A Taxonomy of Graph Representations

by

Gabor Barla-Szabo

Supervisor: Bruce W. Watson

Submitted in partial fulfillment of the requirements for the degree

Master of Science

in

Computer Science

in the Faculty of Natural & Agricultural Science

University of Pretoria

Pretoria

November 2002

Summary

Graphs are mathematical abstractions that are useful for solving many types of problems in computer science. In this dissertation, when we talk of graphs we refer to directed graphs (digraphs), which consist of a set of nodes and a set of edges between the nodes, where each edge has a direction.

Numerous implementations of graphs exist in computer science however, there is a need for more systematic and complete categorisation of implementations together with some proof of correctness. Completeness is an issue because other studies only tend to discuss the useful implementations and completely or partially ignore the rest. There is also a need for a treatment of graph representations using triples instead pairs as the base component. In this dissertation, a solution to each of these deficiencies is presented.

This dissertation is a taxonomic approach towards a comprehensive treatment of digraph representations. The difficulty of comparing implementations with each other is overcome by a creating a taxonomy of digraph implementations.

Taxonomising digraph representations requires a systematic analysis of the two main building blocks of digraphs implementations namely maps and sets. The analysis presented in the first part of the dissertation includes a definition of the abstract data types to represent maps and sets together with

a comprehensive and systematic collection of algorithms and data-structures required for the implementations thereof. These algorithms are then written and re-written in a common notation and are examined for any essential components, differences, variations and common features. Based on this analysis the maps and sets taxonomies are presented.

After the completion of maps and sets implementation foundations the dissertation continues with the main contribution: a systematic collection and implementation of other operators used for the manipulation of the base triple components of digraphs and the derivation of the the final taxonomy of digraphs by integrating the maps and sets implementations with the operators on the sets of triples.

With the digraph taxonomy we can finally see relationships between implementations and we also can easily establish their similarities and differences. Furthermore, the taxonomy is also useful for further discussions, analysis and visualisation of the complete implementation topography of digraph implementations.

Acknowledgments

I would like to thank the a number of people[1] for their support during the course of the research that is reported in this dissertation. First, I would like to thank my immediate family, my wife Gayle, for her continuous loving selfless support, my parents, Gabor Barla-Szabo and Iren Barla-Szabo, my sister and brother in law, Eszter and Theo Rapanos for their loving support, my brother, Daniel Barla-Szabo for all his support with his books and proof-reading my code.

I would like to thank my supervisor Bruce Watson for his countless hours of extraordinary helpfulness and support in all areas related to this dissertation.

Further thanks go to Derrick Kourie and my father in law Ian Kennedy for extremely useful research pointers, Theo Koopman for the C++ programming discussions, Loren Rhodes and Ali Khattak for pointing me to research sources, Kobus Fick and Pieter Conradie for supporting the idea of a study leave. This dissertation would not have been possible without the above people and my thanks go to all.

[1] All titles have been omitted and first names are used.

Contents

List of Figures

Chapter I

Introduction

In this chapter an introduction to the content and structure of this dissertation is presented. **Graphs** are mathematical abstractions that are useful for solving many types of problems in computer science. Consequently, these abstractions must also be represented in Computer programs.

Graphs are very powerful tools for creating mathematical models of a wide variety of situations. Graph theory has been instrumental for analyzing and solving problems in areas as diverse as computer network design as depicted by Figure 1.1, urban planning [DNU00], and molecular biology with DNA-Based Nanocomputers [Adl94]. Because of the large diversity of use of graphs, the field had centuries of research since the research by Euler (1783) and Hamilton (1865) [Enc99] on this topic.

In more general terms, graphs are mathematical objects that are made of dots (nodes) connected by lines (edges). In this dissertation we refer to graphs which are generally discussed in discrete mathematics books. These graphs consists of a set of nodes, and a set of edges between the nodes. Furthermore, discussing graphs we always refer to graphs with edges that have labels and also directions, i.e. directed graphs (Digraphs). It is also important to note

that that these graphs have no layout information whatsoever. I.e. any two graphs are the same as long as the same nodes are connected by the same edges.

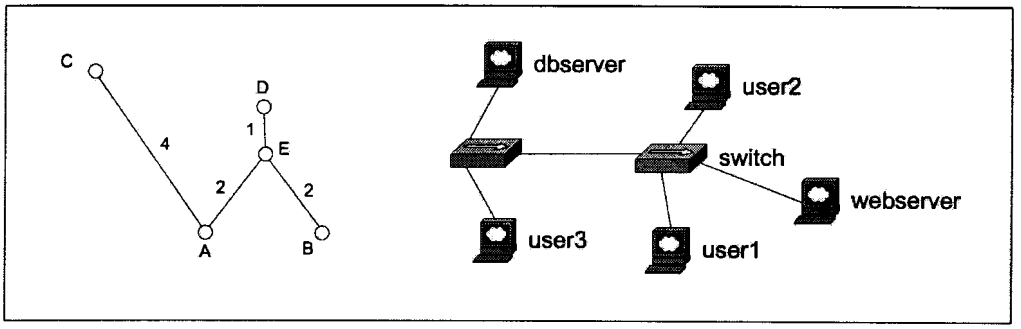


Figure 1.1: Example of two graphs.

Graphs can be implemented in many ways and implementations tend to vary as the application of the graph varies. For example, one implementation could use a Binary Search Tree (BST) for its underlying data-structure and another would use a linked list. One may ask the question: is there a best graph implementation? The answer is a very simple no. There is no best graph implantation that works best for all applications. There could be however, a best graph implementation for a given **context**. The more information we can obtain about the application the more successful implementation is possible. There are programming libraries available which provide standardised implementation [Ste01, JGSL01, Gmb99].

Graph implementation can cover vast areas: from small graphs comprising only a few nodes and edges to huge graphs like the one used in the Human Genome Project [Gen02] representing the 3-billion DNA base pairs. Most likely an implementation that works well for a small graph will not work well when implemented against the task of searching for genetic code

in the human genome.

How do we insert, retrieve and delete information in these graphs efficiently? The programmed implementation of the graph defines the underlying data-structures and policies among other things. The implementation defines the stored format of data and the method of storage and retrieval. If we look at all possible implementations we will soon realise that there are thousands.

At this point, it is important to define what we consider as two different implementations. Obviously, there are almost infinite implementations of graphs, a programmer could —for example— create a large number of hash functions. There is no telling how many hash functions we could create and we cannot count something that is not quantifiable. We therefore will only look at structural implementation differences on a higher level.

As a brief introduction to taxonomies we look at a **definition** as provided by the online technical dictionary [Tec00]:

'**Taxonomy** (from Greek *taxis* meaning arrangement or division and *nomos* meaning law) is the science of classification according to a pre-determined system, with the resulting catalogue used to provide a conceptual framework for discussion, analysis, or information retrieval.'

Using a taxonomy, we can **categorise** graph implementations and depict how the branches of implementations are related to each other.

One of the best known taxonomies is the one devised by Linnaeus (1707–1788) [Wag00], whose taxonomy for biology is still widely used (with modifications). He became known as the Father of Taxonomy. Other interesting taxonomies are also widely available [How95, Alt96].

1.1 Taxonomy Definitions

A **good taxonomy** can provide a general overview of the diversity of graph representations and the relationships between them. In this dissertation, a new type of graph taxonomy is proposed. **The concluding taxonomy in this dissertation uses classification according to relations within the underlying implementations.** The benefits of this new type of arrangement are numerous. The taxonomy study will help us identify the *classification*, *relation*, *diversity* and *enumeration* of graph implementations.

Classification is the arrangement of graph implementations into groups with regards to a predetermined system. In this text the **system of classification** depends on the method of implementation: First, graph implementations are categorised on the basis of their underlying Abstract Data Type (ADT). Then we further refine the categories on the basis data structures and the numerous implementations thereof. For example: Linked List Maps and Array Maps have Lists and Arrays as data structures respectively, and therefore will be two separate categories.

Relationships between the different implementations are identified on the basis of implementation properties such as their representation invariants (e.g. sorting), underlying data structures and variance of implementations within data structures. For example: Linked List Maps and Array Maps are both related to each other by being unsorted and linearly searchable maps.

This taxonomy enables us to more clearly **visualise** the diversity of implementations. With the systematic analysis of our taxonomy we can easily identify the different types of possible implementations and see how one implementation is related to another.

This taxonomy study enables us to **enumerate** the number of possible implementations within our own frame of reference. The number of imple-

mentations is important because it provides us with better understanding of the implementation diversity.

In this dissertation all of the abstract algorithms derived will be presented in **pseudo object pascal**. With the object pascal language we can achieve a well structured and readable code and we can expand on pure pascal (pseudo pascal) to represent structures only available other languages such as C++.

1.2 Dissertation structure

The brief structure of this dissertation is as follows:

- Chapter III builds on currently well known data-structures and presents them in a different way from the classic 'class diagram'. The chapter presents a taxonomy of maps implementations which focuses a systematic overview of map implementations and structuring them into a taxonomy.
- Chapter IV similarly to Chapter III, presents a taxonomy of sets implementations.
- Chapter V is discussion of digraphs and its elements together with the discussion of digraph implementations using maps and sets. The chapter presents a taxonomy of digraph implementations.
- Chapter VI is a concluding discussion of this dissertation.

Chapter II

Preliminaries

2.1 Mathematical

This dissertation assumes basic mathematical knowledge of discrete math. This text has been compiled with compliance to the mathematical notational reference of the book: 'A Logical Approach to Discrete Math' [GS93]. We will adopt the following general additional naming conventions:

- x,y,i,j,k,n for integer variables.
- U,V for arbitrary sets.
- $u_1 ... u_n$ for the elements of the U set.
- $v_1 ... v_n$ for the elements of the V set.
- a,b,c for alphabet symbols.

2.2 Typesetting

New terms, foreign words and jargon words will be typeset in an *italic* shape. Emphasised words will be typeset in **bold** font. Programs, algorithms and code segments will be typeset in `typewriter` font.

2.3 General Glossary

The following section is intended as a reference for later

- **Taxonomy**: (from Greek taxis meaning arrangement or division and nomos meaning law) is the science of classification according to a predetermined system, with the resulting catalogue used to provide a conceptual framework for discussion, analysis, or information retrieval. In theory, the development of a good taxonomy takes into account the importance of separating elements of a group (taxon) into subgroups (taxa) that are mutually exclusive, unambiguous, and taken together, include all possibilities. In practice, a good taxonomy should be simple, easy to remember, and easy to use. [Tec00]

- **Abstract Data Type (ADT)**: A mathematically specified collection of data-storing entities with operations to create, access, change, etc. instances.

- **Concrete Data Structures (CDS)**: Data structures used to implement Abstract Data Types.

- **Central Processing Unit (CPU) intensive**: Normally we refer to a task as being CPU intensive when some operation requires a relatively large portion of available processing to complete it.

- **Class Invariants (CI)**: Class invariants express the fundamental integrity constraints on a class. Some come directly from the axioms of the underlying abstract data type; others (called representation invariants) express properties that the representation must satisfy to be in accordance with the ADT specification. [Mey98]

- **Representation Invariants (RI)**: Representation Invariants express the fundamental integrity constraints on a particular ADT implementation. E.g. 'Sorted' is a RI if the data structure must remain sorted at all times for correct operation.

- **AVL tree**: Balanced tree named after Russian mathematicians G. M Adel'son-Vel'skii and E.M. Landis [AVL62] who published their first result on the AVL balancing conditions in 1962.

2.4 Graphs Glossary

Glossary of special terms of graphs that we need to define:

- **Node**: A vertex is called a node.

- **Edge**: A 'line' between two nodes is called an edge.

- **Edge Label**: The 'label' is the unique identifier of an edge.

- **Graph**: A graph is made up of nodes connected by edges with labels. An edge always has a source and a destination node.

- **Digraphs or Directed Graph**: The direction of the edge is noted, i.e. there is a difference between an edge's source and target node. A node can be the source or the target (or both) of any edge.

- **Path**: A path is a route that one can travel along edges and through nodes in a graph. In Digraphs we can only travel one way on a edge (i.e. from source to target).

Chapter III

Maps

3.1 General Introduction to maps

This chapter is a discussion of the following topics: the concept of maps, the implementations of maps and the taxonomy of maps.

The data-structures discussed in this chapter are available from a number of sources in literature and by no means are presented as something new. However, it is very important to systematically collect and discuss the applicable data-structures (of maps) and their implementation before we can investigate graph implementations. After a comprehensive collection of map implementations at the end of this chapter we present a map taxonomy tree based on the implementations and how they are related to each other. This taxonomy is **similar** to *class diagrams* presented in data-structures books e.g. the *class diagram* of Preiss (1999) [Pre99] and it is also **different** from them in presentation. It is similar because both contain many common elements such as the names of data-structures. It is different because our focus is only on maps implementation, instead of presenting data-structure classes on another level (e.g. a complete class diagram of all data-structures in a

book). Another difference is that our focus is not on the class structure e.g. linking via inheritance, but rather focusing on common higher level features like sorted and unsorted implementations.

The taxonomy is presented in a top down manner, however, the construction process proceeds from bottom up. Each of the algorithms required are rewritten in a common notation and examined for any essential components, differences, variations and common features. The common features are then collected and can be presented together in a taxonomy. Some taxonomy building techniques were adopted from [Wat95] (with modifications).

For a quick and simple mathematical explanation of maps let us define U and V as non empty sets. Let f be a partial function $f \in U \not\rightarrow V$ to represent a mapping of elements of set U to elements of set V. **Note: We use the same names for mathematics concepts and for the data types representing them.** We can also write $dom(f) = U$.

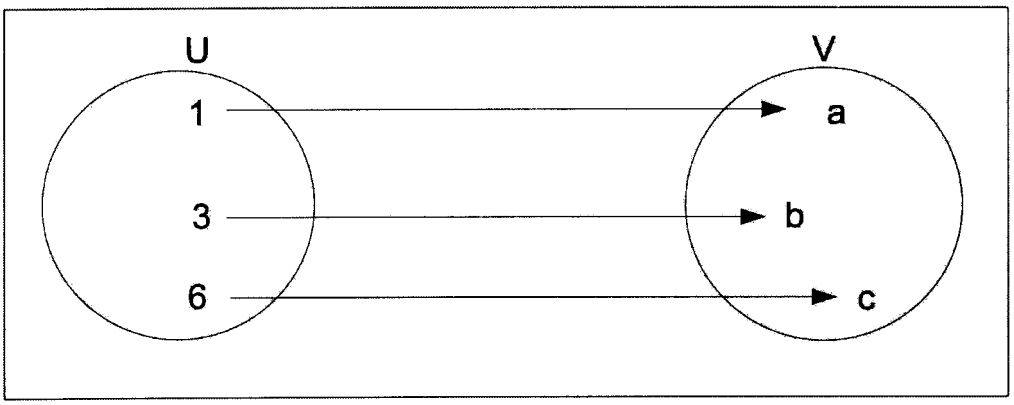


Figure 3.1: Example Binary relation ($1 \mapsto$ a, $3 \mapsto$ b, $6 \mapsto$ c)

Define the domain of $(f) = \{u | (\exists v : v \in V : f(u) = v)\}$. We take $U \not\rightarrow V$ to be an ADT (also written as $map < U, V >$ in the literature). Figure 3.1

depicts an example of the sets U and V and the map relation between their elements. In this figure, the elements in U map to elements in V, with no more than one mapping in V allowed, but not all elements in U require a mapping to a V, hence the **partial** function f. We assume that all maps are from U to V. We will define in this chapter the ADT to represent partial functions like f and give it four operators:

- *indom:* Operator to query the existence of a mapping of some u.
- *lookup:* Operator to query the v value mapped to some u.
- *insert:* Operator to create a mapping of some u to some v.
- *remove:* Operator to remove a mapping of some u to some v.

In programming terms a **map is an object that provides a mapping from a set of key objects to a set of value objects**. Let us call the data-types representing elements of $< U, V >$ in memory `U_TYPE` and `V_TYPE`. Note: `U_TYPE` and `V_TYPE` are **not defined** types since they **can be of any type**. However, `V_TYPE` and `U_TYPE` are usually integer or string as container types for node identifiers and edge labels of digraphs. We declare `U_TYPE` and `V_TYPE` as type `string` in the algorithms in this chapter, see Algorithm 3.1.1 . In Algorithm 3.1.2 we define the `map` class with its four operators. This class will be used in the rest of the chapter as a base class from which we can inherit. Note that these algorithms have been tested using the Delphi compiler.

Algorithm 3.1.1

```
// Type declaration of U_TYPE and V_TYPE declaration in pseudo code
```

```
type
   U_TYPE = string;
   V_TYPE = string;
```

Algorithm 3.1.2

```
 // Map Class declaration in pseudo code
type Map = class
   public function indom(u:U_TYPE):boolean;virtual;abstract;
   public function lookup(u:U_TYPE):V_TYPE;virtual;abstract;
   public procedure insert(u:U_TYPE,v:V_TYPE);virtual;abstract;
   public procedure remove(u:U_TYPE);virtual;abstract;
end;
```

3.2 Signatures and conditions

In all cases throughout this chapter, the following pre and post-conditions hold *true* and are not given explicitly every time:

1. `function indom(u:U_TYPE):bool`

pre: *true*

post: f is unchanged

return: $indom(u) = u \in dom(f)$

2. `function lookup(u:U_TYPE):V_TYPE`

pre: $u \in dom(f)$

post: f is unchanged

return: $lookup(u) = f(u)$

3. `procedure insert(u:U_TYPE,v:V_TYPE)`

pre: $u \notin dom(f)$

post: f is unchanged, except $f(u) = v$

4. `procedure remove(u:U_TYPE)`

pre: $u \in$ dom(f)

post: f is unchanged, except $u \notin dom(f)$

3.3 Unsorted Maps

3.3.1 Introduction

By definition Unsorted Maps (referred to as UM from now on) must be traversed in a linear fashion to find the corresponding mapped value. This is quite inefficient by nature and we will have to traverse $\mathcal{O}(|dom(f)|)$ records before we have a solution. Implementation however is normally quite quick and straight forward. The implementation of UMs can be done in a number of ways. The following sections of this chapter will discuss the different ways UMs can be implemented. There are no fundamental integrity constraints on UMs. Thus, the top level **Representation Invariant (RI)** for UMs: **none**. Note: this does not mean that all UM implementations do not have any RI, but rather means the fact that a structure is an UM does not strengthen the RI.

3.3.2 Policies on UMs

Policies are implementation techniques that are often specified additionally as an 'operational feature' of the implementation. Often policies are used to improve the performance, although this is not a requirement. To further clarify policies let us consider an example: since searching for an element in the list happens from the front to the end, it would be more efficient to move often accessed elements to the front so we can find them faster[1]

Note: We refer to *current record* when we talk about the record that we are currently using. E.g. *current record* during the execution of the lookup operator will change from record to record until we find the u value we searched for. At this point *current record* will point to the record with the requested u.

The following five *on-found* policies can be applied after the indom and lookup operators:

- None: No technique applied. Requires no extra implementation.
- Move to Front: The *current record* is moved to the front of the data structure. Used when the elements in the UM are accessed non-randomly and access repetitions tend to occur.
- Move to End: The *current record* is moved to the end of the data structure.

[1]Note: there are scenarios where implementing these policies will have a negative effect on speed. E.g. if data searches are totally random, the moving of the *current record* to the front will be a waste of computation. Totally random data access occurs rarely in real life scenarios and thus policies work well.

- Move Forwards: The *current record* is moved towards the front of the data structure.
- Move Backwards: The *current record* is moved towards the end of the data structure.

The followings three *on-insert* policies can be applied on the insert operator:

- None (Insert at Front) : Insert at the front of the data structure.
- Insert at End : Insert at the end of the data structure.
- Insert at Random : Randomly insert anywhere into the data structure.

Other, less known policies are not discussed in this text.

3.3.3 Linked List Maps

The function f or `map` class can be represented using a linked list. Among the numerous implementations of UMs the *linked list* is one of the most well known. A single *linked list* is a list of mappings, where each mapping is linked to the next element of the list with a pointer. The element at the front of the list is often called the *head* and the one at the end is called the *tail*. The **RI** of linked list are: *head* always represents the pointer to the first element of the list, *tail* represents the pointer to the last.

By ensuring that the *tail* of the list is always pointing to the *head* — except when the list is empty where *head* = `null` — , we can build a circularly linked list. If the external pointer (the one in type DListNode in the implementation later), points to the current *tail* of the list, then the *head* is found trivially via *tail*→*next*, permitting us to have either *Last In First Out* (*LIFO*) or *First In First Out* (*FIFO*) lists with only one external pointer.

In modern processors, the few bytes of memory saved in this way would probably not be regarded as significant. A circularly linked list would more likely be used in an application which required '*Round-Robin*' scheduling or processing. Figure 3.2 depicts an example consisting of three elements of a single linked list.

A *double linked list* includes a second pointer per list element to store the address of the previous element in the list. *double linked lists* are easier to navigate, as one can move backwards in the list. In this text we investigate the *double linked lists* only.

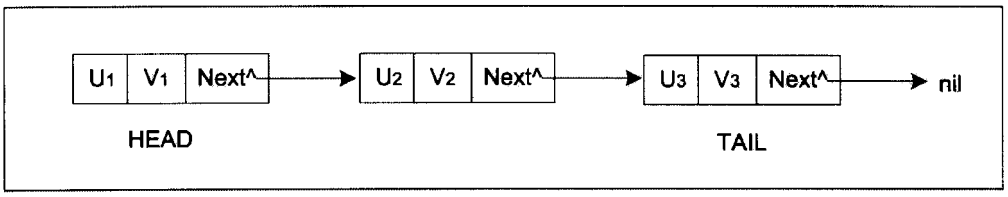


Figure 3.2: Example of a single linked list

Algorithms 3.3.1,3.3.2,3.3.3 and 3.3.4 represent the implementation of the base list node class (`DListNode`), the list class (`DList`), the node and the list constructors respectively. C++ implementations are also commonly available [Mor98, Pre99].

Algorithm 3.3.1

```
 // DListNode Class declaration in pseudo code
type    DListNode = class(TObject)
  private
     fu:  U_TYPE;
     fv:  V_TYPE;
     fnext:  DListNode;
     fprev:  DListNode;
```

```
   public
      constructor create;
      destructor destroy; override;
      property u:  U_TYPE read fu write fu;
      property v:  V_TYPE read fv write fv;
      property next:  DListNode read fnext write fnext;
      property prev:  DListNode read fprev write fprev;
end;
```

Algorithm 3.3.2

```
 type DList = class(Map)
public
   procedure insert(u:  U_TYPE; v:  V_TYPE); override;
   procedure remove(u:  U_TYPE); override;
   function indom(u:  U_TYPE): boolean; override;
   function lookup(u:  U_TYPE): V_TYPE; override;
   constructor create;
   destructor destroy; override;
private
   fhead :  DListNode;
   ftail :  DListNode;
   procedure remove_first();

// Published members have the same visibility as public members.
// The difference is that runtime type information is generated
// for published members.  Publishing allows an application to
// query the fields and properties of an object dynamically
// and to locate its methods.
published
   property head:DListNode read fhead write fhead;
   property tail:DListNode read ftail write ftail;
```

```
end;
```

Algorithm 3.3.3

```
 // The Linked list node constructor and destructor pseudo code
constructor DListNode.create;
begin
   inherited create;
   prev := nil;
   next := nil;
end;

destructor DListNode.destroy;
begin
   inherited destroy;
   prev := nil;
   next := nil;
end;
```

Algorithm 3.3.4

```
 // The Linked list constructor and destructor pseudo code
constructor DList.create;
begin
   inherited create;
end;

destructor DList.destroy;
var
   p1,p2 :  DListNode; //iterators
```

```
begin
   inherited destroy;
   p1:=head;
   // Loop through the list and delete all nodes from memory
   While not (p1 = nil) do
   begin
      p2:=p1.next;
      p1.destroy;
      p1:=p2;
   end;
end;
```

3.3.3.1 The *indom* operator:

The *indom* operator is used to find out whether a v exists for a given u (under the function f). The *indom* operator returns a boolean, *true* or *false* value.

The *indom* operator traverses the *linked list* one by one starting at the *head* until it found the value v for the corresponding u. *true* is returned if u was found, and *false* is returned if *tail* was reached.

In the implementation we will introduce a new element at the end of the list. We will set the new element's u value to the key u we are looking for. We are thus guaranteed to find u in the list. According to our precondition this is not a requirement. This so called 'sentinel' reduces the number of comparisons required in our algorithm and makes it faster.

This implementation requires a new operation *remove_first*. *remove_first* deletes the element pointed by *tail* of the list and moves the *tail* one element

back. The *remove_first* operation is required for efficient implementation. It should be noted however that this technique breaks thread safety, and two threads will not be able to search the same list simultaneously. The *remove* operator alone would have to first find u and then *remove* which is too slow and would defeat the purpose of adding a u to the end of list for better performance. We must also note that *insert* operator adds u to the front of the list. Policies on the *lookup* and *indom* functions:

- None : No technique applied.
- Move-to-front : The *current record* is moved to *head* of the list.
- Move-to-end : The *current record* is moved to *tail* of the list.
- Move-forwards: The *current record* is moved one place towards the front of the data structure except if it is already the *head.*
- Move-backwards: The *current record* is moved one place towards the end of the data structure except if it is already the *tail.*

Algorithm 3.3.5 and 3.3.6 represent the implementation of the *remove_first* and *indom* operation.

Algorithm 3.3.5

```
 // The remove_first operator pseudo code
procedure DList.remove_first();
var
   p :  DListNode;
begin
   // If the list is not empty
   if not(head = nil) then
   begin
```

```
      p:= head;
      if not(head.next = nil) then
      begin
         head.next.prev := nil;
         head := head.next;
      end
      else
      begin
         head := nil;
      end;
      p.destroy;
   end;
end;
```

Algorithm 3.3.6

```
 // The indom operator pseudo code
function DList.indom(u:  U_TYPE): boolean;
var
   p :  DListNode; // p is an iterator
begin
   insert(u, ''); // Insert NULL or '' into the head
   p := tail;
   while not (p.u = u) do
   begin
      p := p.prev; // Go backwards in list
   end;
   if (p = head) then
   begin
      result := false; // result does not quit the function
   end else
   begin
```

```
      result := true; // We found u and will return true
   end;
   remove_first; // Remove the temp first element
end;
```

3.3.3.2 The *lookup* operator

A *lookup* operator is used to return the value v for a given u. (under the function f). The *lookup* operator returns a value from V.

A practical implementation of the *lookup* operator would probably include the *indom* operator also. These two functions are quite similar in implementation. The *lookup* also has to traverse the list, starting from the *head* until it found the key i.e. u and the corresponding v. The corresponding value v of u is then returned.

The pre-condition for *lookup(u)*: *indom(u)* is *true*. Algorithm 3.3.7 represents the implementation of the *lookup* operation.

Algorithm 3.3.7

```
 // The lookup operator pseudo code
function DList.lookup(u:  U_TYPE): V_TYPE;
var
   p:  DListNode; // p is an iterator
begin
   p := head;
   while not (p = tail) do
   begin
      if (p.u = u) then
      begin
         //if we found u then break out of while loop
```

```
            break;
        end
        else
        begin
            //iterator goes to the next element
            p := p.next;
        end;
    end;
    result := p.v;
end;
```

3.3.3.3 The *insert* operator

An *insert* operator is used to add a new element into the linked list. The new element is added to the end of the list. Figure 3.3 depicts an example the *head* and *tail* elements together with their pointers in single liked list.

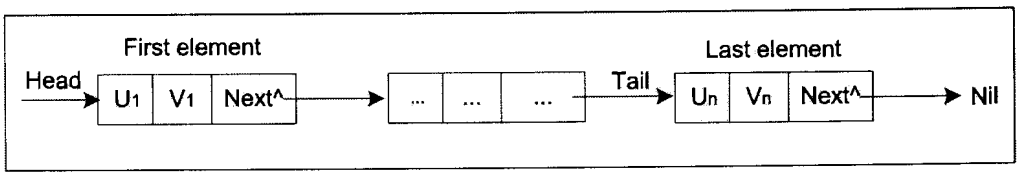


Figure 3.3: Linked list example with pointers.

Applicable policies on *insert* operator:

- None (Insert at Front) : Insert to the front of the list. New element becomes the *head* of the list.
- Insert at End : Insert at the end of the list, ie. the new Tlink is created

and linked at the end of the list. The new element becomes the *tail* of the list.

- Insert at Random : Randomly Insert to the anywhere in the list.

The implementation of the *insert* operator involves adding the new element to the front of the *linked list*. The `next` pointer of the new element is then set to point to the *head*. The new element becomes the *head* and the pointer to *head* changes to point to the new element.
In the implementation we only consider the 'None' (Create at front) policy. Algorithm 3.3.8 represents the implementation of the *insert* operator.

Algorithm 3.3.8

```
 // insert operator pseudo code
procedure DList.insert(u:  U_TYPE; v:  V_TYPE);
var
   newNode:  DListNode;
begin
   newNode := DListNode.create;
   newNode.u := u;
   newNode.v := v;
   // If list is empty
   if (tail = nil) then
   begin
      // Make the tail = the new node.
      tail := newNode;
   end;
   // If the list already has some nodes
   if not (head = nil) then
   begin
      // Make the current head's prev pointer = new node.
```

```
      head.prev := newNode;
   end;
   // Make the new node's next = the current head.
   newNode.next := head;
   // Make the head = the new node.
   head := newNode;
end;
```

3.3.3.4 The *remove* operator

The *remove* operator is used to remove an element from the linked list.

The implementation of the *remove* operator involves finding the element while storing the pointer to the previous element and the *next* element. Once this is done we can remove the *current* element, jump to *previous* element and set its *next* to the stored *next*.

Three special cases must be considered:

- If the removable element is the only element in the list.
- If the removable element is the *head* then after the location of the element containing u, we can remove the element, jump to the stored *next* element and set it to be the *head*.
- If the removable element is the *tail* then after the location of the element containing u, we can remove the element, jump to the stored previous element and set it to be the *tail*.

Algorithm 3.3.9 represents the implementation of the *remove* operator.

Algorithm 3.3.9

```
 // The remove operator pseudo code
procedure DList.remove(u:  U_TYPE);
var
   p:  DListNode; // Iterator
begin
   p := head;
   while not (p = tail) do
   begin
      if (p.u = u) then
      begin
         break; // breaks out of while loop
      end
      else
      begin
         p := p.next;
      end;
   end;

   // ---- List has 1 item ----
   if (p = head) and (p = tail) then
   begin
      p.destroy;
   end
   else
   begin
      // ---- If p is head ----
      if (p = head) then
      begin
         // Make p.next's previous pointer nil
         p.next.prev := nil;

         // Make p.next the head of list
         head := p.next;
```

```
            p.destroy;
        end
        else
        begin
            // ---- If p is tail ----
            if (p = tail) then
            begin

                // make p.prev's next pointer nil
                p.prev.next := nil;

                // make p.prev the tail
                p.prev := tail;
                p.destroy;
            end
            else
            begin
                p.next.prev := p.prev;
                p.prev.next := p.next;
            end;
        end;
    end;
end;
```

3.3.4 Array Maps

3.3.4.1 Introduction to Array Maps (AM)

Array maps utilise the array data-structure to store records. Arrays are probably the most common way to aggregate data because they can be easily accessed using an direct index to the record. Figure 3.4 depicts an example

of array of length ten. By definition the elements of an unsorted array are

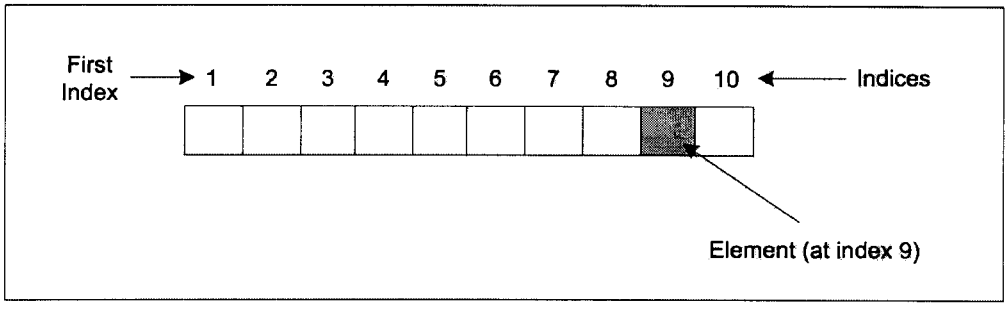


Figure 3.4: Example of an Array

not in any particular order. Due to unavailability of structural information (e.g. sorting), the only way to find a particular element in the array is to traverse the array until we found it. The efficiency of this technique is rated to be quite bad $\mathcal{O}(|dom(f)|)$ as it requires $|dom(f)|$ steps in the worst case to find the entry. The operations that were applicable to *linked lists* are also applicable to unsorted arrays i.e. *indom*, *lookup*, *insert*, *remove*.

The implementation inherits the four operations from the `map` class (Algorithm 3.1.2). We define an array of n by 2 dimensions. Where n is the number of mapping in f , i.e. n is the same number as the domain of f. We assume that array `A` can dynamically[2] grow.

Algorithms 3.3.10,3.3.11,3.3.12 represent the implementations of the base array data structure declaration (`MyElem`), the array class (`DArray`) and the array constructor respectively.

Algorithm 3.3.10

```
// Declaration of data structure for array
```

[2]In the Pascal language the procedure `setLength(A,n)` dynamically sets the size of the array `A` to n.

```
type
MyElem = record
   u :  U_TYPE;
   v :  V_TYPE;
end;
```

Algorithm 3.3.11

```
// Declaration of the DArray Class
type DArray = class(Map)
public
   constructor create;
   destructor destroy; override;
   procedure insert(u:  U_TYPE; v:  V_TYPE); override;
   procedure remove(u:  U_TYPE); override;
   function indom(u:  U_TYPE): boolean; override;
   function lookup(u:  U_TYPE): V_TYPE; override;
private
   A: Array of DArrayElem;
   A_Size:  integer;
   procedure Remove_Last();
end;
```

Algorithm 3.3.12

```
// The Array constructor and destructor pseudo code
constructor DArray.create;
begin
   inherited create;
   A_Size := 0;
```

```
end;
destructor DArray.destroy;
begin
   inherited destroy;
end;
```

3.3.4.2 The *indom* operator

The *indom* operation will return *true* if v exists for a given u, otherwise it returns *false* (under the function f).
The following four 'on found' policies are applicable to the *indom* and *lookup* operators.

- None : No technique applied.

- Move to Front : The *current record* is moved to the front of the array (`A[0]`) and all the other elements are moved up one element in the array. This is quite computationally intensive because we have to move all elements.

- Move Forwards : The *current record* is swapped with the previous element unless it is already `A[0]`.

- Move Backwards : The *current record* is swapped with the *next record* unless it is already `A[A_Size]`.

Implementation[3] Algorithm 3.3.13 represents the implementation of the *indom* operation.

Algorithm 3.3.13

```
 // The indom operator pseudo code for arrays
function DArray.indom(u:  U_TYPE): boolean;
var
   i:  integer;
begin
   i := 0;
   // Insert an empty element to the end of the Array
   insert(u, '');
   while not (u = A[i].u) do
   begin
      inc(i);
   end;
   //Test if we reached the end of the array.
   if (i = A_Size-1) then
   begin
      result := false;
   end
   else
   begin
      result := true;
   end;
   // Remove the temp last element
   Remove_Last;
end;
```

[3]We use the variable `A_Size` to represent size of the array in the implemented array structure. The first element of an array is 0 and we have `A_Size` elements in the array. If we need n mappings `A_Size=n-1`.

3.3.4.3 The *lookup* operator

The *lookup* operator is used to return the value v for a given u under the function f). The implementation of *lookup* for arrays traverses the array `A` (defined above) similarly to the *indom* operator and then returns value v for the corresponding u. Algorithm 3.3.14 represents the implementation of the *lookup* operation.

Algorithm 3.3.14

```
 // The lookup operator pseudo code for arrays
function DArray.lookup(u:  U_TYPE): V_TYPE;
var
   i:  integer; // index variable
begin
   i := 0;
   // pre:  indom is true, hence while true can be used.
   while true do
   begin
         if (A[i].u = u) then
      begin
         // Break out of the while loop when u was found.
         break;
      end
      else
      begin
         // Increase the index variable i
         inc(i);
      end;
   end;
```

```
   result := A[i].v;
end;
```

3.3.4.4 The insert operator

The *insert* operator is used to add a new entry into the array. The following implementation adds the new element to the end of the array. Similarly to the array used in the *lookup* operator array `A` in this implementation has n elements.

The following three 'on insert' policies may be applied on the *insert* operator.

- None (Insert at front) : The new element is inserted at `A[0]` and rest of the array elemts get bumped to a one higher index. This requires a lot of swapping of elements but can result in faster linear searching.
- Insert at Random : The new element is inserted at `A[RND(A_Size)]` i.e. at the randomly somewhere in the array `A`.
- Create at End : The new element is inserted at `A[A_Size+1]` i.e. at the end of the array.

In the implementation we only consider the 'Insert at End' policy. Algorithm 3.3.15 represents the implementation of the *insert* operation.

Algorithm 3.3.15

```
 // The insert operator pseudo code for arrays
procedure DArray.insert(u:  U_TYPE; v:  V_TYPE);
begin
   inc(A_Size);
   setLength(A, A_Size); // Increase array length
```

```
   A[A_Size-1].u := u;
   A[A_Size-1].v := v;
end;
```

3.3.4.5 The *remove* operator

The *remove* operator is used to remove an entry from array A. Algorithm 3.3.16 represents the implementation of the *remove* operation and the *Remove_Last* operator.

Algorithm 3.3.16

```
 // The remove operator pseudo code for arrays
procedure DArray.insert(u:  U_TYPE; v:  V_TYPE);
begin
   inc(A_Size);
   setLength(A, A_Size);
   A[A_Size-1].u := u;
   A[A_Size-1].v := v;
end;

procedure DArray.Remove_Last();
begin
   dec(A_Size);
   setLength(A, A_Size);
end;
```

3.4 Sorted Maps

3.4.1 Introduction

Sorted Maps (SM) are a modification of Unsorted Maps (UM). By sorting[4] on the u part of the container we can achieve a data-structure, which can be searched much more efficiently than the UMs. SMs have a $\mathcal{O}(\log |dom(f)|)$ efficiency during a search which is good, especially for a large number of records. The downside to the 'binary' approach is the need for sorting of the records and maintaining the **RI** of records being sorted (kept in a binary structure) at all times.

3.4.2 Policies

The **policies** of the previous section are not applicable for the SM. The reason for this is that moving elements (e.g. Move to front policy) **would violate the RI**, i.e. sorting. The only two policies for SMs are the following:

- Ascending : Data structure is sorted ascending.
- Descending : Data structure is sorted descending.

Unlike the policies discussed for the UMs, these two policies are fixed at construction time. I.e. Once the Ascending policy is used, it cannot be changed later and because of this they form part of the RI.

[4]Sorting of records from lowest to highest should be done using an efficient sorting algorithm. The *quick sort* algorithm is recommended. [C.A61]

3.4.3 Sorted Array Maps

Sorted Array Maps (SAM) are arrays with a stronger **RI** than the unsorted array maps. Ordering must be established and maintained. As a practical choice for this section we will use integers for our index types, note that this is not a requirement and any type can be used as long as an ordering rule can be established. The declaration of data structure for sorted array map is the same as the definition for a non-sorted array map. The only difference is in the ordering of the stored data.

Algorithm 3.4.1 represents the implementation of the `DArray` class for Sorted Array maps.

The 3.4.2 represents the implementation of the binary search algorithm which returns the position in the array where the element can be found. The function returns -1 if the value is not in the array. All three operators, namely indom, lookup and remove make use of the `binarysearch` function to avoid redundant code.

Algorithm 3.4.1

```
 // Declaration of the DArray Class for SAM
type DArray = class(Map)
public
   constructor create;
   destructor destroy; override;
   procedure insert(u:  U_TYPE; v:  V_TYPE); override;
   procedure remove(u:  U_TYPE); override;
   function indom(u:  U_TYPE): boolean; override;
   function lookup(u:  U_TYPE): V_TYPE; override;
private
   A: Array of DArrayElem;
   A_Size:  integer;
```

```
   procedure remove_last();
   procedure sort();
   function BinarySearch(u:  U_TYPE): integer;
end;
```

Algorithm 3.4.2

```
 // The BinarySearch operator pseudo code for arrays
function DArray.BinarySearch(u:  U_TYPE): integer;
var
// L is the left and R is the Right boundary of the search
// interval and M is the midpoint of the search interval.
   L, R, M: integer;
   foundIndex:  integer;
begin
   L := 0;
   R := A_Size-1;
   foundIndex := -1;
   while (L <= R) do
   begin
      M := floor((L + R) / 2);
      if (u = A[M].u) then
      begin
         foundIndex := M;
         break; // Break out of the while loop
      end
      else
      begin
         if (u > A[M].u) then
         begin
            L := M + 1;
         end
         else
```

```
         begin
            R := M - 1;
         end;
      end;
   end;
   result := foundIndex;
end;
```

3.4.3.1 The *indom* operator

The indom operator will return true if v exists for a given u, otherwise it returns $false$ (under the function f). The implementation requires the availability of $=$ and $<$ operations on $\mathtt{A}(u)$.

Precondition: `A[n]` is a fully sorted array of n elements.

As we traverse our sorted array maps it will take us at most $(n+1)/2$ comparisons before we will find our goal. This worst case scenario is described in the text of [Sta98] with reference to Binary Search Trees which have the same traversal performance properties as sorted arrays.

Algorithm 3.4.3 represents the implementation of the *indom* operation.

Algorithm 3.4.3

```
 // The indom operator pseudo code for SAMs
function DArray.indom(u:  U_TYPE): boolean;
begin
   // If Binarysearch(u) = -1 then u was not found
   if (BinarySearch(u) > -1) then
   begin
      result := true;
   end
```

```
   else
   begin
      result := false
   end;
end;
```

3.4.3.2 The *lookup* operator

The *lookup* operator is used to return the value v for a given u. Precondition: `A[n]` is sorted. Algorithm 3.4.4 represents the implementation of the *lookup* operation.

Algorithm 3.4.4

```
 // The lookup operator pseudo code for SAMs
function DArray.lookup(u:  U_TYPE): V_TYPE;
begin
   // Since Pre:  Indom(u) = true
   // BinarySearch(u) returns the index in the array.
   result := A[BinarySearch(u)].v;
end;
```

3.4.3.3 The *insert* operator

There are at least two *on-insert* policies for SAMs.

- None: The new element inserted into the already sorted array after searching for the the right position.

- Insert at End: The new element can be added to the end of the array, and then the the array is sorted on u because the new element possibly disordered the array. This technique is not very efficient.

Algorithm 3.4.5 represents the implementation of the insert operation which inserts an element at the end of the array.

Algorithm 3.4.5

```
 // The insert operator pseudo code for SAMs
procedure DArray.insert(u:  U_TYPE; v:  V_TYPE);
begin
   inc(A_Size);
   setLength(A, A_Size);
   A[A_Size-1].u := u;
   A[A_Size-1].v := v;
   sort; //Where sort is a sort function.
end;
```

3.4.3.4 The *remove* operator

The *remove* operator is used to remove an entry from the sorted array. Note: the removal of an entry does not change the sorting order.

Algorithm 3.4.6 represents the implementation of the *remove* operation.

Algorithm 3.4.6

```
 // The remove operator pseudo code for arrays
procedure DArray.remove(u:  U_TYPE);
var
   i,j:  integer;
```

```
begin
   // Return the array index into i
   i := BinarySearch(u);
   // and move the rest of the array one element back
   for j := i to (A_Size-2) do
   begin
      A[j].u := A[j+1].u;
      A[j].v := A[j+1].v;
   end;
   // Remove the last element from the end of the array.
   Remove_Last;
end;
```

3.4.4 Binary Search Tree maps

Basic concepts and terminology of a *tree*: *Tree* diagrams are formed from nodes and line segments. The lines segments are called either edges or branches. The three sources of terminology for *trees* are:

- Family relationships (parents, children)
- Directional relationships (left, right, bottom, top)
- Biological description (roots, leaves, branch)

The Binary Search Tree (BST) is a *tree* in which each node has exactly two children. These children nodes are permitted to be empty. A *leaf* of a *tree* is defined by a node with two empty children.

The **RI** of a BST: The *tree* structure is such that the branch to the left of the node contains the u's which are smaller ($<$) than the u of the parent

node, and the branch to right contains the u's which are larger ($>$) than the u of the parent.

The **RI** can also be presented in a recursive fashion: A BST is either an empty *tree* or consists of a node that has left and right subtrees that are BSTs.

Each node contains a mapping from an element of U to an element of V in function f. Furthermore, each node has a left and right pointer which point to other nodes. The structure of the *tree* is such that the branch to the left of the node contains the us which are smaller ($<$) than the u of the parent node, and the branch to right contains the u's which are larger ($>$) than the u of the parent.

Figure 3.5 part a depicts an example BST where the value of every index element u_i is the same as its index i. $u_1 = 1$,$u_2 = 2$ and so on. The part b of the figure shows the real values assigned to each u. The values assigned to each v remain inconsequential. A BST gives the best **performance** when it is balanced. Balanced or AVL tree (see.2.3) is a BST where the left and right subtrees of every node have the give or take one the same height. A good balance condition will ensure that the height of the tree with n nodes is $\mathcal{O}(\log n)$. Also, the work required to balance the tree when an item is inserted or deleted is $\mathcal{O}(1)$.

AVL tree balancing condition: $|h_L - h_R| \leq 1$, where h_L is the height of the left subtree and h_R is the height of the right subtree of any node.

3.4.4.1 Policies

The only two policies for BSTs are the following:

- None: No technique applied.

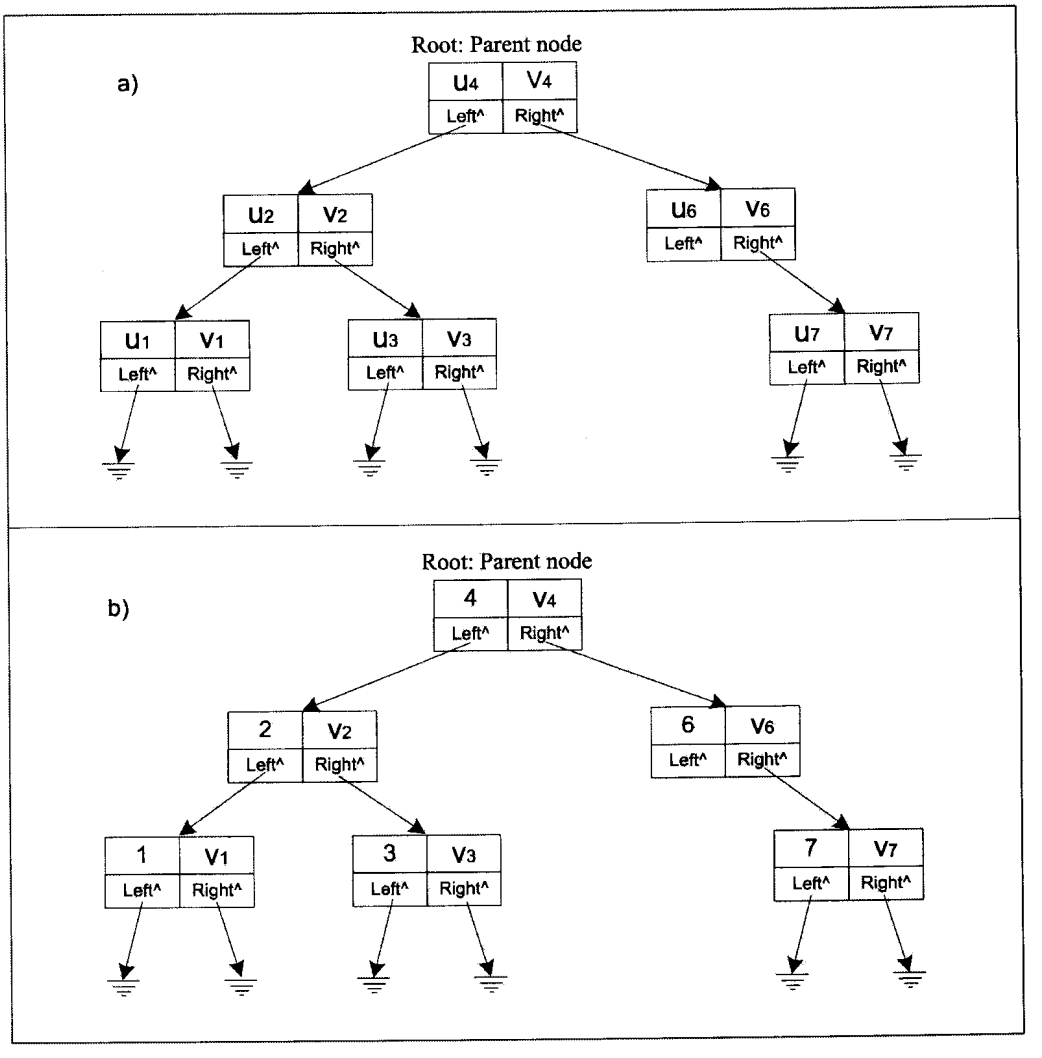


Figure 3.5: a) An abstract BST b) The same BST with real index values

- Balancing: The tree is balanced using AVL Tree balancing condition.

These two policies are fixed at construction time and cannot be changed dynamically at run-time and thus form part of the **RI**.

3.4.4.2 The *indom* operator

The key (u) is the part of the information which we have to test for and hence an equality operation (=) is required. For the traversal we will need a less than (<) and a larger than operation (>). Some of these operations may not always be necessary under certain implementations.

Post-condition: *true* is returned if the v does exist for the given u, where $v \in V$, $u \in U$ and $U \not\rightarrow V$.

Algorithms 3.4.7,3.4.8,3.4.9,3.4.10 represent the implementations of the tree node class declaration (`DTreeNode`), the tree class declaration (`DTree`), the tree node constructor and the tree constructor respectively.

Algorithm 3.4.7

```
 // Dynamic Tree Node Class declaration in pseudo code
type DTreeNode = class(TObject)
   private
      fu:  U_TYPE;
      fv:  V_TYPE;
      fleft:  DTreeNode;
      fright:  DTreeNode;
   public
      constructor create;
      destructor destroy; override;
   published
      property u:  U_TYPE read fu write fu;
      property v:  V_TYPE read fv write fv;
      property left:  DTreeNode read fleft write fleft;
      property right:  DTreeNode read fright write fright;
end;
```

Algorithm 3.4.8

```
 // Dynamic Tree Class declaration in pseudo code
type DTree = class(Map) public
   constructor create;
   destructor destroy; override;
   procedure insert(u:  U_TYPE; v:  V_TYPE); override;
   procedure remove(u:  U_TYPE); override;
   function indom(u:  U_TYPE): boolean; override;
   function lookup(u:  U_TYPE): V_TYPE; override;
private
   froot:  DTreeNode;
end;
```

Algorithm 3.4.9

```
 // The Tree node constructor and destructor pseudo code
constructor DTreeNode.create;
begin
   inherited create;
   left := nil;
   right := nil;
end;

destructor DTreeNode.destroy;
begin
   inherited destroy;
   if not (left = nil) then
   begin
      left.destroy;
```

```
   end;
   if not (right = nil) then
   begin
      right.destroy;
   end;
   left := nil;
   right := nil;
end;
```

Algorithm 3.4.10

```
 // The Tree constructor and destructor pseudo code
constructor DTree.create;
begin
   inherited create;
   froot := nil;
end;

   destructor DTree.destroy;
begin
   inherited destroy;
   if not (froot = nil) then
   begin
      froot.destroy;
   end;
   froot := nil;
end;
```

The *indom* operator traverses the BST in a top-down fashion. The root node is tested first whether it is larger than, less than or equal to the input

variable $u \in U$. If the input value is less than the u value of the node then the next node to be investigated is the node in the left pointer. If the input value is larger than the u value of the node then the next node to be investigated is the node the right pointer. If the input value is equal to the u of the node then we found the corresponding mapping and *true* is returned. If the next node is `null` then the given value has no corresponding mapping in the *tree* and thus *false* is returned by the *indom* operation. Algorithm 3.4.11 represents the implementation of the *indom* operation.

Algorithm 3.4.11

```
 // The indom operator pseudo code for Binary Search Trees
function DTree.indom(u:  U_TYPE): boolean;
var
   p:  DTreeNode; //Iterator
   FoundPosition:  boolean;
begin
   p := froot;
   FoundPosition := false;
   if not (p = nil) then // not empty tree
   begin
      while (true) do
      begin
         if (u = p.u) then
         begin
            FoundPosition := true;
            break;
         end;
         if (u < p.u) then
         begin
            if (p.left = nil) then
            begin
```

```
                break;
            end
            else
            begin
                p := p.left;
            end;
        end
        else
        begin
            if (p.right = nil) then
            begin
                break;
            end
            else
            begin
                p := p.right;
            end;
        end;
    end;
  end;
  result := FoundPosition;
end;
```

3.4.4.3 The *lookup* operator

The *lookup* operator is used to return the value v for a given u. (under the function f). The *lookup* operator returns a value from V.

The implementation is once again very similar to the *indom* operator's implementation. The difference is that the *lookup* operator assumes that the *indom* operation is *true* for the given $u \in U$, and the operation returns the

corresponding v instead of reporting whether there is such a mapping.

If the list is empty then *head=tail*. Algorithm 3.4.12 represents the implementation of the *lookup* operation.

Algorithm 3.4.12

```
// lookup operator pseudo code for Binary Search Trees
function DTree.lookup(u:  U_TYPE): V_TYPE;
var
   p:  DTreeNode; //iterator
begin
   p := froot;
   while (true) do
   begin
      if (u = p.u) then
      begin
         break; // Break out of while loop
      end;
      // If u is less then the iterators u then
      if (u < p.u) then
      begin
      // go down the left branch
         p := p.left;
      end
      else
      begin
         p := p.right;
      end;
   end;
   result := p.v;
end;
```

3.4.4.4 The *insert* operator

The *insert* operator inserts a new node into the *tree*. Inserting an element into the *tree* can be quite challenging. The mapping of the new element u to v must be stored in a node, in the *tree* and in the correct branch.
The implementation of the *insert* operator is shown in Algorithm 3.4.13. A C++ implementation is available from [Pre99].

Algorithm 3.4.13

```
 // lookup operator pseudo code for Binary Search Trees
procedure DTree.insert(u:  U_TYPE; v:  V_TYPE);
var
   p, NewNode:  DTreeNode; //Iterator and new node
   // FoundPosisiton is a boolean to represent if a suitable
   // position was found for insert.
   FoundPosition:  boolean;
begin
   NewNode := DTreeNode.Create;
   NewNode.u := u;
   NewNode.v := v;
   FoundPosition := false;
   if (froot = nil) then
   begin
      froot := NewNode;
   end
   else
   begin
      p := froot;
      while not FoundPosition do
      begin
         if (u < p.u) then
         begin
```

```
            if (p.left = nil) then
            begin
               p.left := NewNode;
               FoundPosition := true;
            end
            else
            begin
               p := p.left;
            end;
         end
         else
         begin
            if (p.right = nil) then
            begin
               p.right := NewNode;
               FoundPosition := true;
            end
            else
            begin
               p := p.right;
            end;
         end;
      end;
   end;
end;
```

3.4.4.5 The *remove* operator

The *remove* operator removes a node from the *tree*. The *remove* operator requires that the u is true under the *indom* operator. When removing an item from the tree, it is imperative that the tree which remains satisfies the

data ordering criterion. A commonly used remove implementation moves the node to the bottom of the branch and then simply deletes the node. The implementation of the *remove* operator for the BST is too long to appear in this text. A C++ implementation is available from [Pre99].

3.5 Hashing Maps

Hashing is the third and final major category of map implementations.

3.5.1 Introduction to hashing

In some cases, although we might have a very large number of possible keys (K), only a fraction of all of these keys would be used in an actual table. In cases like when we consider efficiency of retrieval of record information we normally find that Hashing Maps (HM) has the best performance. The other benefits for using a hash table are that its insertion and removal operations are on average very fast. The hash table as a data structure combines the direct access used in vectors with the flexibility of linked lists. For these reasons, the hash table has become very popular data structure for many purposes.

Suppose that we have to store 5000 citizens 13 digit ID number and their corresponding names in a table. For a quick comparison of techniques of retrieval we can place the records into an array `A` in ascending order of keys and to use binary search to locate the record. Binary search is known to take approximately $(\log_2 5001 - 1)$ comparisons (about 11.3). On the other hand with double hashing 3.5.4 if we were to store the 5000 records in a table `A` that had space for 6000 records, we could reduce the average

number of comparisons needed to locate an employee record to fewer than 2.15 comparisons, improving our efficiency more than four times.

A **hash function** is a mapping, `hash(`K`)` that sends a key K onto the address of a table entry in table `A`. In cases when `hash(`u_1`)=hash(`u_2`)` we have a **collision** of hashed keys and we cannot store the both entries under the same unique key. For cases like this we need to introduce a collision resolution policy to find extra storage.
Note: Traversal in sorted order performs poorly under hashed tables, due to the fact that the hashing function scatters items as randomly as possible throughout the array.

For our implementation the mapping `U_TYPE` to `V_TYPE` is required. The elements in U are the unique keys K, which we seek to find storage for.
Let us define `A` to be an array of type of records in V.
Array `A` could also be called the hash table. The following are the RI for HM:
The mapped records are mapped using the hash function(s). All collisions are resolved to different storage space.

3.5.2 Direct hashing maps

In case of direct hashing a hash function takes an arbitrary integer i and map it into an integer that we can use as an array index. Note: In general direct hashing uses large amounts of memory as it stores an array element for each address. Algorithms 3.5.1,3.5.2 and 3.5.3 represent the implementations of the base data structure declaration together with the hash table class (`MyElem` and `DHashTable`), the hash table constructor and the a hashing

function respectively.

Algorithm 3.5.1

```
 // Declaration of base data structure for an array
   type
   MyElem = record
      u :  U_TYPE;
      v :  V_TYPE;
   end;

// Partial Declaration of the DHashTable Class
DHashTable = class(Map)
private    Count:  Integer
   Alpha:  Double;
   Capacity:  Integer;
   MaxFillRatio:  Double;
protected
   function hash(x:U_TYPE):integer;
   procedure SetCapacity(NewCapacity:  Integer);
public
   procedure insert(x:U_TYPE;y:V_TYPE); override;
   procedure remove(x:U_TYPE); override;
   function indom(x:U_TYPE): Boolean; override;
   function lookup(x:U_TYPE): V_TYPE; override;
   constructor Create;
   destructor Destroy; override;
private
   A : Array of Myelem;
end;
```

Algorithm 3.5.2

```
// The DHashTable constructor and destructor pseudo code
constructor DHashTable.create;
begin
   inherited create;
   Alpha := (Sqrt(5.0) - 1) / 2.0;
   MaxFillRatio := 0.8;
   SetCapacity(256);
end;
destructor DHashTable.destroy;
begin
   inherited destroy;
end;
```

Algorithm 3.5.3

```
// The Hash function implementation example pseudo code
function THashTable.Hash(x:  U_TYPE): Integer;
begin
   if Key < 0 then Error('Error:  Keys must be positive.');
   Result := Trunc(Capacity * Frac(Alpha * x));
end;
```

3.5.2.1 The *indom* operator

With direct hashing our hash function h is a one to one mapping. `A[h(`u_1`)]=` v_1

In the implementation, all elements of the array `A` are assumed to be `null` when not in use. The *indom* operator checks whether the given u maps to `null` in then hash table array. If the result is `null` then *indom* returns `false`, otherwise it returns `true`. Due to the volume of the implementation we will only look at the most important part of the implementation and omit the rest. For a complete Object Pascal implementation of hash tables please refer to [Joh99]. Algorithm 3.5.4 represents the implementation of the *indom* operation.

Algorithm 3.5.4

```
 // The indom operator pseudo code
function DHashTable.indom(u:  U_TYPE): boolean;
begin
   if (A[hash(u)]=null) return false
   else return true;
end;
```

3.5.2.2 The *lookup* operator

The *lookup* operator will return the value v for the given u.
Algorithm 3.5.5 represents the implementation of the *lookup* operation.

Algorithm 3.5.5

```
// The lookup operator pseudo code
function DHashTable.lookup(u:  U_TYPE): V_TYPE;
begin
   return A[hash(u)];
end;
```

3.5.2.3 The *insert* operator

The *insert* operator creates a new entry in the hash table Algorithm 3.5.6 represents the implementation of the *insert* operation.

Algorithm 3.5.6

```
// The insert operator pseudo code
procedure DHashTable.insert(u:U_TYPE,v:V_TYPE);
begin
   if count=capacity then throw domain_error('Hash table full') else
   A[hash(u)]:=v;
   inc(count);
end;
```

3.5.2.4 The *remove* operator

The *remove* operator removes an entry in the hash table which corresponds to the given u. Algorithm 3.5.7 represents the implementation of the *remove* operation.

Algorithm 3.5.7

```
 // The insert operator pseudo code
procedure DHashTable.remove(u:U_TYPE);
begin
   A[hash(u)] := null;
   dec(count);
end;
```

3.5.3 Bucket Hashing

Let us consider the problem caused by collisions in the hash table. Suppose that our hash function hashes two different input values into the same container in hash table A. One method of resolving collisions in the hash table is to **introduce sub maps**. A sub map is an additional data structure linked within a hash table entry which can store and access the values which had collision in the hash function. These sub maps are often referred to as *buckets* in the literature.

We will consider the case when the sub map is Linked List Map (LLM)[5]. Figure 3.6 depicts the *bucket* sub map to as a single linked list. In the figure, when a collision occurs the elements sharing the same entry in the hash table are to be placed in a LLM. The LLM can be traversed, queried and modified as we have discussed in the earlier section.

3.5.3.1 The *indom* operator

With direct hashing our hash function h is a one to one mapping. $f(z) =$ `A[h(z)]`

[5]Similarly, other maps can also be used.

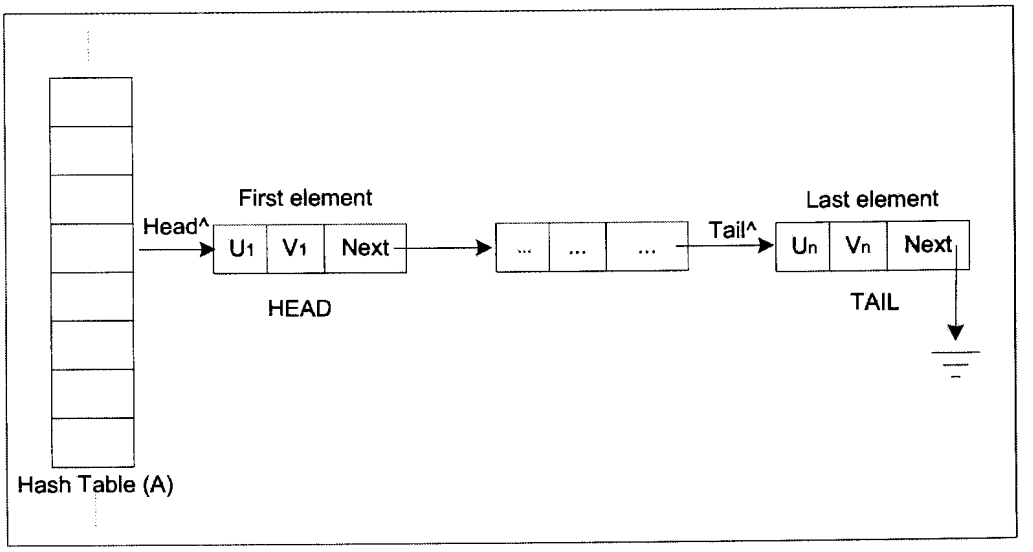


Figure 3.6: Example of a hash table a with Linked List (sub) Map

for example

$h(1) = 1$ and `A[h(1)]= v1`

As in the previous section, in bucket hashing the hash table is an associative data structure that uses keys to refer to its elements. The elements of a hash table reside in an array `A`. All elements of the Array `A` link to a secondary data-structure e.g. a LLM. When an element of `A` does not have a corresponding value mapped, then the LLM is of zero length.

The *indom* operator checks whether the given u maps to a `null` in then hash table array. If the result is `null` then *indom* returns `false`, otherwise it returns true. If the value in `A` is not `null` for a given u then the *indom* operator traverses the LLM to verify whether the given u is present. `True` is returned when u was found and `false` is returned when it was not.

For the implementation we need an array of lists. Let us define an array

A of type list. Let us assume that array A can dynamically grow. Algorithm 3.5.8 represents the implementation of the hash table with embedded lists.

Algorithm 3.5.8

```
// Declaration of data structure for Hash table with embedded Lists
  type
  MyElem = record
     lst:  list;
  end;
  MyVector = array of MyElem;
  var
     A: MyVector;
```

With this new data-structure we can implement the four operations. Algorithm 3.5.9 represents the implementation of *indom* operation.

Algorithm 3.5.9

```
 // The DHashTable.indom operator pseudo code
function indom (u:  U_TYPE): boolean;
begin
   if A[hash(u)].lst.indom(u) then return true
   else return false;
end;
```

3.5.3.2 The *lookup* operator

The *lookup* operator will return the value v for the given u.

Algorithm 3.5.10 represents the implementation of *lookup* operation.

Algorithm 3.5.10

```
 // The lookup operator pseudo code
function DHashTable.lookup (u:  U_TYPE): V_TYPE;
begin
   return A[hash(u)].lst.lookup(u);
end;
```

3.5.3.3 The *insert* operator

The *insert* operator creates a new entry in a list indexed by the hash table.

Algorithm 3.5.11 represents the implementation of *insert* operation.

Algorithm 3.5.11

```
 // The insert operator pseudo code
procedure DHashTable.insert(u:U_TYPE,v:V_TYPE);
begin
   A[hash(u)].lst.insert(u,v);
   inc(count);
end;
```

3.5.3.4 The *remove* operator

The *remove* operator removes an entry in a list indexed by the hash table. Algorithm 3.5.12 represents the implementation of *remove* operation.

Algorithm 3.5.12

```
 // The remove operator pseudo code
procedure DHashTable.remove(u:U_TYPE);
begin
   A[hash(u)].lst.remove(u);
   dec(count);
end;
```

3.5.4 Hashing with re-hashing

Hashing with **re-hashing**, also known as *double hashing* is an open-addressing scheme that drastically reduces clustering. Double hashing defines a key dependent probe sequence, in other words the sequence of probing depends on the key as described by [FCV98].

In this scheme the probe sequence still searches the table in a linear order, starting at the location $hash(u)$, but a second **hash function** $hash_2$ **determines the size of the steps taken.**

The general guidelines to be followed for choosing the hash functions 1 and 2 are:

- $hash_2(u) \neq 0$
- $hash_2(u) \neq hash(u)$

We need nonzero step size for $hash_2(u)$ to define the probe sequence. Furthermore, $hash_2$ must differ from $hash$ to avoid clustering. While more than two hash functions can be desirable, such schemes are difficult to implement

In schemes involving open-addressing, as the hash table fills, the probability of a collision increases. At a certain point a larger hash table becomes necessary. Using dynamic arrays one can increase the size of the array whenever necessary.

Algorithm 3.5.13 demonstrates the new probing function which can be used to determine the probing sequence.

Algorithm 3.5.13

```
 // The probing function pseudo code
function DHashTable.probing(u:U_TYPE);
begin
   return (hash(u)+hash2(u));
end;
```

3.5.4.1 The *indom* operator

The *indom* operator checks whether the given u maps to a `null` in then hash table array. The implementation *indom* operator will traverse the hash table similarly as to the *indom* of open hashing the only main difference is the use of the `probing` function to determine the probing order.

3.5.4.2 The *lookup* operator

The implementation *lookup* operator will traverse the hash table similarly as to the *indom* operator. The `probing` function is used to determine the

probing order. The function returns v for the given u.

3.5.4.3 The *insert* operator

The *insert* operator traverses the hash table until an unoccupied hash table entry is found and v is inserted. The `probing` function is used to to retrieve the probing sequence.

3.5.4.4 The *delete* operator

The *delete* operator traverses the hash table until an u is found. The hash table entry is marked *deleted*. NB: We cannot just simply make the entry empty because our traverse method would populate future records into the wrong place.

The side effect of marking entries to *deleted* is that eventually the hash table looses all its empty slots. Another implementation would remove the entry and move all the following elements one down in the hash order until *empty* is found. The last element's old slot is then marked as empty. This technique is preferred if there are *remove* operator is used repeatedly.
Policies on the *remove* operator:

- Mark as *deleted* : The hash table entry is marked as *deleted* .
- Move-forwards, mark as empty: The *current record* is removed and all following records are moved up. The slot from where the last was moved is set to *empty*.

A C++ implementation of double hashing is available from [Pre99].

3.6 Taxonomy of Maps

In this chapter we have defined an ADT representing maps. There are three main categories of maps implementations namely, sorted, unsorted and hashed. Given a particular problem area (for example, implementations of maps), the implementation will be derived from a common starting point. Each of the implementations appears as a vertex in the taxonomy graph, and at the root of the taxonomy we place the common starting concept of implementation (for example, maps).

Depicted by Figure 3.7 is the taxonomy of maps implementations. At the root of the taxonomy tree is highest level concept 'Maps' from which the three branches of implementations of maps is sourced. The first branch of the taxonomy (from the left) is a tree of unsorted maps implementations corresponding to the structure of Section 3.3. Similarly, the second and third branches correspond to the structures of Section 3.4 and Section 3.5 of Chapter III respectively. The systematic collection and discussion of the applicable data-structures (of maps) and their implementation was essential before we can continue to investigate graph implementations. This taxonomy allows us to **compare implementations** and determine easily what they have in **common** and where they **differ**. In the figure the abbreviations of each implementation is given together with its short meaning. Each of these implementation techniques correspond to a section in this chapter thus providing us with a **catalogue** structured implementation of maps.

3.7 Summary of Maps

In this chapter we have defined a map and introduced an ADT for maps with four main operators. We have continued with the systematic categorised im-

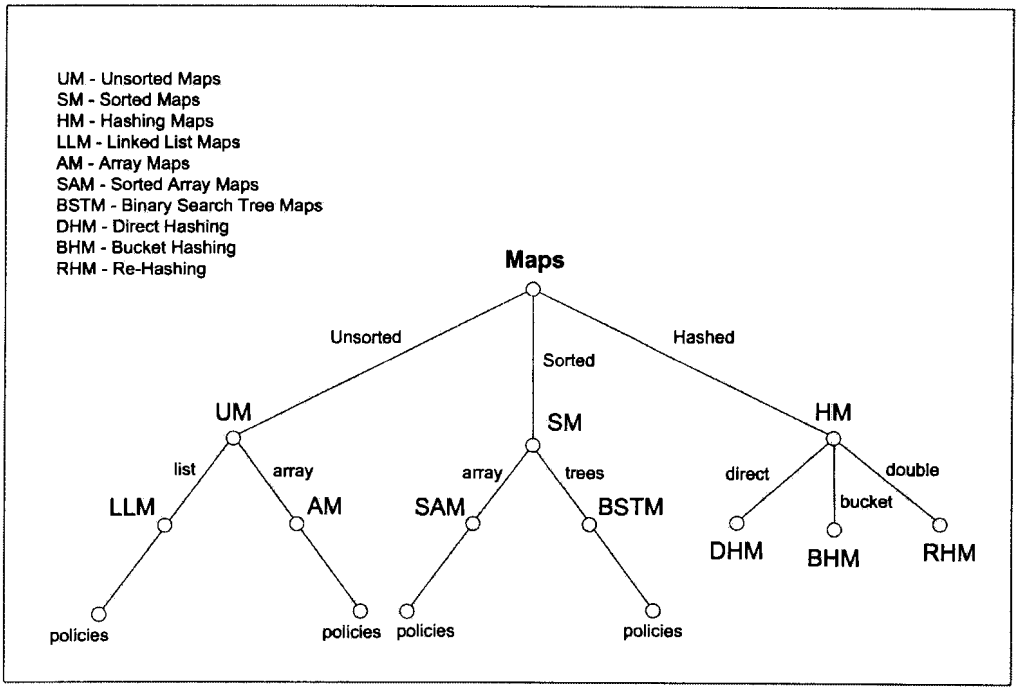


Figure 3.7: A Taxonomy of map implementations.

plementations of unsorted, sorted and hashed maps. We have then collected the implementations in a taxonomy tree which represents the different implementations categorised according to similar features.

In the next chapter (Chapter IV) we will take a similar look at the other main building block of digraphs: **sets**.

Chapter IV

Sets

4.1 General introduction to sets

In the previous chapter we have investigated closer the mapping between the two sets U and V.

Set implementations are similar to map implementation. The only difference being that while a map is a function that maps a set U to a set V, in sets there is no such V only a U.

Therefore, we cannot have a lookup operation. However we may still have the other three previously discussed operations. Let us take U to be an ADT and give it three operations:

- *indom:* Operator to query the existence of some u in the set.
- *insert:* Operator to insert u in the set.
- *remove:* Operator to remove u from the set.

Algorithm 4.1.1 represents the interface declarations of three operations.

Algorithm 4.1.1

```
//Abstract interface Declaration to the Map class in Object-Pascal
code
type Set = class
   public function indom(u:  U_TYPE): boolean ;virtual;abstract;
   public procedure insert(u:  U_TYPE); virtual;abstract;
   public procedure remove(u:  U_TYPE); virtual;abstract;
end;
```

4.2 Signatures and conditions

In all cases throughout this chapter, the following pre and post conditions hold true and are not given explicitly every time:

We take the four operations from maps chapter (Chapter III) and drop the parts of the operations which deal with V.

1. function indom(u:U_TYPE):bool

pre: true

post: f is unchanged

return: $f.indom(u) = u \in dom(f)$

2. procedure insert(u:U_TYPE)

pre: $u \notin dom(f)$

post: $u \in dom(f)$

3. procedure remove(u:U_TYPE)

pre: $u \in$ dom(f)

post: $u \notin dom(f)$

Implementation is the same as all implementations under maps except for the following:

- There is no *lookup* operation.
- `V_TYPE` is not declared under sets. Whenever a line refers to a `V_TYPE` type or container of type `V_TYPE` it should be ignored.

NB: for the rest of the chapter one can assume that the structure of the Sets Chapter is the same as the one for Maps Chapter (Chapter III) whilst taking into account the points above. Similarily, as we have discussed the Linearly Searchable Maps in the Maps Chapter earlier, we have Linearly Searchable Sets. The same holds for other data-structures.

4.3 Taxonomy of Sets

Figure 4.1 depicts the taxonomy of set implementations. Set implementations are very similar to Map implementations. All data-structures which are useful for map implementation can be used to implement sets. Consequently, the same method of taxonomy construction can be applied to this taxonomy as was applied in the previous chapter. As a result, the set taxonomy has the same amount of nodes as the map taxonomy and looks practically the same.

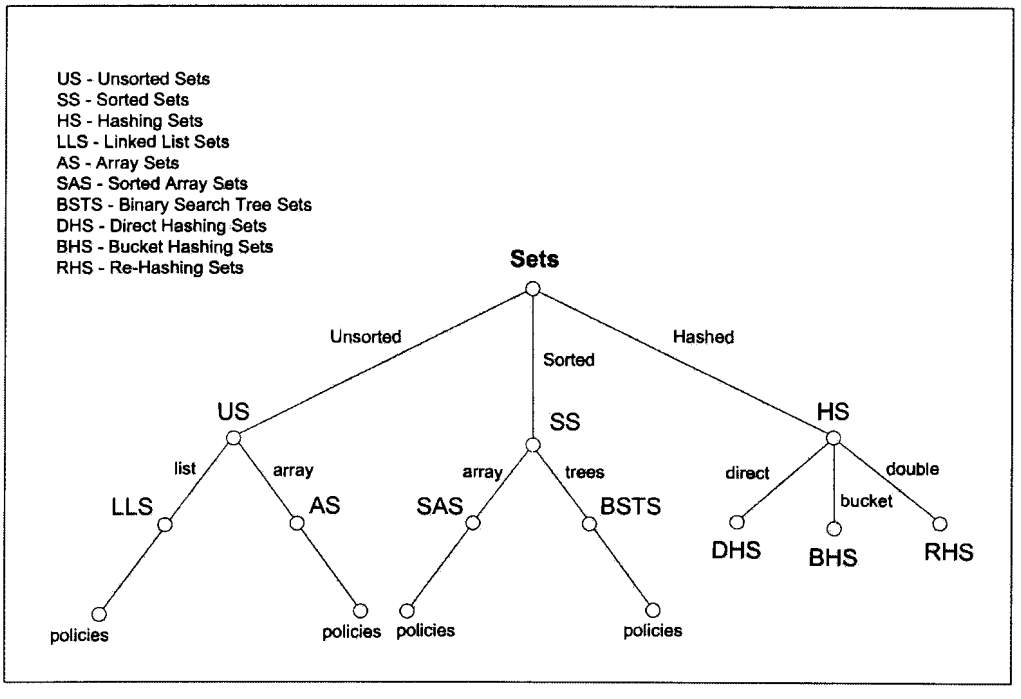


Figure 4.1: A Taxonomy of sets implementations.

Chapter V

Directed Graphs (Digraphs)

5.1 Digraph Basics

This chapter is a discussion of the digraph implementation taxonomy. The taxonomy presented at the end of this chapter is based on the previous two taxonomies namely maps and sets taxonomies. Maps and sets form the basis of all digraph implementations and the importance of the discussion of implementations in Chapter III and Chapter IV becomes evident as we place them into the new frame of reference: The Digraph Taxonomy. There are a number of other comprehensive treatments of graphs available such as the *LEDA* C++ class library [Gmb99, MN99] or the *BOOST* graph library [Ste01, JGSL01]. Both LEDA and BOOST primarily focus on the physical implementation of the C++ library interface and both libraries place emphasis on generic programming, efficiency and quality of code.

The taxonomy presented in this dissertation differs from the approach of LEDA and BOOST in the following ways: This text focuses on compilation of a taxonomy based on the systematic collection and categorisation of implementations of maps and sets (i.e. the maps and sets taxonomies).

Furthermore, this treatment of digraphs is the result of a systematic analysis, categorisation, cataloguing of digraph implementation components (together with a sketch of the correctness arguments thereof) based on the mathematical concepts of maps and sets. Another important difference is that in this dissertation we talk of triples to represent an arrow in a graph whereas BOOST presents this concept in a more generic way via using a pair for the vertices and map for the extendable parameters such as the label of the arrow.

At this point a **revision and further discussion of graphs** is in order: **graphs** are mathematical abstractions that are useful for solving many types of problems in computer science. Consequently, these abstractions must also be represented in Computer programs.

In more general terms, graphs are mathematical objects that are made of nodes connected by edges. In this text we are talking about graphs which are generally discussed in discrete mathematics books. These graphs consists of a set of nodes, and a set of edges between nodes. Furthermore, in our discussion of graphs we always refer to graphs with edges that have labels and also directions, i.e. directed graphs. These graphs have no layout information whatsoever. I.e. any two graphs are the same as long as the same nodes are connected by the same edges.

There can be several ways to depict graphs including:

- Drawing: Graphs are most commonly represented by drawings. In common practice graphs are drawn using small circles (representing the nodes) and lines or arrows (representing the edges and directions). Alphabetical labels are commonly used to mark the edges.

- Assuming a **set N of nodes**, an edge-label alphabet E, and a set of edges.

For the rest of this chapter let us say that N is a node type (normally of integer type) and E is an edge-label type (normally of string type). We can implement a graph simply by defining its type as a set of arrows. Algorithm 5.1.1 represents an example declaration of a directed graph. For this declaration to be complete we will have to declare `Arrow`.

Algorithm 5.1.1

```
// The simple declaration of a Directed Graph in object pascal
type
   DiGraph = set of Arrow;
end;
```

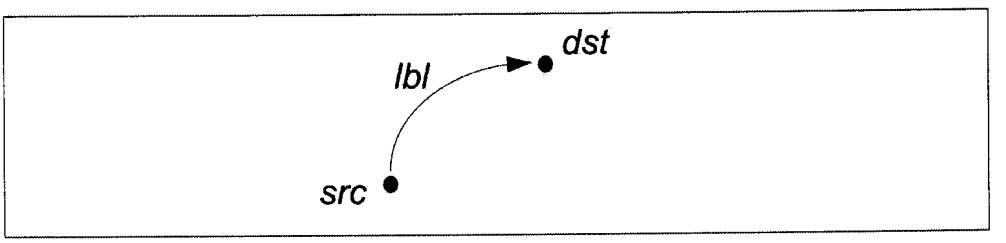


Figure 5.1: Example of an arrow (src, lbl, dst)

An arrow of a graph can be a ternary (3 way) relation on N, E, N. Figure 5.1 depicts an example of such a ternary relation of (src, lbl, dst). In basic mathematics a ternary relation is an element of $N \times E \times N$. A large number of discrete structures books discuss the topic of mathematical representation of graphs [Epp95].

In 'computer terms' an arrow is just a data-structure of an arbitrary Source-node,Label,Destination-node or *src,lbl,dst* for short. A graph is a set of arrows. In this chapter, we will focus **only** on the representations

of **directed graph arrows**. For this reason, we will distinguish between source and destination nodes, and view the edge set as any arbitrary arrow = *src,lbl,dst*. NB: the ordering of *src,lbl,dst* does matter. Let us designate *src,lbl,dst* as the beginning-point (*root*) of our taxonomy. Let us call this root α.

$$\alpha = src, lbl, dst$$

In programming α must be declared as a record of *src,lbl* and *dst* each with their corresponding types. Algorithm 5.1.2 represents the declaration of α.

Algorithm 5.1.2

```
// The simple declaration of α in object pascal
type
   TArrow = record
      src :  N;
      lbl :  E;
      dst :  N;
   end;
```

Note: In most directed graphs the collection of src and the collection dst nodes are the same.

Using simple permutations, the number of ternary relations i.e. the number of representations of *src,lbl,dst* is 3! = 3 * (3-1) * (3-2) = 6
Figure 5.2 depicts the six permutations. These six representations are isomorphic or functionally equivalent. However, they do differ in implementation and they form the basic starting point of the digraph implementation taxonomy later in the chapter.

src,lab,dest	src,dest,lab
lab,src,dest	dest,lab,src
dest,src,lab	lab,dest,src

Figure 5.2: The isomorphic representations (permutations) of *src,lbl,dst*

5.2 Graph example: The 'borrow' relation

Let us continue with a case study:

- John borrowed $15 from Peter (and therefore John owes Peter $15).
- Irene borrowed $5 from Peter.
- Peter borrowed $2 from John.
- Peter bought something for $100 and now he is short $100 from next week's rent. (He will have to get the money back for it so practically he borrowed from his own stash.)

Consider both a graph and table representation for easy visualisation of the 'Borrow' relation as depicted by Figure 5.3. The names of the borrowers are the *src* parts in our arrows. The amounts borrowed are the *lbl* parts in our arrows. The persons who lent money are the *dst* parts in our arrows. The direction of source to destination represents the direction of who owes who, i.e. for every arrow the person on the source node owes money to the person on the destination node.

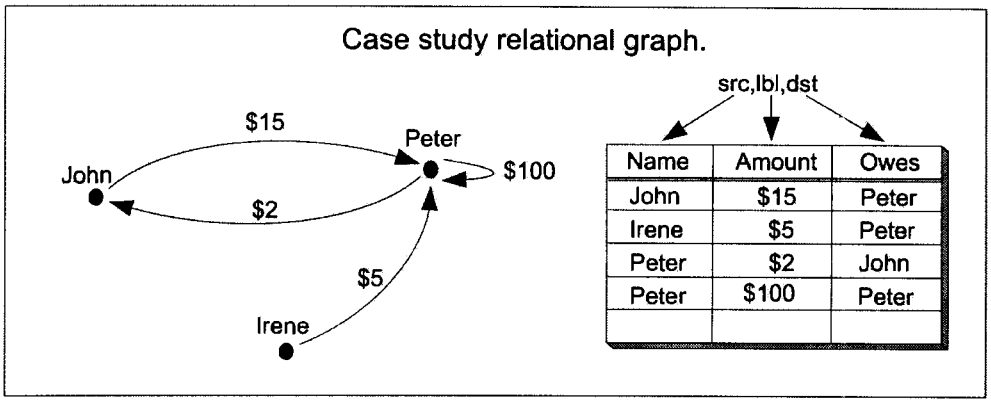


Figure 5.3: Example representation of the original case study.

5.3 The Reverse and Transpose operators

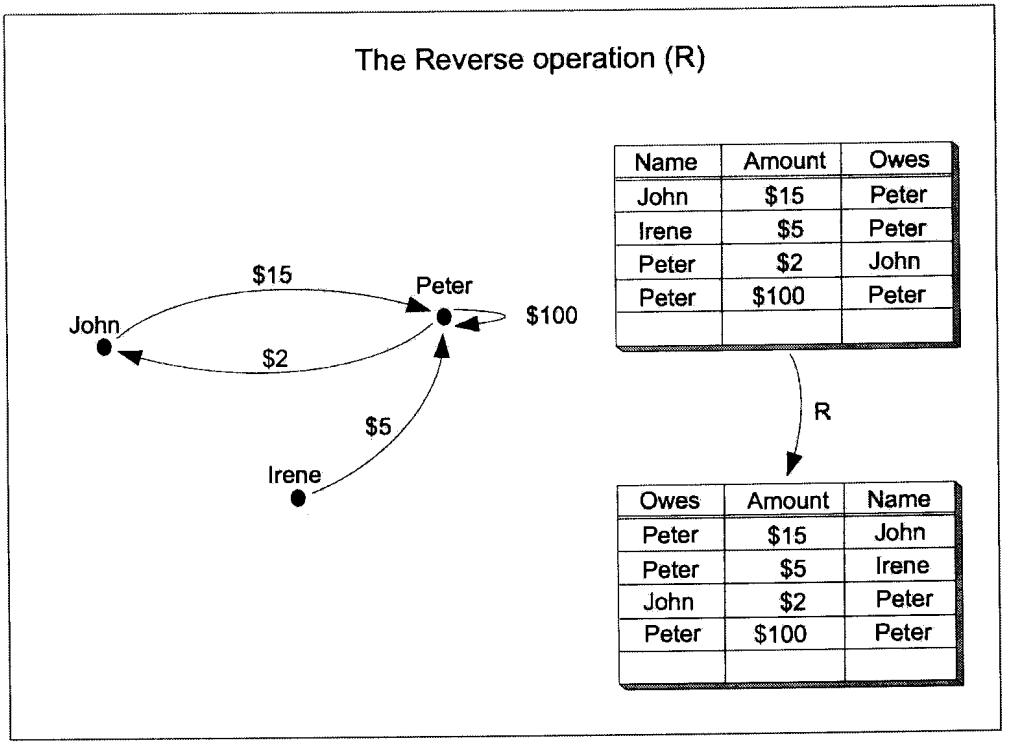
We have identified all forms that α can be represented and we now need to introduce operators to transform α into these forms. These operators will then be also derived in programming implementation form. The six triples of are depicted by Figure 5.2. The two operators are the reverse operator (R) and the transpose operator (T).

5.3.1 The Reverse operator

R maps α to *dst,lbl,src*, i.e. the R operator swaps the first and last elements. Note that $R < R < \alpha >>= \alpha$.

Figure 5.4 depicts the use of the R operator on the 'borrow' relation example. In the figure the R operator has reversed the first and last columns of the table. Note that this does not change the graph representation itself but will influence the implementation in code.

The R operator works at compile time. The implementation involves the compiler to create the record type containers of TArrow in *dst,lbl,src* order



Name	Amount	Owes
John	$15	Peter
Irene	$5	Peter
Peter	$2	John
Peter	$100	Peter

Owes	Amount	Name
Peter	$15	John
Peter	$5	Irene
John	$2	Peter
Peter	$100	Peter

Figure 5.4: Example of the reverse operator

in memory. Algorithm 5.3.1 represents the declaration of $R < \alpha >$.

Algorithm 5.3.1

```
// The R < α > data structure declaration in object pascal
type
   RTArrow = record
      dst :  N;
      lbl :  E;
      src :  N;
   end;
```

5.3.2 The Transpose operator

The T operator swaps the first and middle elements in the set. For instance T maps *src,lbl,dsc* to *lbl,src,dsc*. Note that $T < T < \alpha >>= \alpha$. Figure 5.5 depicts the use of the T operator on the 'borrow' relation example.

The T operator works at compile time. The implementation involves the compiler to create the type containers in *lbl,src,dst* order in memory. Algorithm 5.3.2 represents the declaration of $T < \alpha >$.

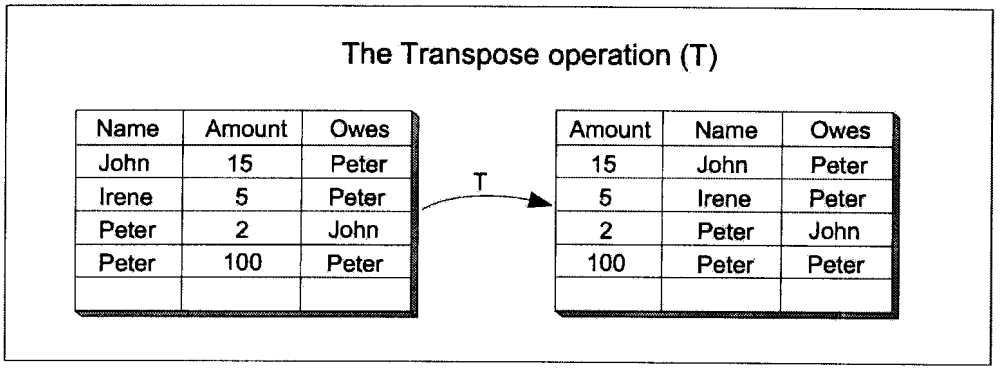


Figure 5.5: Example of the Transpose (T) operator

Algorithm 5.3.2

```
// The T < α > data structure declaration in object pascal
type
   TTArrow = record
      lbl :  E;
      src :  N;
      dst :  N;
   end;
```

5.3.3 The combined use of Transpose and Reverse

Note that both T and R functions are their own inverses[1]. Figure 5.6 depicts how the six permutations of α are obtained with the use of T and R. This figure helps us to visualise the cyclic nature of the transformations. Furthermore, Figure 5.7 depicts how the six permutations of α can obtained by applying the T and R transformations. In this figure the last representation i.e. *src,dst,lbl* can be derived from α in two equally simple transformations: applying R,T,R or T,R,T. α is the most basic building block of a digraph. The six other forms of α are similarly basic and they can be implemented using T and R. If we take α, and implement it using any data-structure from Chapter IV (Sets) we arrive to the simplest implementation of a digraph. This implementation is not a very practical one and we would have to use all three (*src,lbl,dst*) elements of α for an index. E.g. the *lookup* operator would look something like `lookup(arrow_1)` where `arrow_1` is of a combined data-type of *src,lbl,dst*.

Therefore, it is logical to consider the six representations of α as a good basis of all digraph implementations and later in this chapter we will use these as the root of the digraph implementation taxonomy.

In Figure 5.8 we can observe the same T and R transformation of α with reference to the 'Borrow' relation example. In this example it is easy to see that the content of the tables remain the same and only the order of the columns changed. Algorithm 5.3.3 represents the record declaration of $T < R < \alpha >>$, $R < T < \alpha >>$ and $R < T < R < \alpha >>>$.

Algorithm 5.3.3

```
// The T < R < α >> declaration in object pascal
```

[1]Applying the same function twice will result in the original set.

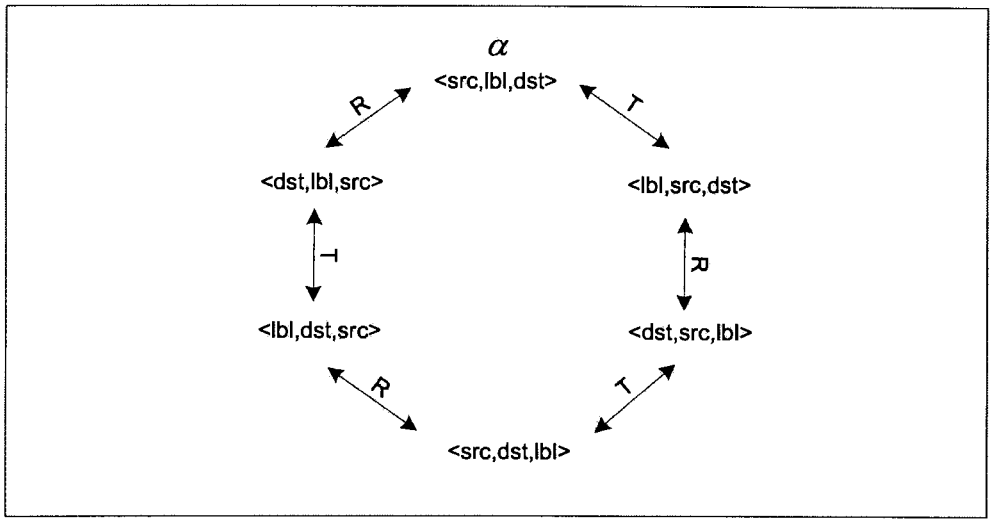


Figure 5.6: The six different representations of α obtained using T and R

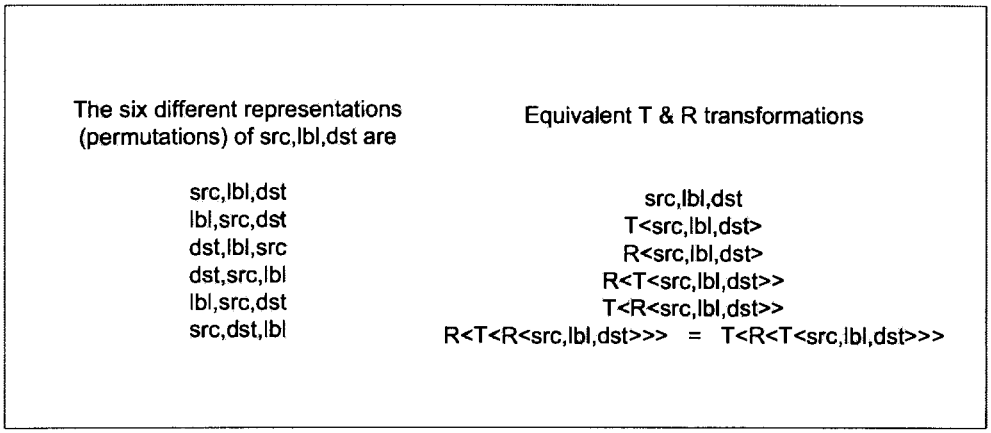


Figure 5.7: The six representations α and their equivalent transformations of α using T and R

```
type
   TRTArrow = record
      lbl :  E;
      dst :  N;
```

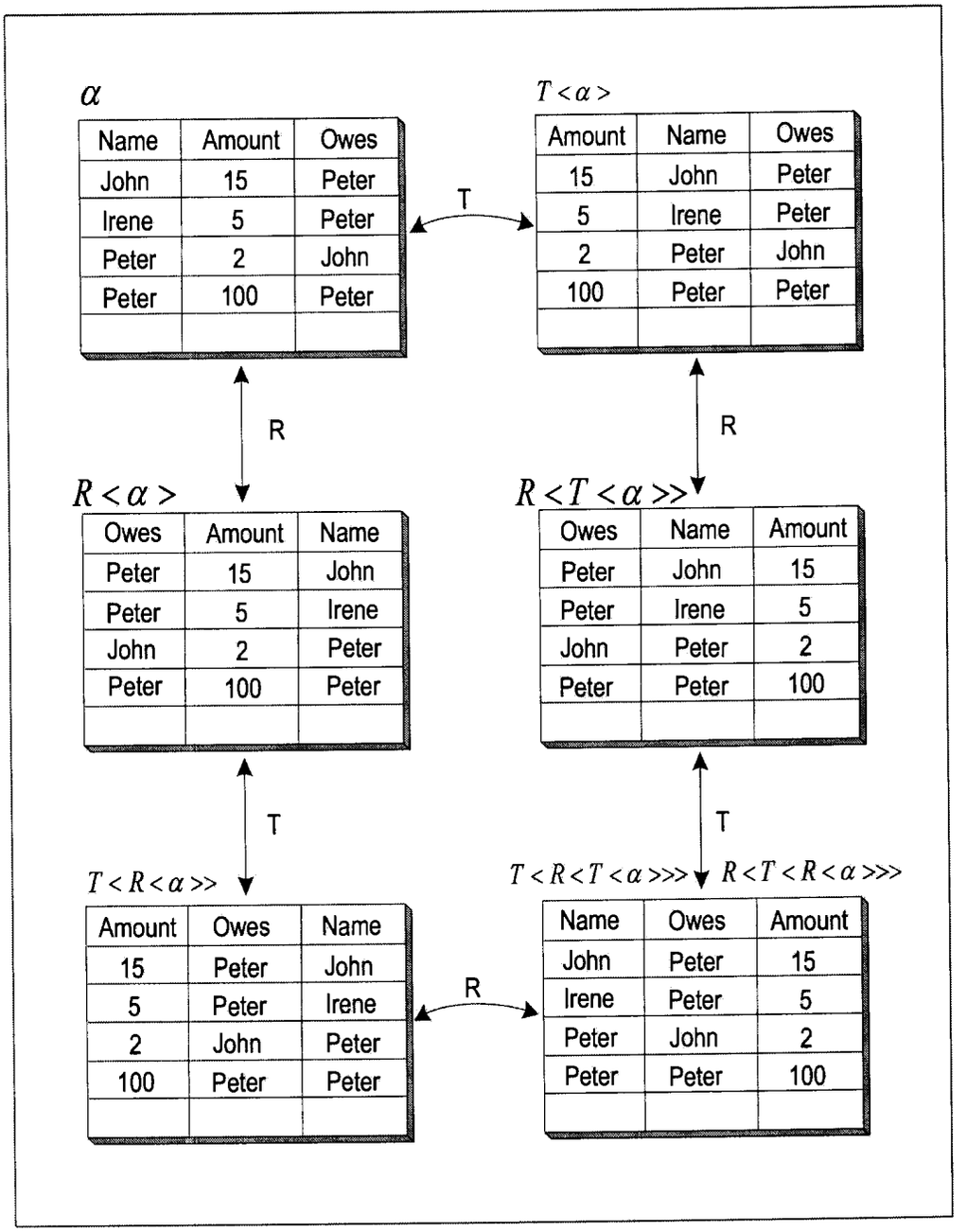


Figure 5.8: The transformations of α using T and R operators.

```
      src :  N;
   end;
NoInd// The R < T < α >> declaration in object pascal
type
   RTTArrow = record
      dst :  N;
      src :  N;
      lbl :  E;
   end;
NoInd// The R < T < R < α >>> declaration in object pascal
type
   RTRTArrow = record
      src :  N;
      dst :  N;
      lbl :  E;
   end;
```

5.4 Other operators on sets of triples

In order for us to make use of the implementation in Chapter III (Maps) we need a *binary* relation. We will have to introduce a new operator to partition a *ternary* relation into a *binary* relation.

5.4.1 The Left Associate operator

The Left Associate (LA) operator takes a record type with three fields and groups the two leftmost fields together, hence the term 'Left Associate'. E.g. $LA < \alpha >=(src,lbl),dst$. Therefore we could say that the LA operator transforms a *ternary* relation into a *binary* relation. Figure 5.9 depicts the LA

transformation of α with reference to the 'Borrow' relation example. In digraph implementation using a set (from Chapter IV) of $LA < \alpha >$ the *src,lbl* elements of α would together represent the index (or the U).

Similarly to T and R, the LA operator is a also a compile time operator. The LA operator takes a record type with three fields and groups the first two together. The implementation creates the first two fields as a record and then uses this record together with the last field to create a record type for the arrow. Algorithm 5.4.1 represents the declaration of $LA < \alpha >$ record.

Algorithm 5.4.1

```
// The LA < α > operator data structure declaration in object
pascal
type
   TLA = record
      src:N;
      lbl:E;
   end;
   LATArrow = record
      LA:TLA;
      dst:N;
   end;
```

5.4.2 The Right Associate operator

Similarly to the LA operator, the Right Associate RA operator also transforms a *ternary* relation into a *binary* relation. The difference is that RA grouping takes a record type with three fields and groups the two rightmost fields together, hence the term 'Right Associate'. E.g. $LA < \alpha >$=*src,(lbl,dst)*.

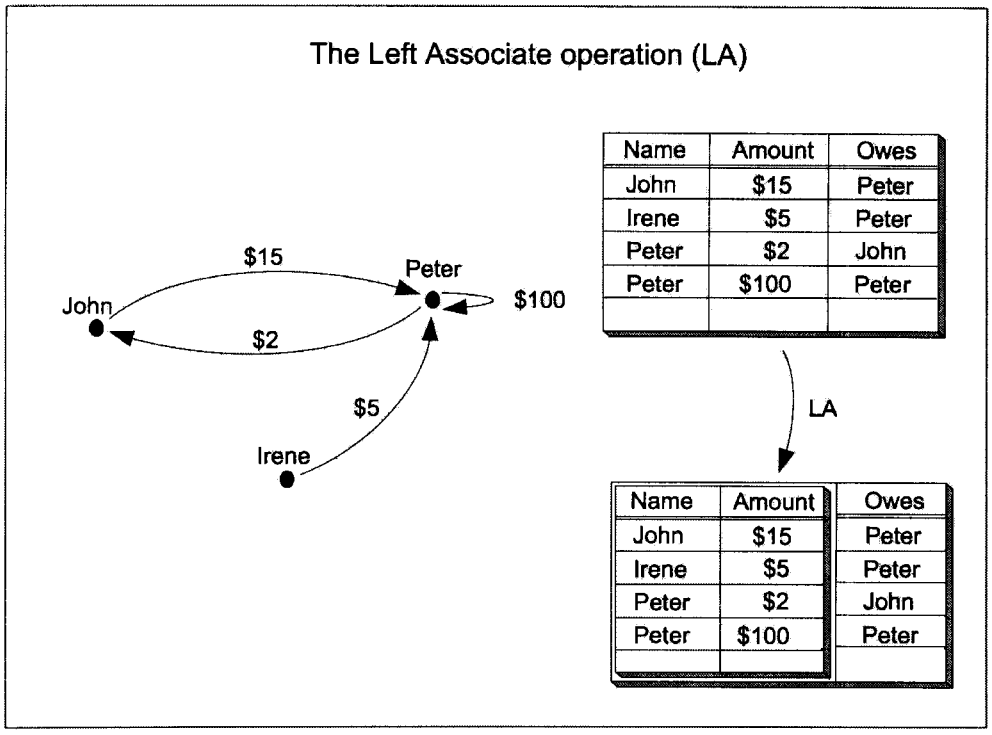


Figure 5.9: The example transformations of α using the LA operator.

The RA operator is a also a compile time operator. Figure 5.10 depicts the RA transformation of α with reference to the 'Borrow' relation example.

The implementation creates the first two fields as a record and then uses this record together with the last field to create a record type for the arrow. Algorithm 5.4.2 represents the declaration of $RA < \alpha >$ record.

Algorithm 5.4.2

```
// The RA < α > data structure declaration in object pascal
type
   TRA = record
      lbl:real;
      dst:string;
```

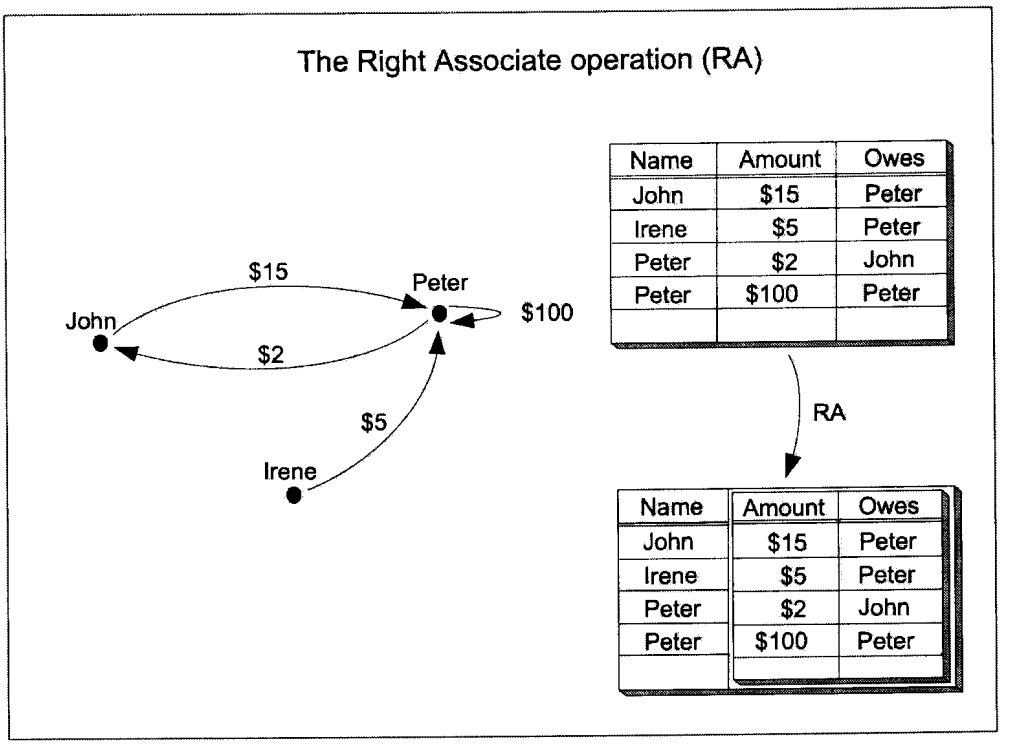


Figure 5.10: The example transformation of α using the RA oprator.

```
end;
RATArrow = record
   src:string;
   RA:TRA;
end;
```

5.4.3 The Map (M) operator

The M operator maps a binary (NB: not ternary) relation to a function. e.g. Mapping *src,(lbl,dst)* to $src \mapsto (lbl,dst)$. Since the M operator creates a map from binary relation, the M operator can only be applied on a α

after it has been partitioned by the LA or RA operators. The M operator is also a compile time operator. The implementation declares the first two fields as a record and then uses this record together with the last field to create the record type for the arrow. Algorithms 5.4.3,5.4.4 and 5.4.5 represent the declaration of $M < RA < \alpha >>$, $M < LA < \alpha >>$ and $M < M < RA < \alpha >>>$ records respectively. The programming implementations of the M operator involves a pointer from the one side of the binary relation to the other, this is easiest to observe in Algorithm 5.4.5 in which $RA < \alpha >$ is mapped twice with two pointers. In Figure 5.12 we can see an example representation of the binary relation $RA < \alpha >$ being mapped into a function. The M operator can be implemented using any of the implementations discussed in Chapter III.

Algorithm 5.4.3

```
// The M < RA < α >> operator data structure declaration in
object pascal
Type
   MRATArrow = Record
      src :  N;
      Link_1 :  ^Tlbldst;
   End;
   Tlbldst = Set
   Record
      lbl:  E;
      dst:  N;
   End;
```

Figure 5.13 depicts the use of the M on $M < RA < \alpha >$ using on the 'borrow' relation example. With the help of the table form representation

we can get an idea of how the implementation will be structured. After the M operator the table is broken up into three separate columns. The first column can be represented by a single map from chapter III (e.g. BSM). The second column is divided up into four partitions each of which can be represented by a map structure. Any of the partitions in the third column can be represented by one of the set representations discussed in Chapter IV.

Algorithm 5.4.4

```
// The M < LA < α >> operator data structure declaration in
object pascal
Type
   MLATArrow = Record
      src :  N;
      lbl:  E;
      Link_1 :  ^Tdst;
   End;
   Tdst = Set
   Record
      dst :  N;
   End;
```

Algorithm 5.4.5

```
// The M < M < RA < α >>> operator data structure declaration
in object pascal
Type
   MMRATArrow = Record
      src :  N;
      Link_1 :  ^Tlbl;
   End;
```

```
Tlbl = Record
   lbl :  E;
   Link_2 :  ^Tdst;
End;
Tdst = Set
Record
   dst :  N;
End;
```

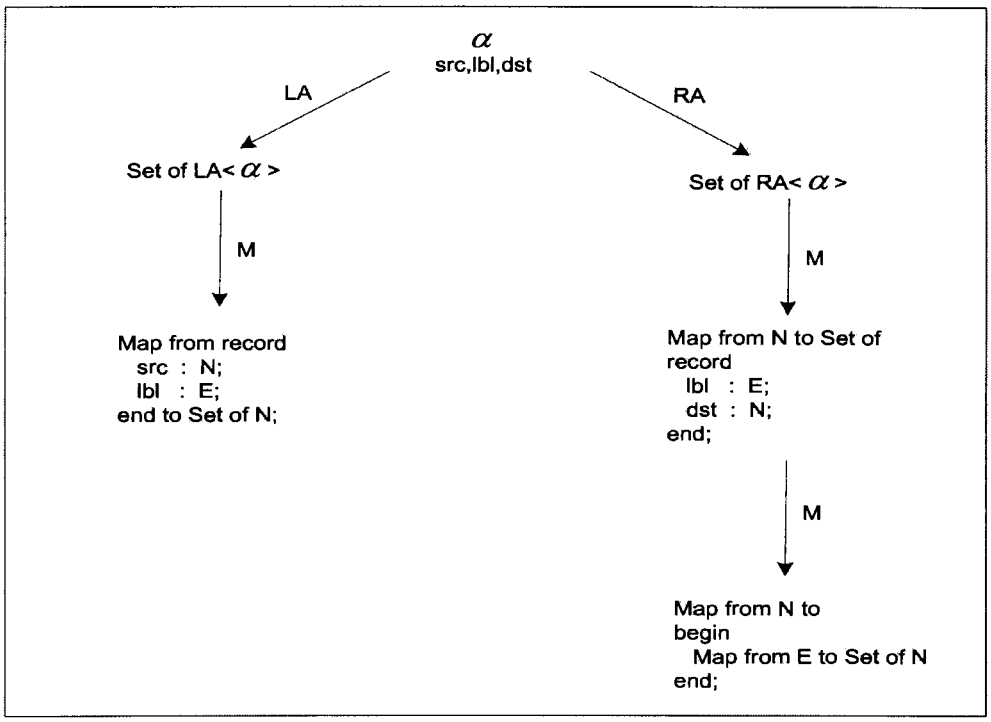


Figure 5.11: A transformation example of $< \alpha >$ using LA, RA and M operators.

In Figure 5.11 we have the six different representations of α being transformed into the six different relations using RA, LA and M.

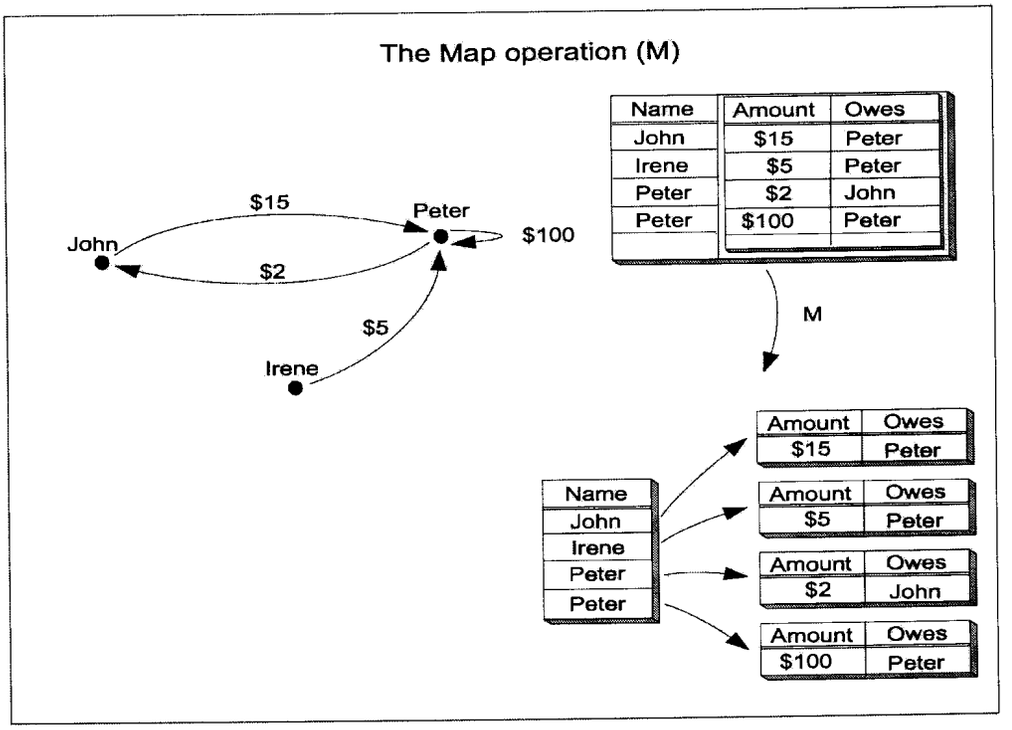


Figure 5.12: A transformation example of $RA < \alpha >$ using the M operator.

In Figure 5.14 we have the source binary relation $LA < \alpha >$ mapped to a function. Note: LA or RA operators are necessary to form a binary relation before M can be applied. Using M, the binary relations are then mapped to a function. Note that instead of starting with α we could have started with any of the original six permutations.

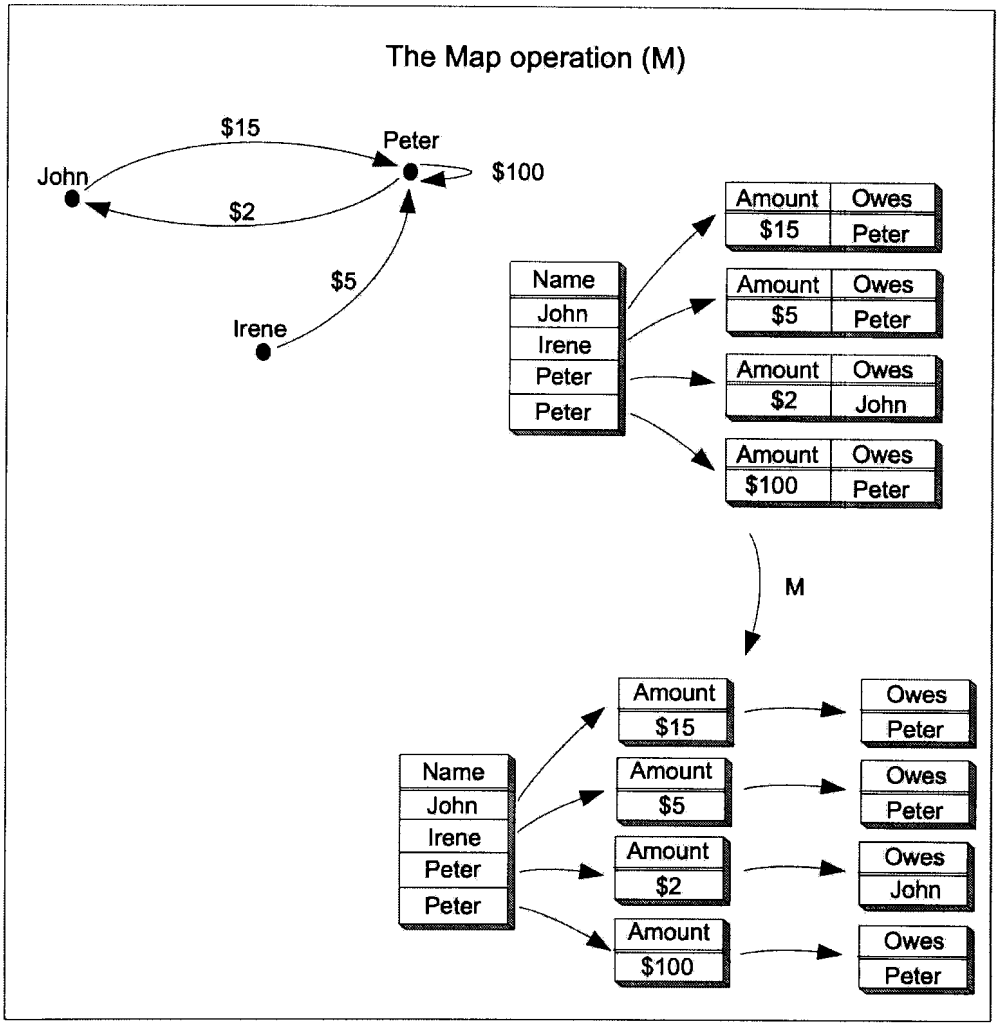


Figure 5.13: A transformation example of $M < RA < \alpha >$ using the M operator.

5.5 Digraph Taxonomy

Now that we have discussed all required operators and structures required to represent digraphs we can continue with building up an implementation taxonomy. This taxonomy has no closely related work associated with it and

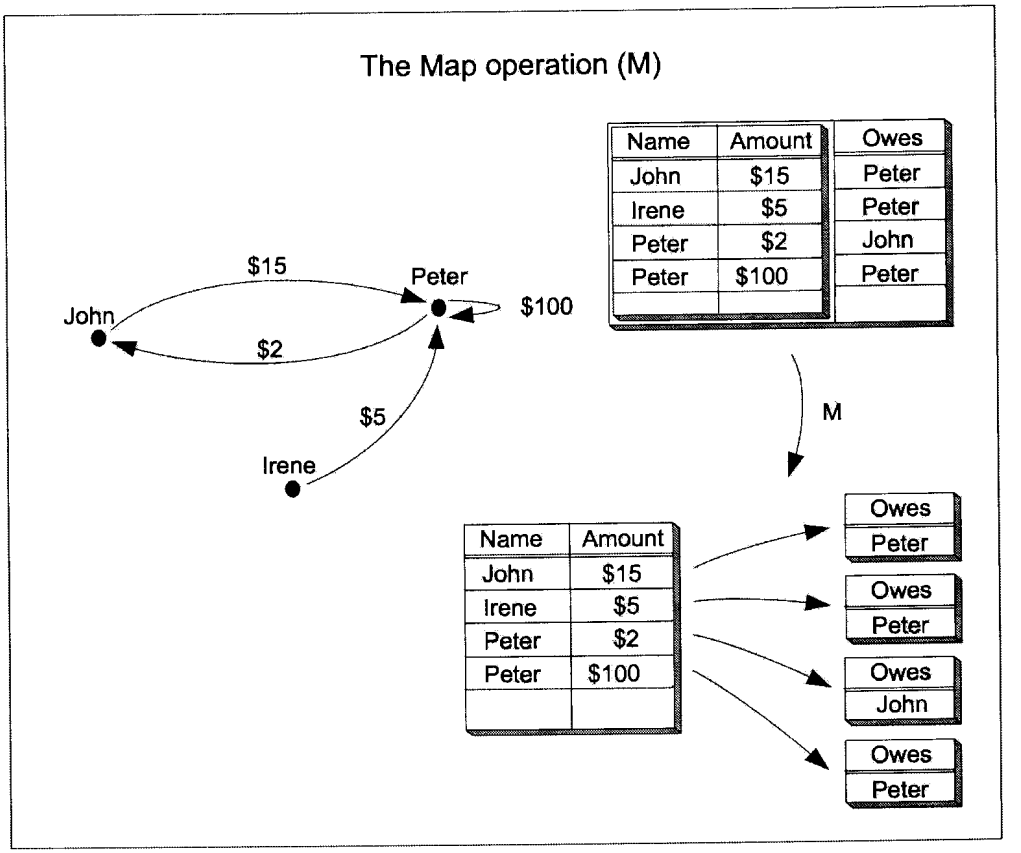


Figure 5.14: A transformation example of $LA < \alpha >$ using the M operator.

the **core idea is presented as new**. The root of the taxonomy is normally represented by the simplest concept (in our case the simplest digraph implementation). The simplest digraph implementation we can consider is a set of arrows in the form of α. Similarly, the six permutations of α are on the same level of implementation as α and therefore we can make all six be the circular root of the taxonomy tree. We can show the differences between the six root nodes by labeling the links between them with the transformation required to get from one to the other (R and T). We can use the word **set** to represent the set implementations and use a right arrow ($\rightarrow$) to represent

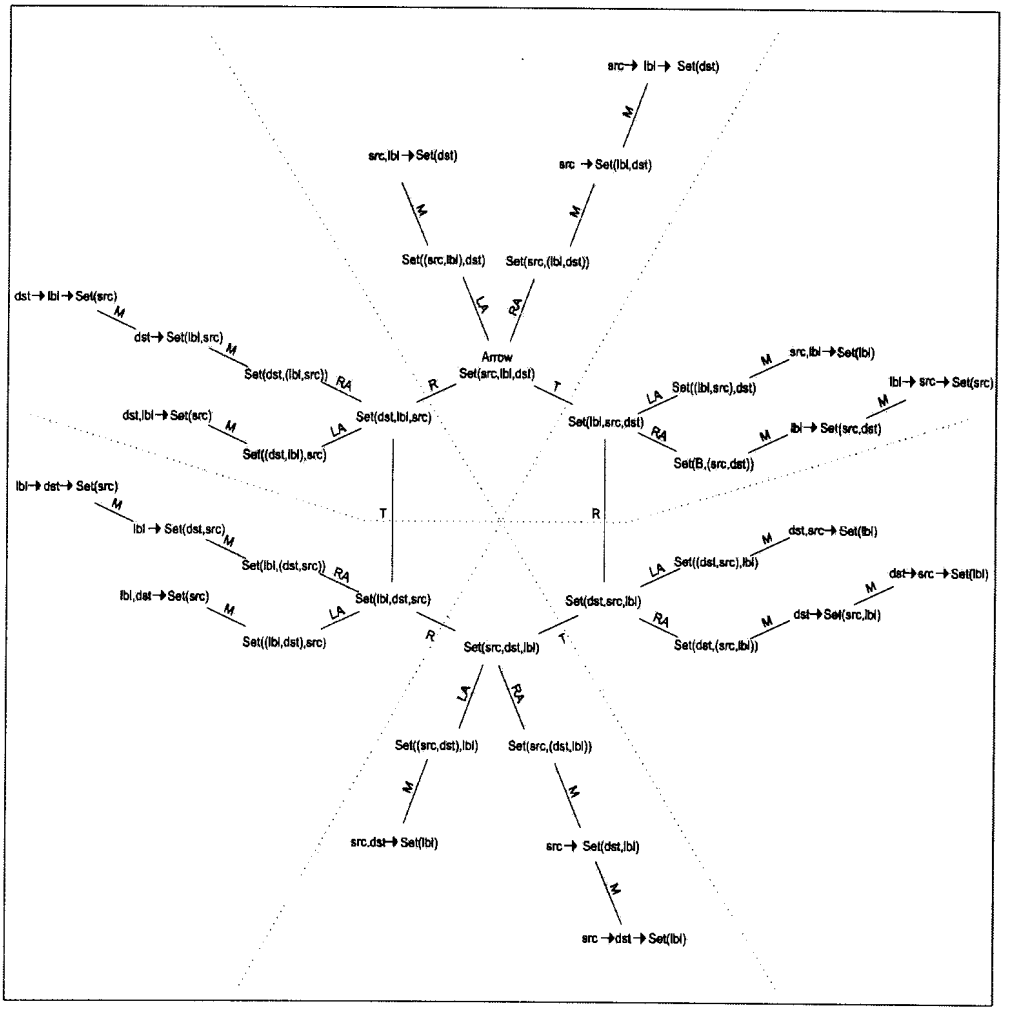


Figure 5.15: Taxonomy of Digraph implementations. Note: → is used as a shorthand to represent the Map relation.

a map.

Figure 5.15 depicts the digraph taxonomy. From the six root nodes we can only continue to other implementations by applying the *RA* and *LA* operations after which we can immediately apply the *M* operator. To make the six branches of this circular rooted tree more obvious a star shape separator

line has been added between the branches. Note that all six implementation branches are identical except for the order of the base elements *src,lbl,dst*.

5.6 Digraph Taxonomy Analysis

As the **final conclusion** to this chapter we can bring together all three taxonomies and investigate the possible digraph implementations:

With the help of the taxonomy of Figure 5.16 we can quantify the number of different sets implementations: LLS, AS, SAS, BSTS, DHS, BHS, RHS. Discounting the policies, we can count seven different major implementations.

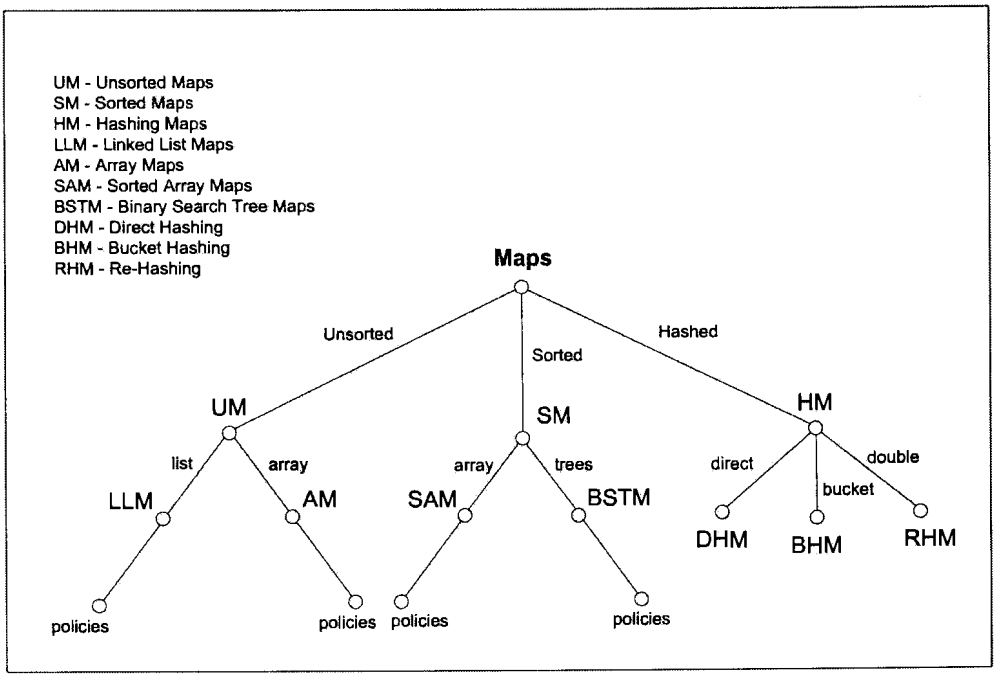


Figure 5.16: A Taxonomy of map implementations (revisited).

Similarly, there are seven major implementations of maps observable in Figure 5.16: LLM, AM, SAM, BSTM, DHM, BHM, RHM.

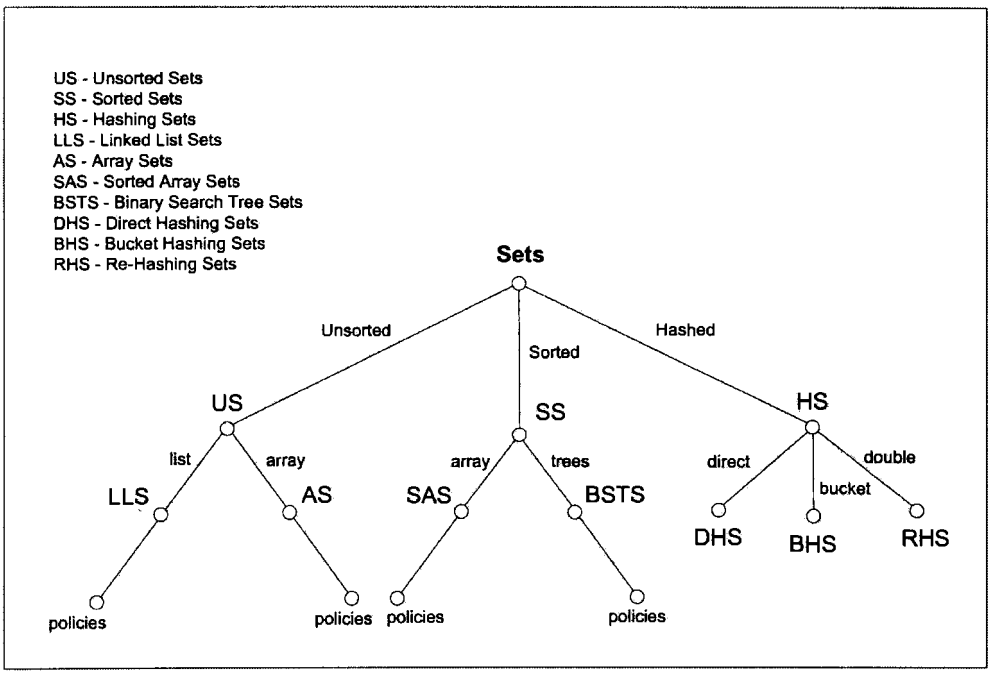


Figure 5.17: A Taxonomy of sets implementations (revisited).

With the help of the taxonomy of Figure 5.15 it is easy to visualise that R and T operators give six variations of basic implementation of α. Furthermore, because of the symmetry between the branches of Figure 5.15 we can consider the number of digraph implementation on one branch to be the same on the other branches. In Figure 5.18 we can see the digraph taxonomy of Figure 5.15 with some additions derived from the Figure: The number of maps and sets required for the implementations.

There are six starting points of the Digraph taxonomy each with the following implementations:

- Three Different Sets implementations i.e. $= 3 * 7 = 21$
- Two Different implementations that are mixture of a single set and a

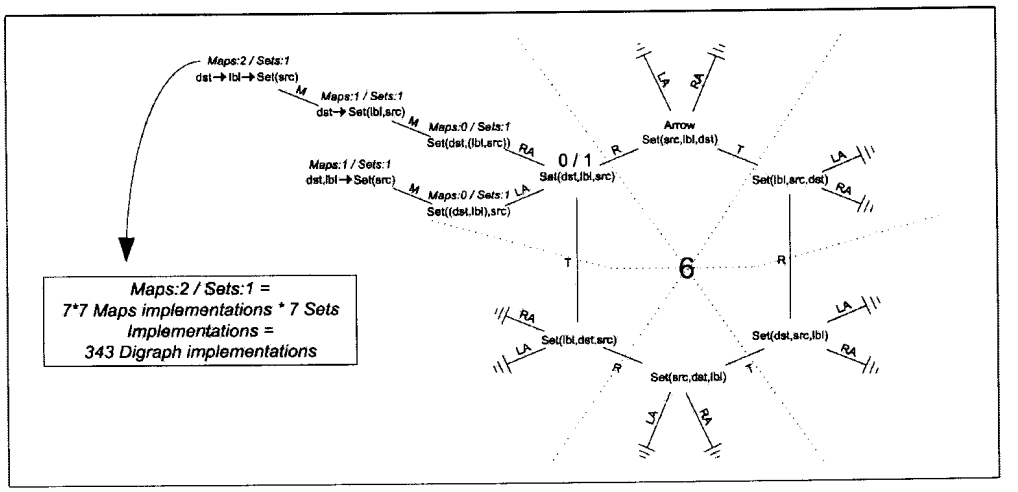


Figure 5.18: Portion of the digraph implementation taxonomy with additions for quantifying.

single map implementation, i.e. $2 * 7 * 7 = 98$

- One implementation that is a mixture of 2 Maps and a single set implementation, i.e. $7 * 7 * 7 = 343$

Because of the six starting points there are $6 * 462 = 2772$ different major digraph implementations within our digraph taxonomy reference discounting policies. Please note that this number is by no means a representation of all possible digraph implementations that may exist. The number of possible implementations is practically infinite. This example demonstrates how easily one can derive useful analytical information from the taxonomies. Without these three taxonomies it would be much harder to analyse digraph implementations in such low level detail and accuracy.

5.7 Summary of Digraphs

In this chapter we have defined digraphs and their program declarations. We have looked at the atomic component of a digraph and manipulated it into all its possible configurations with operators that we have introduced for this purpose. Then we derived a taxonomy and integrated our foundings with the taxonomies in the previous chapters. We shall now continue with the final conclusions of this dissertation.

Chapter VI

Conclusion

The conclusions of this dissertation are summarised in this chapter.

In Chapter III we have defined a map and introduced an ADT for maps with four main operators. We have continued with the systematic, categorised and uniform implementations of unsorted, sorted and hashed maps. We have then collected the implementations in a taxonomy tree which represents the different implementations categorised according to similar features. Similarly, in Chapter IV we have drawn up a taxonomy of set implementations. As a result, we can now see how the map and set implementations are related to each other. In Chapter V we investigated the implementations of the basic element of a graph (an arrow) and defined basic operators for the manipulation of the arrows into all of its possible representations. Built on the the taxonomies of the previous chapters, in chapter V we derived the taxonomy of digraph implementations.

6.1 General Conclusions

The general conclusions of this dissertation are:

- The first two taxonomies presented in this dissertation cover areas of computer science which have been discussed in data-structures books for about three decades. A large effort was put into recreating the implementations in a uniform representation from different sources, different language and often from scratch. With the help of re-working an example of each of the implementations a number of hidden implementation possibilities were revealed and discussed.

- The taxonomies were built from a bottom up approach, with the consideration of the most essential components, variations and common features. The common features were then collected and were be presented together in a taxonomy.

- It does not not appear that taxonomies could be constructed for all digraphs implementations. The implementations treated in this dissertation are the major forms of implementations of digraphs. However, the implementations were structured in a manner where we could make a clear distinction between what we consider two different implementations within our own frame of reference.

- The concluding digraph taxonomy integrated the work laid out by the maps and sets taxonomies. The digraph taxonomy's root were the six simplest graph implementations. The links between two the nodes of the taxonomy represented the transformations that distinguished two implementations. In contrast to this the maps and sets taxonomies used representation invariants as the main basis for its categorisation. These two different approaches proved to be quite practical. It is was possible to draw up a concrete taxonomy of digraphs based on the map and set taxonomies because both maps and sets were based on the

well-known mathematical concepts of maps and sets.

6.2 Final word

At first glance, it may appear that the taxonomies only serve to classify existing digraphs. This assumption would be highly erroneous because most of the implementations derived from the taxonomy are not really useful and probably do not exist. By combining the taxonomy details in new ways, or by making use of techniques developed during the taxonomisation, it is possible to construct new digraphs. The aim of the taxonomy was completeness and practicality. Although practical implementation are easily identifiable directly from the taxonomy, some practical elements used in digraph programming such as iterators were not discussed because they were not required in this model. These practical elements were missed to certain extent during the compilation of this dissertation. An open problem of incorporating some more practical elements used in some available C++ models still remains and parts of the solution are currently in the works [Koo03]. As a final conclusion, we could state that a fine balance of theoretical and practical computer science is probably the most important aspect of useful scientific research.

Bibliography

[Adl94] Leonard Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1023, 1994.

[Alt96] Irfan Altas. Taxonomy of computer architectures. [Online]. `http://athene.csu.edu.au/~ialtas/module1/section_3.html`, 1996.

[AVL62] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklay SSSR*, 3:1259–1263, 1962.

[C.A61] C.A.R.Hoare. Quicksort. *Comm. of the ACM*, (4):321, 1961.

[DNU00] Hiroaki Nishino Dahlan Nariman and Kouichi Utsumiya. Collaborative and interactive urban planning simulation system. [Online]. Available `http://www.vsmm.org/vsmm2000/review/papers/734-678-5.PDF`, 2000.

[Enc99] Encyclopedia. Encyclopedia britannica 1999. CD-ROM, 1999.

[Epp95] Susanna S. Epp. *Discrete Mathematics with Applications.* Brooks/Cole, second edition, 1995.

[FCV98] Paul Helman Frank Carrano and Robert Veroff. *Data Abstraction and Problem Solving with C++*. Addison-Wesley, second edition, 1998.

[Gen02] Genome project website. [Online]. Available: `http://www.ornl.gov/TechResources/Human_Genome/`, 1990-2002.

[Gmb99] Algorithmic Solutions Software GmbH. Leda – library of efficient data types and algorithms. [Online]. Available: `http://www.algorithmic-solutions.com/enleda.htm`, 1999.

[GS93] David Gries and Fred B Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, first edition, 1993.

[How95] John D. Howard. A taxonomy of computer and network attacks. [Online]. `http://www.cert.org/research/JHThesis/Chapter6.html`, 1989-1995.

[JGSL01] Lie-Quan Lee Jeremy G. Siek and Andrew Lumsdaine. *Boost Graph Library, The: User Guide and Reference Manual*. Addison-Wesley, first edition, 2001.

[Joh99] Tommi Johtela. Delphi persistent container library. [Online]. Available: `http://www.cs.utu.fi/tjohtela/software.htm`, 1999.

[Koo03] Theodore Koopman. A generative graph toolkit for c++. MSc dissertation, University of Pretoria, to appear, 2003.

[Mey98] Bertrand Meyer. The importance of the class invariant. [Online]. Available: `http://www.elj.com/eiffel/bm/invariants/`, 1998.

[MN99] K. Mehlhorn and St. Nher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, [Online]. Preprint available: `http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html`, first edition, 1999.

[Mor98] John Morris. Linked list C++ implementation. [Online]. Available: `http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/source/collection_ll.c`, 1998.

[Pre99] Bruno R. Preiss. *Data Structures and Algorithms with Object Oriented Design Patterns in C++.* John Wiley and Sons, Inc., first edition, 1999.

[Sta98] Thomas A. Standish. *Data Structures in JAVA.* Addison-Wesley, first edition, 1998.

[Ste01] Alexander Stepanov. Boost graph library. [Online]. Available: `http://www.boost.org/libs/graph/doc/table_of_contents.html`, 2000-2001.

[Tec00] Techtarget. Taxonomy. [Online]. Available: `http://whatis.techtarget.com` , Search-string:'taxonomy', 2000.

[Wag00] Ben Waggoner. Biography of linnaeus. [Online]. Available: `http://www.ucmp.berkeley.edu/history/linnaeus.html`, 2000.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, September 1995.