{\substack{\text{P: Tr\_Pos} \ni \\ \text{P} \equiv_T (\Lambda, \text{T})}}{\text{Max}}\left(|\text{P.Path}|\right) + 1$;

Inductive Def. on T: Tr **of** Yld(T): Str **is** (∗ Yield ∗)

(i) Yld(Ω) = Λ;

(ii) $\text{Yld(Jn}(\alpha, \text{x}) \text{) } = \begin{cases} \langle \text{x} \rangle & \textbf{if } \alpha \in \text{Str}(\{\Omega\}) \\ \text{Ag}_{(\circ, \Lambda)}(\text{Yld}[[\alpha]]) & \textbf{otherwise} \end{cases}$;

Corollary 1: ∀ T: Tr, Yld(T) = Λ **iff** T = Ω;

Corollary 2: ∀ T: Tr, |Yld(T)| = L_C(T);

Corollary 3: ∀ T, U: Tr, **if** T ≅ U, **then** |Yld(T)| = |Yld(U)|;

Corollary 4: ∀ T: Tr, $\text{Yld}(\text{T}^{\text{TRev}}) = \text{Yld(T)}^{\text{Rev}}$;

Corollary 5: ∀ T: Tr, Yld(T) ⊆ Occ_Set(T);

Corollary 6: ∀ T, U: Tr, **if** T Is_Subtree U, **then** Yld(T) Is_Substring Yld(U);

⋮

end General_Tree_Theory;

Appendix F

Left_Right_Conformality_Ext

Extension Left_Right_Conformality_Ext **for** General_Tree_Theory **with** Relativization_Ext;

Def. Is_L_R_Cfml_w($\curlywedge$: (Γ: $\mathbb{Set}$)⊠Γ→$\mathbb{B}$, T: U_Tr(2, Γ)) : $\mathbb{B}$ = (∀ ρ: Str(U_Site(2, Γ)),
∀ LT, RT: U_Tr(2, Γ), ∀ y: Γ, **if** ρ Ψ Jn(〈LT, RT〉, y) = T,
then ∀ x: Occ_Set(LT), ∀ z: Occ_Set(RT), x $\curlywedge$ y **and** y $\curlywedge$ z);
(∗ Is Left Right Conformal with ∗)

Corollary 1: ∀ Γ: $\mathbb{Set}$, ∀ $\curlywedge$: Γ⊠Γ→$\mathbb{B}$, Is_L_R_Cfml_w($\curlywedge$, Ω);

Corollary 2: ∀ Γ: $\mathbb{Set}$, ∀ $\curlywedge$: Γ⊠Γ→$\mathbb{B}$, ∀ LT, RT: U_Tr(2, Γ), ∀ y: Γ,
if Is_L_R_Cfml_w($\curlywedge$, LT) **and** Is_L_R_Cfml_w($\curlywedge$, RT) **and** ∀ x: Occ_Set(LT), x $\curlywedge$ y
and ∀ z: Occ_Set(RT), y $\curlywedge$ z, **then** Is_L_R_Cfml_w($\curlywedge$, Jn(〈LT, RT〉, y));
⋮

end Left_Right_Conformality_Ext;

Appendix G

Search_Tree_Balancing_Ext

Extension Search_Tree_Balancing_Ext **for** General_Tree_Theory **with** Relativization_Ext;
Def. Is_Balanced (T: U_Tr (2, Γ: $\mathbb{Set}$)) : $\mathbb{B}$ = (∀ ρ: Str (U_Site (2, Γ)),
∀ LT, RT: U_Tr (2, Γ), ∀ y: Γ, **if** ρ Ψ Jn(⟨LT, RT⟩, y) = T,
then $0 \le |ht(LT) - ht(RT)| \le 1$
Corollary 1: ∀ Γ: $\mathbb{Set}$, Is_Balanced (Ω);
Corollary 2: ∀ Γ: $\mathbb{Set}$, ∀ LT, RT: U_Tr (2, Γ), ∀ y: Γ,
if Is_Balanced (LT) **and** Is_Balanced (RT) **then** Is_Balanced (Jn (⟨LT, RT⟩, y));
⋮
end Search_Tree_Balancing_Ext;

REFERENCES

[1] Adel'son-Vel'skiĭ G.M. and Landis E.M.: An algorithm for the organization of information, In: USSR Academy of Sciences 146: 263266, 1962.

[2] Beckert B., Hähnle R.,and Schmitt P.H.: Verification of object-oriented software: The KeY approach, Springer-Verlag, 2007.

[3] Bjørner N., Ganesh V., Michel R., and Veanes M.: An SMT-LIB Format for Sequences and Regular Expressions, In: Strings, 2012, p. 24.

[4] Cook C.T., Harton H., Smith H., and Sitaraman M.: Specification Engineering and Modular Verification Using a Web-integrated Verifying Compiler, In: Proceedings of the 34th International Conference on Software Engineering, IEEE, 2012, pp 1379 – 1382.

[5] Harms D., Weide B.W.: Copying and Swapping: Influences on the Design of Reusable Software Components, In: IEEE Transaction on Software Engineering, vol. 17, no. 5, 1991, pp. 424 – 435.

[6] Harton H.: Mechanical and Modular Verification Condition Generation For Object-Based Software, Ph.D. Desertation, Clemson University , 2011.

[7] Kabbani, N.M., et al.: Formal Reasoning Using an Iterative Approach with an Integrated Web (IDE), In: Proceedings Second International Workshop on Formal Integrated Development Environment, F-IDE, 2015, pp. 56 – 71.

[8] Klebenov, V., et al.: The 1st Verified Software Competition: Experience Report. In: Proceedings of the 17th international conference on Formal methods, FM'11, Springer, 2011, pp. 154 – 168.

[9] Kirschenbaum, J., et al.: Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping? In: Proceedings 11th International Conference on Software Reuse, Springer LNCS 5791, 2009, pp. 31 – 40.

[10] Kulczycki, G., et al.: The Location Linking Concept: A Basis for Verification of Code Using Pointers. In: VSTTE'12 Proceedings of the 4th international conference on Verified Soft-ware: theories, tools, experiments, pp. 34 – 49.

[11] Leino K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness, In: LPAR 2010. Springer, 2010, pp. 348 – 370.

[12] Piskac R., Wies T., Zufferey D.: Automating Separation Logic using SMT, In: Computer Aided Verification,Springer Berlin Heidelberg, 2013, pp. 773 – 789.

[13] Sedgewick R. and Wyne K.: Binary Search Trees, In: Algorithms, Boston, MA 02116, Pearson Education, Inc, 2011, pp. 396.

[14] Sitaraman M., Adcock B., Avigad J. et al.: Building a push-botton resolve verifier: Progress and challenges, In: Formal Aspects of Computing 23(5), 607-626, 2011

[15] Smith W.H.: Engineering Specifications and Mathematics for Verified Software. PhD. Desertation, Clemson University, 2013.

[16] Wies T., Muñiz M., Kuncak V.: An Efficient Decision Procedure for Imperative Tree Data Structures. In: Automated Deduction – CADE-23, Springer, 2011, pp. 476 – 491.