# SPLAYING IN THE IMPLEMENTATION OF THE LINK/CUT TREE OPERATIONS USED IN SOLVING THE MAXIMUM FLOW PROBLEM

By

ZIAD ISHAQ RIDA

Bachelor of Science

in Electrical Engineering

Oklahoma State University

Stillwater, Oklahoma

1984

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1986

SPLAYING IN THE IMPLEMENTATION OF

THE LINK/CUT TREE OPERATIONS

USED IN SOLVING THE

MAXIMUM FLOW

PROBLEM

Thesis Approved:

Donald D Fisher

Thesis Adviser

D. E. Hedrick

Donald W Grace

Norman N. Durham

Dean of Graduate College

# PREFACE

This study is concerned with the analysis of the splaying operation, which moves a certain node in a tree all the way up the tree to become the new root. Splaying is an operation on a self-adjusting data structure called the Splay tree. New methods are suggested to perform top-down restructuring of splay trees when moving a given node to the root providing faster and simpler algorithms. The splaying operation is used in implementing the link/cut tree operations thus solving the linking and cutting trees problem in an amortized time bound of $O(\log_2 n)$ per tree operation. Finally, the use of the link/cut tree in solving the maximum flow problem giving an algorithm of $O(n*m*\log_2 n)$ is illustrated to give a precise idea of how this powerful data structure can be applied.

The author would like to express his appreciation to his thesis advisor, Dr. D. D. Fisher, for his precious help and guidance throughout the study. Appreciation is also expressed to the other committee members, Dr. G. H. Hedrick and Dr. Grace for their suggestions and advice in completing this study.

Finally, special gratitude is expressed to my family and friends for their understanding, support, and encouragement at all times.

TABLE OF CONTENTS

# LISTS OF TABLES

## LIST OF FIGURES

Figure page

## LIST OF SYMBOLS

| | |
|---|---|
| E | Set of edges in a graph G |
| f | Flow in a graph G |
| G | A graph with V vertices and E edges defined by G=(V,E) |
| h:l | high:low frequency of access assigned initially to the items in a tree. (See tables I-VI) |
| i | A reference to an item in a node |
| L | Level graph |
| n | Number of different values generated by a random number generator (See Tables I-VI) |
| na | Number of accesses to a tree (See tables I-VI) |
| p% | The percent of values that have high frequency of access (See table I-VI) |
| Ph | Probability to access an item with a high frequency of access (See table V) |
| $P_i$ | A set of vertices representing a path (See table VII) |
| Pl | Probability to access an item with a low frequency of access |
| R | Residual graph |
| r | The range of the n values generated is 1-r. (See tables I-VI) |
| TPh | Total probability to access any of the high frequency items |
| V | Set of vertices in a Graph G |
| v | A reference to a vertex |
| x | A reference to a node |

# CHAPTER I

## INTRODUCTION

Static data structures may be both time and space efficient for some applications but for most practical applications a dynamically changing data structure is needed. Self-adjusting data structures refer to data structures that provide the advantage of changing dynamically with the changing patterns of access.

Self-adjusting data structures provide ways to achieve amortized efficiency. Amortization is the average time per operation on a given data structure over a worst case sequence of operations. Amortized efficiency is the objective in applications where more than one operation is performed at one time.

One type of self-adjusting data structure is the splay tree. A splay tree is a binary tree over which the splaying operation is defined. Splaying involves moving a node up the tree until it becomes at the root position.

The splay tree, even when compared with balanced and optimum trees, achieves better amortized efficiency. The splay tree adjusts itself to suit the initial frequencies of access distribution of the items in the tree. It will readjust itself if the frequencies of access of some or all the items change dynamically with time. The idea of

self-adjusting can be applied to data structures other than trees. For example, there exists self-adjusting lists and heaps.

Splay trees, as is the case with all self-adjusting data structures, require too much restructuring. Methods to move a node to the root position were suggested. Among these were the move-to-root and splaying methods. Splaying can be done both bottom-up and top-down. The top-down version is harder to understand and thus more difficult to implement.

The move-to-root operation is a bottom-up process. It is not possible to do this operation in a top-down fashion using simple rotations. Since top-down restructuring in this case is more time and space efficient it becomes convenient to be able to use it.

In this research new methods to move a node to the root are suggested and tested. These new methods simplify the splaying operation but still provide the same efficiency enjoyed by the original methods, namely move-to-root and splaying.

Splay trees have many applications, one application is in link/cut trees which are used in many network applications like the maximum and minimum flow problems. A link/cut tree is a data structure with a set of operations defined to operate upon it. Among these operations are the link and cut operations. These two operations dynamically modify the link/cut tree by adding and deleting edges to vertices in the tree, respectively. There are few different methods to implement the link/cut tree operations. One

method, is to divide each link/cut tree into a set of paths. A set of operations is defined over this set of paths. These path operations are used to implement the link/cut operations. This method provides an amortized efficiency of $O(\log_2 n)$ per operation and an amortized efficiency of $O(m*\log_2 n)$ for a sequence of m worst case operations over the link/cut tree. This time bound is significantly better than that obtained by the more basic methods used to solve the linking and cutting trees problem.

The choice of the method to be used to implement the link/cut tree operations is relevant to the efficiency of the each method. In this research, the link/cut tree operations are implemented and a comparison is conducted between the different methods for implementing these operations. The major application of the link/cut tree is in the implementation of the maximum flow problem. An illustrated presentation of the use of link/cut tree to find a maximum flow is done in this research giving an idea of how this powerful data structure can be used in practice. Using the link/cut tree in the maximum flow problem implementation produces an algorithm with a time bound better than the fastest previously known algorithm.

# CHAPTER II

# LITERATURE REVIEW

## Introduction

The splay tree is a type of self-adjusting data structure which dynamically adjusts itself with accesses. Splaying is an operation that is used on the splay trees to move a node to the root position. The link/cut tree problem can be solved by finding a systematic method to implement the operations defined over the link/cut tree data structure. Splaying is used in the implementation of the link/cut tree operations. This improves the amortized efficiency of the link/cut tree. The link/cut tree has several applications, an important one is in the solution of maximum and minimum flow problems in networks. The work presented here was mainly done by R. E. Tarjan and D. D. Sleator (1,14,15).

## Self-Adjusting Data Structures

Self-adjusting data structures like any other type of data structures have both advantages and disadvantages (2,4,8).

Advantages:

1. They require less storage and provide a faster running time where amortization is of interest.

2. They tend to adjust dynamically according to the changes in the access frequencies of the items involved.

3. Their maintenance is simple due to the simple structure they usually have.

Disadvantages:

1. They require more readjusting and restructuring than usual data structures like balanced trees because restructuring takes place not only after adding or deleting items but also after searching for an item.

2. Operations in a sequence may be unusually expensive.

3. There is always a sequence of operations that make them perform poorly. (such sequences are very rare to occur in practice)

Lists as Self-Adjusting Data Structures

Perhaps the simplest form of self-adjusting data structures is the list data structure. A restructuring method on lists suggested by Tarjan is the move-to-front rule. The move-to-front rule gives amortized efficiency by moving the accessed node to the front of the list on the basis that an accessed item will have a better chance to be accessed again in the near future. Thus frequently accessed items are kept closer to the front of the list dynamically. Move-to-front has many applications, one is in the implementation of the Least Recently Used (LRU) paging rule (11).

Another rule that can be a variant to move-to-front is the move-to-tail rule. Simply, move-to-tail rule moves a node to the end to the list after it is accessed.

Move-to-tail rule performs at its best when the access frequency of an item decreases after being accessed. If there are n items in a list L, $a_1,a_2,\ldots,a_n$ , then if $a_j$ is accessed its probability of being accessed again decreases until *most* other items $a_k$ , $1 \leq k \leq n$ , $k \neq j$ are accessed.

## The Splay Tree

The splay tree is another form of a self-adjusting data structure. Splay trees are binary search trees that provide amortized efficiency by using a restructuring technique called splaying; moving an accessed node to the root using defined restructuring techniques. Splay trees are competitive with balanced trees that guarantee a minimum average access time. Splay trees prove to be even superior when the usual case of having different frequencies of access applies. Balanced trees, while efficient, require extra storage space and extra effort to maintain their balance. A splay tree is simpler in both its operation and its structure. Furthermore, a splay tree has a practical advantage over the optimum tree in that it does not require a fixed frequency distribution of the items in the tree.

# Methods For Moving a Node to the Root Position

## Move-to-root Heuristics

Move-to-root was first described by Allen and Munro

(1). The idea is simple, move a node from any position in a tree to the root position. The move to the root is done in a systematic way called the simple exchange. Simple exchange is used on a node until it becomes the new root of the tree.

Simple exchange is done using left and right rotations. See Figure 1. The simple exchange procedure is described as follows:

case 1: Rotate left if node x is a right child of its parent.

case 2: Rotate right if node x is a left child of its parent.

The move-to-root heuristic provides an amortized time bound of $O(\log_2 n)$, given a sufficiently long sequence.

Splaying Operation

Splaying is a restructuring technique that is similar to move-to-root in that it does rotations until the accessed node is at the root position. The difference is that rotations are done in pairs which helps halving the accessed path.

Let p(x) and g(x) be the parent and grandparent of node x, respectively. The following is a description of splaying a node x:

Zig : If p(x) is the root, rotate the edge joining x and p(x). Terminate. See Figure 2.a.

Zig-zig: If p(x) is not the root and both x and p(x) are right or left children of g(x), rotate the edge joining p(x) with g(x) and then

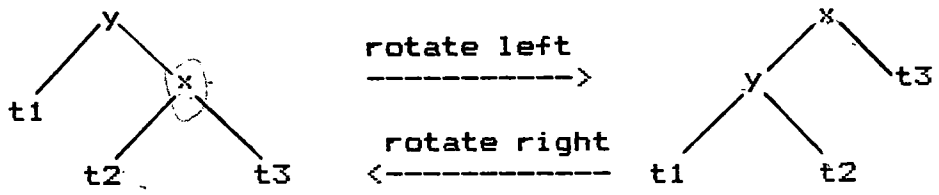



Figure 1. Single Rotations. Subtrees t1,t2,t3 may be null.


a.) Zig : right rotation.


b.) Zig-zig : double right rotation.


c.) Zig-zag : left right rotation.

Figure 2. Splaying rotation cases. For each case there is a symmetric case not shown above.

rotate the edge joining x with p(x). See Figure 2.b.

Zig-zag: If p(x) is not the root and x is a left child of p(x) and p(x) is a right child of g(x), (or vice versa), rotate the edge joining x with p(x) and then rotate the edge joining x with the new p(x). See Figure 2.c.

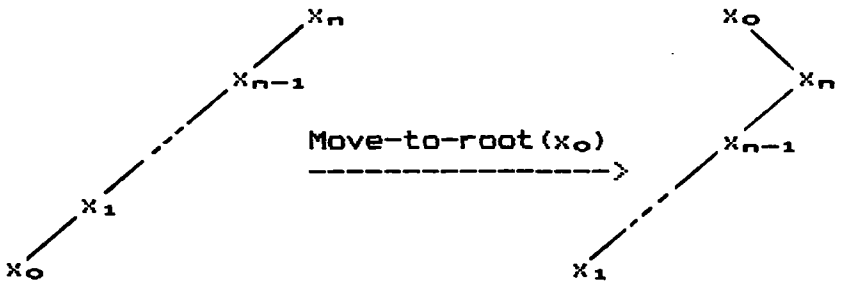
Move-to-root provides better running time but it suffers some potential problems that do not occur in splaying. A comparison between splaying and move-to-root in one special case is shown in Figure 3. The effect of halving the access path effectively is what makes splaying avoid the problems caused by different special cases.

Splaying can be used to implement many operations like insert, delete, join and split. One way to do these implementations is as follows:

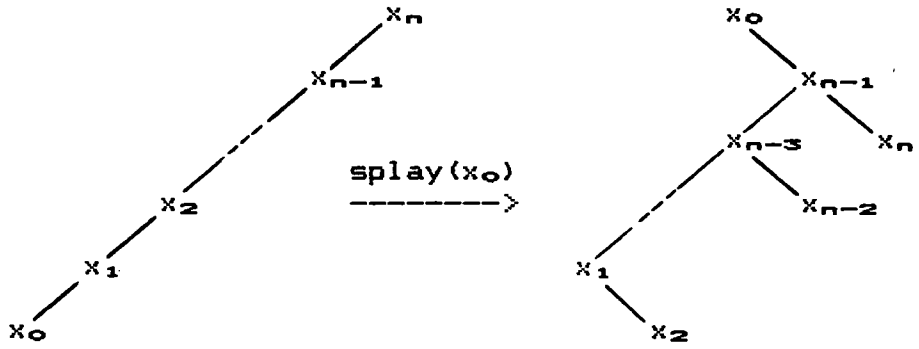
Access(i,t): Search for item i in the tree t. If a node x containing i is found splay at x and return a pointer to the new root x. If the item i is not found then splay at the last item accessed during the search and return a pointer to null.

Join(t1,t2): Access the largest item i in t1 and let t2 be the right subtree of the node containing i. See Figure 4.a.

Split(i,t): Access(i,t) and let the node x be the new root of t. If x contains an item less

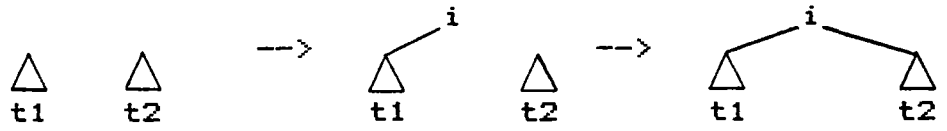



a.) Effect of move-to-root($x_0$).

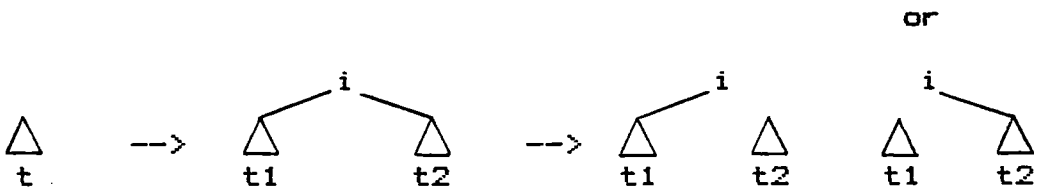



b.) Effect of splay($x_0$).

Figure 3. Illustration of the effect of the operations move-to-root and splaying in a special case.

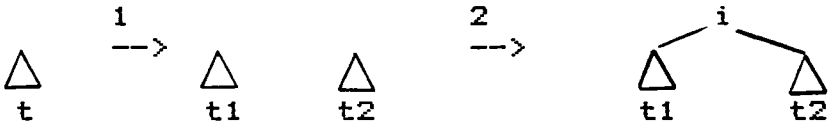



a.) Join(t1,t2).

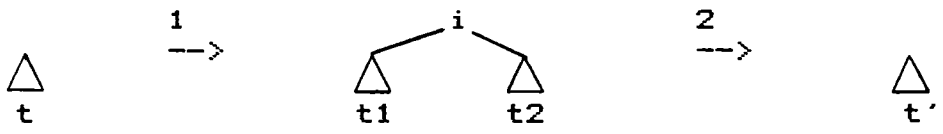



b.) Split(i,t), returns two trees one with i at root.

Figure 4. The effect of join and split. The split operation returns two trees.




a.) Insert(i,t): 1. Split 2. Let t1,t2 be the left and right subtrees of i.




b.) Delete(i,t): 1. Access(i,t) 2. Join(t1,t2).

Figure 5. Insertion and deletion using split and join, respectively.

than or equal to i then return (x,t1) and t2, otherwise return t1 and (x,t2). See Figure 4.b.

There are different ways to implement insert and delete. Two methods are described below:

Insert(i,t): Split(i,t) and let the trees t1 and t2 returned by split be the left and right subtrees of i, respectively. See Figure 5.a.

Delete(i,t): Access(i,t) and then join the left and right subtrees of the new root which contains the item i. See Figure 5.b.

Another way to implement insert and delete is as follows:

Insert(i,t): Search for i in t and let the null pointer encountered at the end of the search be pointing to the new node x containing i. Finally, complete the procedure by splaying at x.

Delete(i,t): Search for i in t, replace node x containing i by the tree resulting from joining the left and right subtrees of x. Complete the deletion by splaying at the parent of x.

Splaying can be done both bottom-up and top-down. Bottom up splaying requires two passes as it was described earlier. One pass to locate the position for splaying and the other is to do the actual splaying process. Bottom-up

splaying requires the ability to reach the parent of any node along the access path, to be able to do that either a parent pointer is to be added to the node structure or the access path may be stacked during the first pass. If space is of great importance, only two pointers may be enough to access the children and the parent of a node, one pointer points to the leftmost child and the other to the right sibling, if there was no right sibling then the other pointer points to the parent of the node. This representation may save space but it causes a loss in time efficiency.

The following is a procedure that does splaying bottom-up. Parent pointers are used explicitly. Pseudo code is used and detailed declarations are skipped.

```
Splay_bottom_up(x);
{
 while (p(x) != null) {
  if (g(x) == null) {
     if(x == right(p(x)) ) {
         rotateleft(p(x));
     else
         rotateright(p(x));
     }
  else { /* grandparent exists */
     if(p(x) == left(g(x)) ) {
         if(x == left(p(x)) ) {
             rotateright(g(x));
             rotateright(p(x));
         }
         else {
             rotateleft(p(x));
             rotateright(p(x));
         }
  }
  else {
      if(x == right(p(x)) ) {
          rotateleft(g(x));
          rotateleft(p(x));
```

```
        }
        else {
            rotateright(p(x));
            rotateleft(p(x));
        }
    }
  } /* while */
return(x);
}
```

Top-down splaying may also be used instead of bottom-up. Tarjan's method for top-down splaying is not as simple as bottom-up splaying but it has few advantages:

1. There is no need for a method to access the parent since all accesses occur from parent to child.
2. It is a one pass operation in the sense that the actual splaying action takes place while searching for the accessed item.

In other words, top-down splaying is more efficient storage and time wise but it is more complex than bottom up splaying. In Chapter III, a simple method for top-down splaying will be described.

Top-down splaying by Tarjan is done while searching the tree for the accessed item. At each accessed node along the search path the tree is split into three parts:

Left tree (L) : Contains all items that are already known to be less than the accessed item i.

Right tree (R): Contains all items that are already known to be greater than the accessed item i.

Middle tree : A subtree rooted by the current accessed node.

Top-down splaying is conducted by repeatedly applying

the cases described in Figure 6 until the accessed item or a null pointer is reached. Complete the top-down splaying operation by assembling the subtrees as shown in Figure 7. Figure 8 shows an example of top-down splaying. A similar result should be obtained if bottom-up splaying is used.

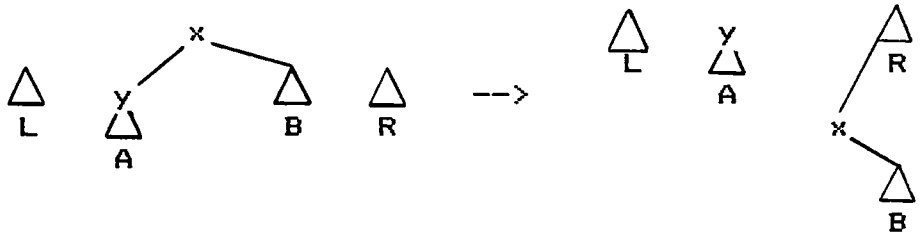
The following is the top-down splaying procedure as described by Tarjan. The variables t, *l*, and r are pointers to the current vertex of the middle tree, left tree, and right tree respectively. The procedures rotateleft and rotateright rotate the edge joining t to its left or right child respectively. The other procedures needed to do the top-down splaying are described as follows:

Linkleft : Break the link joining t to its left child and attach the resulting tree to the right of the left tree.

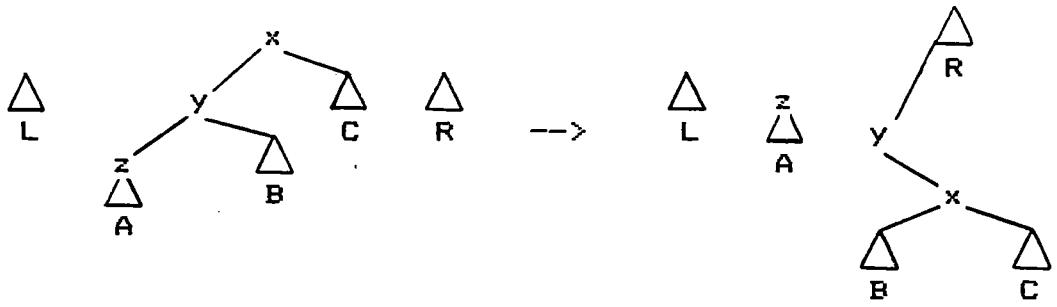
Linkright: Break the link joining t to its right child and attach the resulting tree to the left of the right tree.

Assemble : Complete the top-down splaying by assembling the left middle and right tree into the final tree. See Figure 7.

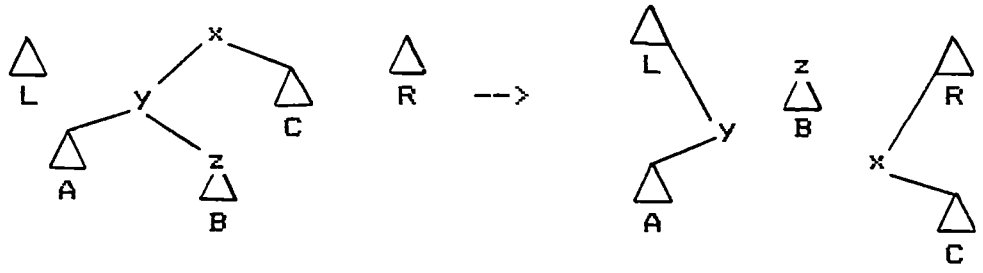
```
Splay_top_down(i,t);
{
   /* Initialize left(null) and right(null) */
  if (l=null && r==null ) {
      left(null)=null;  right(null)=null;
  }

  while(i != item(t)) /* item(t) -> key in t */
      if (i < item(t) ) {
          if i=item(left(t))
```




a.) Zig: Single rotation. Item i is in A.




b.) Zig-zig: Two similar single rotations. Item i is in A.




c.) Zig-zag: Two different single rotations. Item i is in B.

Figure 6. Top-down splaying cases. Item i is splayed. Symmetric cases exist but not shown.

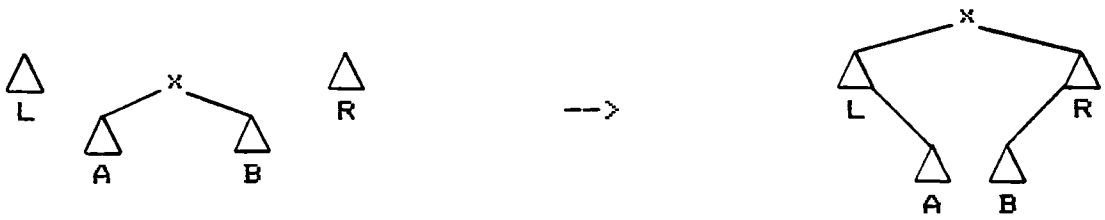
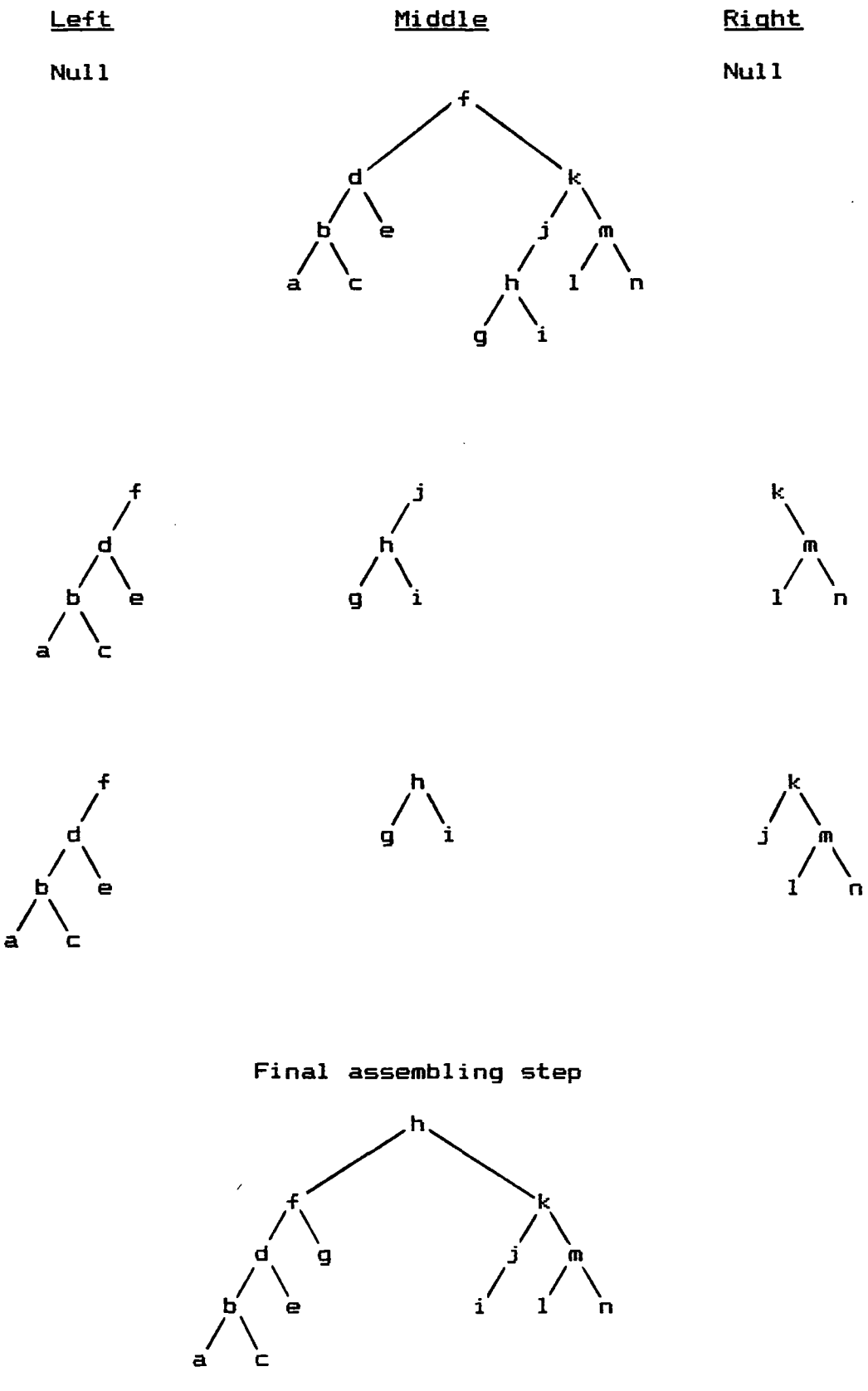



Figure 7. Final assembling step of top-down splaying completed by putting the various subtrees together as shown.




Final assembling step




Figure 8. Top-down splaying example. Splay at h.

```
                linkright;
            else if (i< item(left(t)) {
                rotateright;
                linkright;
            }
            else {
                linkright;
                linkleft;
            }
        }
        else {
            if (i>item(t)) {
                if(i=item(right(t))
                    linkleft;
                else if (i>item(right(t)) {
                    rotateleft;
                    linkleft;
                }
            else {
                linkleft;
                linkright;
            }
        }
    }
}
```

The following is the implementation of linkleft, rotateleft, and assemble: (Linkright and rotateright are symmetric)

```
Procedure linkleft;
{
    t=right(t);
    l=t;
    right(l)=t;
}


Procedure rotateleft;
{
    t=right(t);
    right(t)=left(right(t));
    left(right(t))=t;
}


Procedure assemble;
{
    left(r)=right(t);  right(l)=left(t);
    left(t)=right(null); right(t)=left(null);
}
```

Splaying in general requires extensive restructuring. The semi-adjusting search tree is a variation to the splay tree. It attempts to decrease the number of times a tree needs to be adjusted. There are many ways to achieve fewer adjustments in the splaying operation, some are listed below:

1. Move a node only part way towards the root instead of all the way to the root.
2. Splay only if the access path to the item to splay at is relatively long.
3. Splay only if the item is in the tree in case of access and insert, do not splay in case of delete.

These suggestions require a lot of studying to determine how they would exactly effect the efficiency of splaying.

## The Link/Cut Tree

The problem of linking and cutting trees is a problem of maintaining a collection of vertex-disjoint trees under a sequence of primarily two kinds of operations namely, link and cut. These two operations effect these vertex-disjoint rooted trees by adding and deleting edges over time. Link is an operation that combines two trees into one tree by adding a new edge and cut is an operation that deletes an edge to divide a tree into two. Link/cut trees have many important applications:

1. Network flow problems like finding minimum, maximum, blocking and acyclic flow.
2. Finding the nearest common ancestors.
3. Implementing the network simplex algorithm for

minimum-cost flow.

4. Computing some kinds of constrained minimum spanning trees.

Tarjan used the link/cut tree to find a maximum flow in $O(m*n*\log_2 n)$ of a network with n vertices and m edges, beating by a factor of $(\log_2 n)$ the fastest algorithm ever known for sparse graphs (12).

The operations on link/cut trees are defined as follows:

Maketree(v) : Create a new tree with the vertex v. Let the cost of v be zero.

Findroot(v) : Return the root of the tree containing the vertex v.

Findcost(v) : Return the pair [w,x] were x is the minimum cost of a vertex w on the path from v to findroot(v). The vertex w is chosen so that it is closest to the root.

Addcost(v,x): Add the real value x to the cost of every vertex on the path from v to findroot(v).

Link(v,w) : Link the trees containing vertices v and w by adding the edge [v,w]. The vertices v and w belong to two separate trees were v is the root of one tree.

Cut(v) : Divide the tree containing vertex v into two trees by deleting the edge coming out from v. The vertex v must not be a

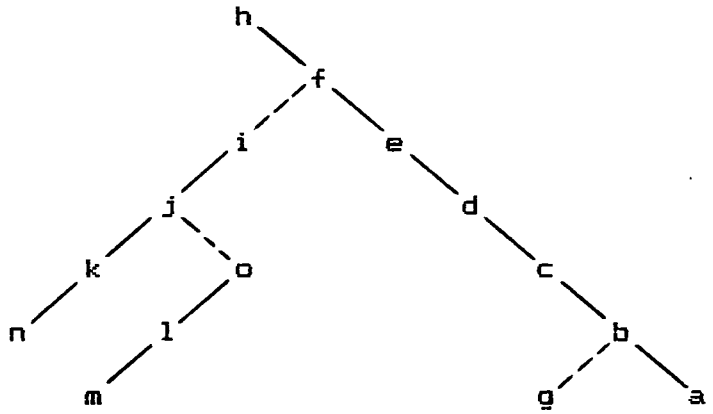
tree root.

One way to solve this problem is by storing parent pointers in each vertex. This method is the simplest and it is used to perform maketree, link, and cut operations in O(1) time each, and findroot, findcost, and addcost each in an order proportional to the depth of the input vertex, which is O(n) in the worst case.

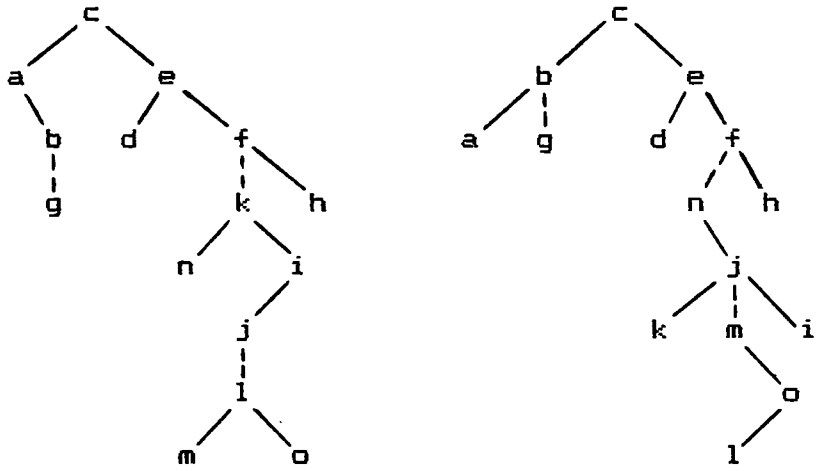
The other method suggested by Tarjan performs each operation in a time bound of $O(\log_2 n)$. The method represents each link/cut tree by a set of paths which constitute what is called a virtual tree. In a virtual tree, each path is represented by a binary tree called a solid subtree, all solid subtrees are interconnected by dashed edges. An inorder traversal of each solid subtree produces its corresponding path in the actual tree. Edges belonging to a solid subtree are all called solid edges. In other words, the virtual tree consists of solid subtrees each representing a path in the actual tree. More than one virtual tree can be constructed for a given actual tree. See Figure 9. The information stored in each node consists of the following:

1. A left child pointer.
2. A right child pointer.
3. A parent pointer.

Other information is stored in the nodes like cost(x) and mincost(x). Cost(x) is a real value assigned to the node of vertex x and mincost(x) is the minimum cost of a descendant of x. Storing the cost and mincost explicitly




a.) Actual tree. Path [*n*,*k*,*j*,*i*] has head *n* and tail *i*.




b.) Two possible virtual trees of the tree in (a).

Figure 9. Both actual and virtual tree representations. Items not in search tree order.

makes addcost operation expensive so if x is the root of a solid tree in a virtual tree then store cost(x), otherwise store cost(x) - cost(parent(x)) for the change in cost field (dcost). The other field that is stored is the change in min. (dmin) which is equal to the cost(x)-mincost(x).

To make the link/cut tree operations efficient, splaying is used to move a node to the root of its virtual tree. Another operation called splicing is needed. Splicing makes any middle child v of a root of a solid subtree a left child and the old left child, if any, becomes a middle child. Figure 10 illustrates how splicing works. Both splaying and splicing will effect the values of cost and mincost and thus the stored values should be adjusted appropriately. The overall splaying operation at a node x is a three pass operation defined as follows:

Pass 1: Move up the tree from x to the root while splaying within every solid tree. This pass should make the path from x to the root consist only of dashed edges. See Figure 11.a.

Pass 2: Move from x to the root while splicing at each node along the dashed path to the root. After this pass x and the root become members of the same solid tree. Node x can be reached from the root by following left pointers. See Figure 11.b.

Pass 3: Splay at x thus completing the overall process by moving the node x all the way to

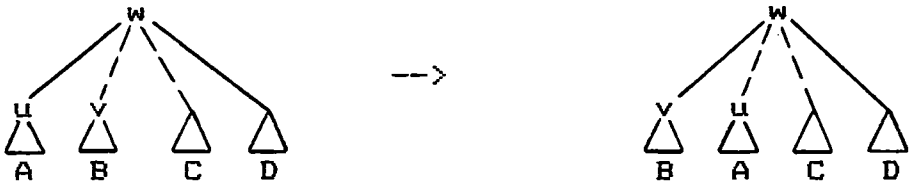



Figure 10. Splicing operation. Node *v* becomes a left child of *w*.

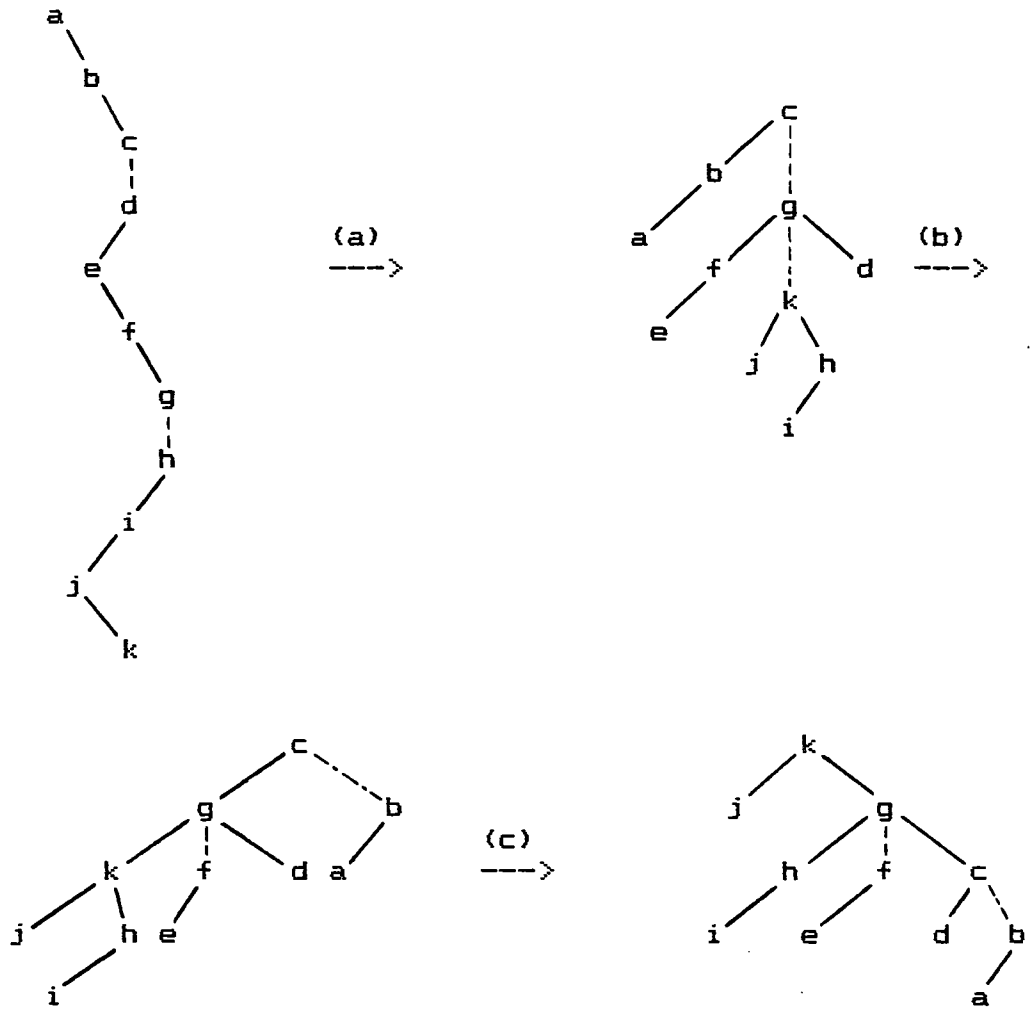



Figure 11. Three pass splaying example. Splay3(*k*) at node *k*. (a) Pass 1: Splaying inside each solid tree. (b) Pass 2: Splicing along the new dashed path from *k* to the root. (c) Pass 3: zig-zig splaying step at *k* in the final solid tree.

the root. Notice that only the zig-zig splaying step is needed for this pass. The final virtual tree will have x at the root position. See Figure 11.c.

All the link/cut operations can be implemented using three pass splaying. Each operation has an $O(\log_2 n)$ amortized time bound and a long sequence of m operations takes an $O(m*\log_2 n)$ time. One important link/cut tree operation that is performed in $O(\log_2 n)$ time is the evert operation. The evert operation makes a node v the root of its tree by reversing the path from v to the original root. An extra bit of storage is needed to know if the meaning of the left and right pointer is reversed. A modification to the data structure would store the costs in the edges rather than the vertices.

# CHAPTER III

## SPLAYING TECHNIQUES

### Introduction

In this chapter different methods for moving a node to the root position are presented. While these new methods may seem totally different, in fact they are basically similar in effect to those methods already discussed in Chapter II, namely move-to-root and splaying.

Both move-to-root and splaying use single rotations to move a node to the root position in a tree. Move-to-root involves one node at a time to do the rotations but splaying differs by taking nodes in pairs. Splaying has a top-down version which was presented in Chapter II, but move-to-root does not have any top-down version since it is not possible to conduct single rotations at the root that involve one node at a time and still be able to pull a node all the way to the root position in a tree.

Two new methods are presented in details in this chapter. The first works with one node at a time and the other takes nodes in pairs. Both methods work top-down and can not be done bottom-up with out loosing some of the time efficiency of the splay tree. Analysis is made on the original splaying technique and on the two new methods to

determine their efficiency under different situations.

## Two New Methods to Move a Node to the Root Position

### Single-Node Adjustment Method

Moving a node x from any position in a tree to the root involves restructuring the access path, the path from the root to node x. All nodes in the tree that do not lie along the access path are not involved in the restructuring and can be thought of as being subtrees of the nodes that are along the path.

In any top-down restructuring method; a left tree and a right tree are required. The left and the right trees contain those nodes that are already known to be less than and greater than the node x, respectively.

The procedure for moving a node x containing an item i to the root position using the single-node adjustment method is described as follows:

- If the current node is x, terminate and apply the assembling step. See Figure 7.
- If the item in the current accessed node is greater than i then let the current node and its right subtree be a left child of the leftmost node in the right tree.
- If the item in the current accessed node is less than i then let the current node and its left subtree be a right child of the rightmost node in

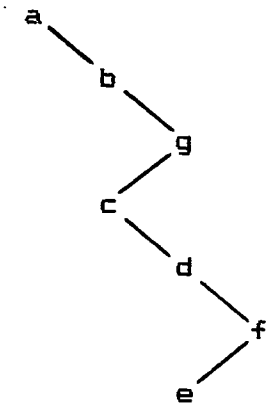
the left tree.

An example of this method is shown in Figure 12.b. The final result is a tree with node x at the root and all the other nodes along the access path lie either along the leftmost path of the right subtree of x or along the rightmost path of the left subtree of x. If the number of nodes along the access path with items less than i is equal to those greater than i then the best result is obtained where the access path is halved. In a random environment this will be the approximate case.

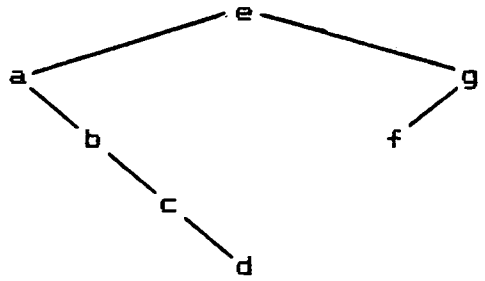
Move-to-root can be done top-down by using this method. Each simple exchange in move-to-root requires three pointer changes and if parent pointers are used, since there must be a way to be able to access parents, then three additional pointer changes are needed. By using this top-down version of move-to-root the total number of pointer changes can be reduced to only one pointer change per node along the access path. To complete the restructuring, four additional pointer changes are needed for the final assembling step. Move-to-root is a bottom-up process and thus two passes are needed, one to find the search item and the other to do the restructuring. In this method, only one pass is necessary since the search is done during the restructuring process. Another advantage to this method over move-to-root is its simplicity, it is easier to understand and apply.

Double-Nodes Adjustments Method

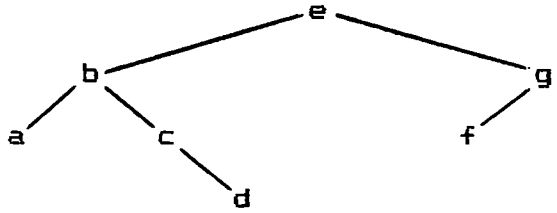
The potential problems with move-to-root still exist in




a.) Original tree.




b.) Single-node method.




c.) Double-node method.

Figure 12. Moving the node e to the root positon. Items in search tree order.

the single-node adjustment method. A simple modification to the single-node method that helps avoiding these problems is to consider nodes in pairs instead of one at a time. This can be achieved by having a lookahead pointer during the restructuring process. The modified method is similar in effect to the top-down splaying method described in Chapter II.

In this procedure, the left and right trees are also used as in the first method. Let the left and right trees be initially null and also initially let *a* and *b* be the top two nodes along the access path. Also let *I* be the last node on the rightmost path of the left tree and r be the last node on the leftmost path of the right tree. An add operation is used in this method. The operation add(z), an add of a node z, is defined as follows:

- If the item j in node z is less than i (the item in node x), then add the node z to the left tree; if j is greater than the item in *I* then let z be a right child of *I* and let *I* represent the node z, otherwise if the item j in node z is less than the item in *I* then let the left subtree of *I* be the right subtree of node z and let node z be a left child of node *I*.
- If the item j in node z is greater than i, then add node z to the right tree; if j is less than the item in r then let node z be a left child of r and let r represent the node z, otherwise, if the item j in node z is greater than the item in r then let the

right subtree of r be the left subtree of node z and let node z be a right child of r. Notice that this is a symmetric case to the above.

To be able to follow the add operation correctly it is helpful to apply the complete algorithm to the tree in Figure 12.a. Excluding few checkings, the following is a description of the complete procedure of moving a node x containing an item i to the root position. Initially, let a and *b* be the first two nodes along the access path starting at the root:

1. If a is the node x, terminate and apply the final assembling step. Figure 7 illustrates the final assembling step.
2. If *b* is the node x, add(a) and terminate. Apply final assembling step.
3. Otherwise, perform Add(*b*) then add(a) and let a and *b* be the next two nodes along the access path. Go to step 1.

An example of this method is shown in Figure 12.c. The order of the vertices of the tree in Figure 12.a that are involved in the add operations are *b,a,c,g,f,d*. This method enjoys all the advantages of top-down splaying by Tarjan plus a few more. Each pair of nodes takes either two or three pointer changes which is an improvement to the four pointer changes in the top-down splaying. Moreover, this method is simpler than the top-down splaying since all the different restructuring cases in top-down splaying are implied in this method and that makes it more straight forward and easier to program.

Other than the above two methods, there is a way to

restructure a path and that is by doing insertions at the root. This is a very simple method as far as programming is concerned. This method is very naive because it does not take advantage of the pattern that exists during the restructuring process just like the first two methods described above.

## Empirical Analysis

In general, when using splaying, the case must be such that the access frequencies of few items in the tree are much higher than other items in the same tree. To be able to conduct an empirical analysis on the splay tree a way to generate items with different frequencies of access must be available. This method is presented in here to give a clearer idea of how the empirical analysis results were obtained.

Let p percent of n numbers have a high:low (h:*l*) frequency with respect to other items in the tree. The procedure used is:

1. Generate n numbers using a random number generator in the range of 1-r.

2. Use the same random number generator to generate p*n numbers in the range of 1-n.

3. Let the numbers generated in step 2 have a high frequency of h and all other numbers to have a low frequency of *l*.

4. Accumulate the frequencies;
```
     for all i = 1..n-1
         freq(i+1)=freq(i+1)+freq(i);
```
   and then normalize the frequencies
```
     freq(i)=freq(i)/freq(n).
```

5. Using the same random number generator, generate

numbers greater than zero and less than one. Use each generated number to search in the normalized frequencies for the the value that is just greater than or equal to this number and then the sequence number of the found value will be the finally generated number.

The probability for a single item to be accessed at any time is:

$$P(i) = f(i) / (\text{total frequencies})$$

were f is the initially assigned frequency to the item which is either h or *I* in this case. The probability to access an item with a high frequency is:

$$Ph = h / (p*n*h + (1-p)*n*I)$$

and the probability to access an item with a low frequency *I* is:

$$Pl = I / (p*n*h + (1-p)*n*I)$$

The total probability to access any of the items with a high frequency is:

$$TPh = Ph * (p*n)$$

and the total probability to access any of the items with a low frequency is:

$$TPl = Pl * (1-p)*n$$

Then,

$$TPh + TPl = 1.0$$

The above number generator used in the analysis can be modified to generate characters instead of numbers. Also a modification can be made to let the high:low frequencies of access be chosen at random. These modifications are not essential for the analysis. The control variables used above are listed below for convenience:

n : Number of different values to be generated.

na : Number of accesses to a tree. Items accessed are already in the tree.

r : The range of the n values generated 1-r.

p% : The percent of values that have high frequency of access.

h:*l*: Initial high:low frequency of access.

Using the above control variables, a table is shown in Table I comparing bottom up splaying, top-down splaying, move-to-root, and the two methods presented in this chapter. The average path lengths shown in the table for the single-node and move-to-root methods are equivalent as is the case also with the double-nodes method and top-down splaying.

In Table II , the simple binary tree which involves no restructuring is shown with the single-node , double-nodes and bottom-up splaying. Also the value of $\log_2 n$ is shown in the table to give an idea of how these methods compare with the balanced tree which has an order $O(\log_2 n)$.

Table III shows the effect of having different percentages of the total items with high frequency of access. To minimize the average path length, the total probabilities of the high frequency items should be a maximum and at the same time the value for p should be such as only a fraction of the n items have high frequencies of access. At p=0.01 in Table III, TPh=0.5 and at p=0.1, TPh=0.91. At the next value p=0.2, Ph=.97 but the average path length starts increasing because of the increase in the

TABLE I

AVERAGE PATH LENGTH RESULTS FOR
DIFFERENT NUMBER OF ITEMS
IN THE SPLAY TREE

| n | Single -node | Move-to -root | Double -nodes | Top-down splaying | Bottom-up splaying | $Log_2 n$ |
|---|---|---|---|---|---|---|
| 1000 | 10.313 | 10.313 | 10.676 | 10.676 | 10.674 | 9.466 |
| 2000 | 11.713 | 11.713 | 12.135 | 12.135 | 12.187 | 10.966 |
| 4000 | 12.912 | 12.912 | 13.449 | 13.449 | 13.470 | 11.966 |
| 8000 | 14.333 | 14.333 | 14.422 | 14.422 | 14.443 | 12.966 |
| 16000 | 15.697 | 15.697 | 16.316 | 16.316 | 16.373 | 13.966 |
| 20000 | 16.091 | 16.091 | 16.729 | 16.729 | 16.760 | 14.288 |

The control variables; h:$I$=20:1, p=0.2, na=n.

TABLE II

EFFECT OF A MODERATE HIGH:LOW FREQUENCY
RATIO ON THE AVERAGE
PATH LENGTH

| N | Single -node | Double -nodes | Bottom-up splaying | Binary | $Log_2 n$ |
|---|---|---|---|---|---|
| 1000 | 9.548 | 9.934 | 9.969 | 11.965 | 9.966 |
| 2000 | 11.019 | 11.364 | 11.476 | 13.346 | 10.966 |
| 4000 | 12.236 | 12.711 | 12.698 | 14.348 | 11.966 |
| 8000 | 13.629 | 14.153 | 14.188 | 15.959 | 12.966 |
| 16000 | 14.998 | 15.590 | 15.623 | 17.280 | 13.966 |
| 20000 | 15.427 | 16.071 | 16.071 | 17.797 | 14.288 |

The control variables; h:$I$=100:1, p=0.2, na=n.

TABLE III

THE EFFECT OF THE PERCENT OF ITEMS WITH A MODERATE HIGH:LOW FREQUENCY ON THE AVERAGE PATH LENGTH

| p | Single -node | Double -nodes | Bottom-up splaying | Binary |
|---|---|---|---|---|
| 0.0005 | 14.935 | 15.527 | 14.802 | 14.934 |
| 0.002 | 13.839 | 13.361 | 14.358 | 14.941 |
| 0.01 | 11.650 | 12.085 | 12.128 | 14.861 |
| 0.1 | 11.583 | 11.958 | 12.019 | 14.942 |
| 0.2 | 12.692 | 13.159 | 13.180 | 14.832 |
| 0.3 | 13.440 | 13.935 | 13.995 | 14.960 |
| 0.5 | 14.32 | 14.885 | 14.889 | 15.064 |
| 0.7 | 14.762 | 15.341 | 15.345 | 14.966 |
| 0.9 | 15.049 | 15.613 | 15.690 | 14.946 |
| 1.0 | 15.247 | 15.809 | 15.840 | 14.885 |

The control variables; h:$I$ = 100:1, n=5000, na=n, $\log_2 n$=12.288.

TABLE IV

THE EFFECT OF THE PERCENT OF ITEMS WITH A HIGH HIGH:LOW FREQUENCY RATIO ON THE AVERAGE PATH LENGTH

| p | Single -node | Double -nodes | Bottom-up splaying | Binary |
|---|---|---|---|---|
| 0.0005 | 1.545 | 2.042 | 1.921 | 12.910 |
| 0.002 | 3.512 | 3.535 | 3.553 | 13.269 |
| 0.01 | 6.254 | 6.417 | 6.441 | 14.828 |
| 0.05 | 9.434 | 9.742 | 9.779 | 14.828 |
| 0.10 | 10.894 | 11.280 | 11.300 | 15.108 |
| 0.15 | 11.814 | 12.225 | 12.256 | 15.180 |
| 0.20 | 12.460 | 12.900 | 12.975 | 15.123 |
| 0.50 | 14.273 | 14.781 | 14.820 | 15.033 |
| 0.90 | 15.009 | 15.590 | 15.590 | 14.964 |

The control variables; n=5000, na=n, h:$I$=1000000:1, $\log_2 n$=12.288.

number of items with high frequency of access and which is computed by the equation p*n.

If the high to low frequency ratio is very high then the average path length becomes an order of $O(\log_2(p*n))$ which means that all the accesses are only to a subset of the total items, which is the subset with the high frequency. This result is tabulated in Table IV.

In Table V. the effect of changing the high:low frequency distribution is shown. Also shown in the table is the percent of the total probabilities for accessing a high frequency item (TPh %). After 50:1 high:low frequency is reached, the value for TPh goes already above 0.9 and any more increase in h:*l* does not make any significant improvement in the average path length.

Finally, it is important to note that there is a steady state that is reached after a number of accesses. The steady state is defined to be a state at which the average access time of consecutive subsets of the accesses remains fairly constant. Being in a steady state also means that the access frequencies of the items are not currently changing. The average path length starts at its peak for the initial accesses then if the access frequencies of the items do not change, the average access length starts decreasing with accesses till a steady state is reached.

In a splay tree the access frequencies may change over time. If this happens then the splay tree adjusts itself to reach another steady state for the new frequencies. Table VI shows the average path length after an increasing

TABLE V

THE EFFECT OF HIGH:LOW FREQUENCY RATIO
ON THE AVERAGE PATH LENGTH

| h:l | single -node | Double -node | Bottom:up splaying | Binary | TPh (%) |
|---|---|---|---|---|---|
| 1:1 | 15.139 | 15.696 | 15.722 | 14.93 | 20.00 |
| 10:1 | 13.947 | 14.469 | 14.487 | 14.913 | 71.43 |
| 50:1 | 12.864 | 13.277 | 13.353 | 14.854 | 92.26 |
| 100:1 | 12.692 | 13.159 | 13.180 | 14.832 | 96.15 |
| 500:1 | 12.423 | 12.888 | 12.915 | 14.816 | 99.21 |
| 1000:1 | 12.386 | 12.843 | 12.854 | 14.811 | 99.60 |
| 5000:1 | 12.378 | 12.820 | 12.843 | 14.781 | 99.92 |
| 10000:1 | 12.379 | 12.821 | 12.881 | 14.783 | 99.96 |
| 100000:1 | 12.374 | 12.812 | 12.867 | 14.787 | 100.00 |

The control variables are; n=5000, na=n, p=0.2, log2n=12.288

TABLE VI

STEADY STATE REACHED AFTER A
RELATIVELY HIGH NUMBER
OF ACCESSES

| na | Single -node | Double -node | Binary |
|---|---|---|---|
| 100 | 10.350 | 10.650 | 11.810 |
| 400 | 9.150 | 9.388 | 11.560 |
| 800 | 8.638 | 8.900 | 11.503 |
| 1000 | 8.536 | 8.798 | 11.533 |
| 2000 | 8.429 | 8.693 | 11.577 |
| 4000 | 8.307 | 8.596 | 11.577 |
| 8000 | 8.197 | 8.469 | 11.580 |
| 16000 | 8.126 | 8.394 | 11.561 |

The control variables are, n=1000, h:l=100:1, p=0.1, $\log_{2n}$=9.966.

number of accesses where a steady state is obtained. Notice also that a steady state is reached after at least 1/Ph accesses. In Table VI, after 100 access, TPh (%) of the high frequency items are accessed. To access all high frequency items at least

$$S = p*n * 100 /(TPh\ \%),$$
$$= p*n/( Ph * p * n),$$
$$= 1/Ph$$

accesses should be made. At a steady state all the items with high frequency of access should have been already accessed at least once and thus a steady state is reached after $S \geq 1/Ph$ accesses assuming that items have same high frequencies (all high frequency items have equal probability to be accessed).

At the end of this chapter, the following can be concluded from the above results:

- Splaying is only time efficient if the access frequencies of *few* items in the tree is *high* compared to other items in the same tree.
- Move-to-root has a better time efficiency than splaying in a random environment. The reason for this is that move-to-root does not disturb the structure of the tree by moving the nodes near the root too far down the tree as much as splaying and its variants do.
- The single-node adjustment method presented in this chapter is indeed as efficient as move-to-root and the double-nodes adjustments method is as efficient

as top-down splaying.

- The appropriate case where a splay tree can be used is when :

$$1/n \leq p < 0.2$$

$$50 < h/l < \text{infinity}$$

$$0.5 < TPh < 1.0$$

It is important to notice that TPh is dependent on p and h/*l*. A proper combination of p and h/*l* must be in the above range of TPh to get best results for the average path length. There is no one optimum value for p or h/*l* but there are several combinations that will provide the optimum result, a minimum average path length.

- The efficiency of the splay tree does not depend on the value of n, the number of items in the splay tree. The independency from the number of items in the tree is important to establish in a general data structure as the splay tree. See Table I.
- A steady state is reached after a sufficiently long number of accesses. At a steady state, the average path length is at its minimum. If the frequency of access of all or some of the items change after a steady state is reached then the splay tree will adjust itself and a new steady state is reached after another sufficiently long sequence of accesses.

The splay tree has many applications and uses. One important use that is presented in Chapter IV is in the

implementation of the link/cut tree operations. Another use of the splay tree is in the lexicographic or multidimensional search tree (8,10). The idea of self-adjusting data structures is not limited to the splay tree. It may be possible for example to have a self-adjusting B-tree where the access frequency of the items is given some consideration by keeping in a systematic way those items with the high frequency of access closer to the root. As is the case with most data structures, a modification to the standard B-tree may be needed to be able to have a self-adjusting B-tree (2,4,13).

# CHAPTER IV

# THE LINK/CUT TREE

## Introduction

In Chapter II, the problem of linking and cutting trees was introduced. In this chapter, different solutions for this problem are presented. The link/cut tree problem can be solved if the operations maketree, findroot, addcost, link, and cut are implemented. These operations have already been defined in Chapter II. A comparison is conducted between the different methods used to implement the link/cut tree operations to show the advantages and disadvantages of each method.

## Solving the Linking and Cutting Trees Problem

### Assigning Parent Pointers

A data structure that will help solve the problem of linking and cutting trees is a tree with parent pointers added to each node. No left or right pointers are needed since all operations are bottom up; they start at the node and work up to the root level. Using this data structure, the link/cut tree operations can be implemented as follows:

Maketree(v) Create a new tree with the node or vertex v of cost zero.

Findroot(v) Follow parent pointers starting at v until a vertex w with a null parent is reached. Return w as the root of the tree that contains the vertex v.

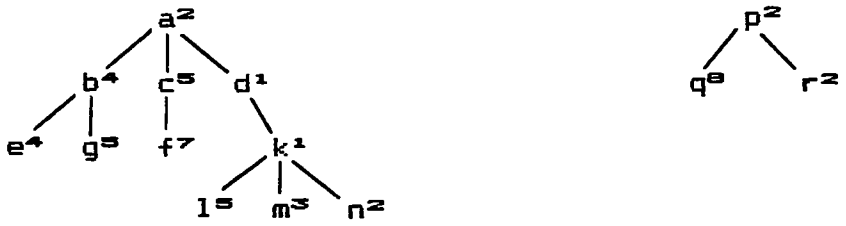
Findcost(v) Start at the vertex v and walk up the path from v to findroot(v). Let the minimum cost along the path be x. Let w be the vertex with cost x. If more than one vertex has the same minimum cost x, then let w be the vertex that is closest to findroot(v). Return the pair [w,x].

Addcost(v,x) Follow parent pointers from v to findroot(v) adding the real value x to every vertex cost along the path.

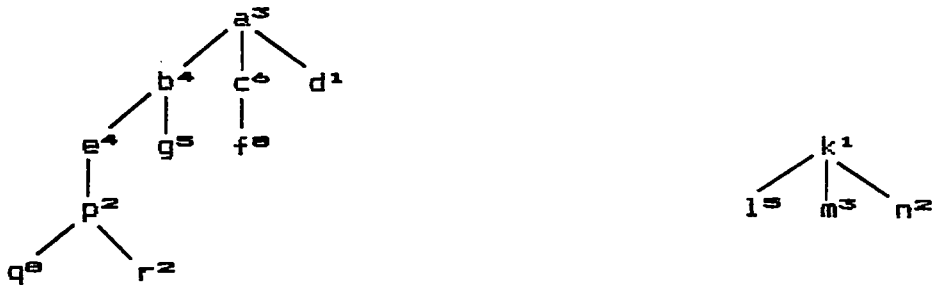
Link(v,w) Let v be a child of w by adding the edge [v,w]. Note that edges are directed from child to parent. This operation requires that v and w be vertices of different trees.

Cut(v) Delete the edge coming out from v to its parent and return the two trees rooted at v and at the old findroot(v).

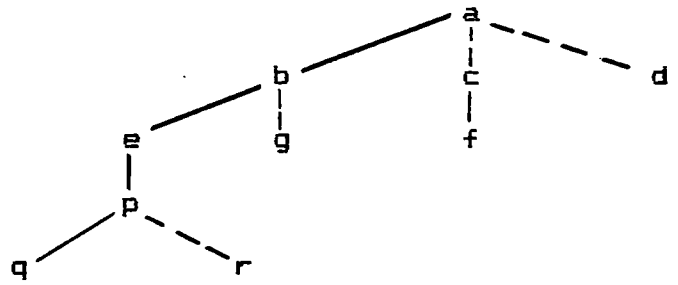
The effect of these operations is shown in Figure 13.a,b. This is the simplest method among those presented in this chapter. The implementation is straight forward and only a parent pointer is needed in each vertex which makes




a.) Two trees. The operations findroot($g$) returns a and findcost($n$) returns [$d$,1].




b.) Result after executing the three operations link($p$,$e$), addcost($f$,1), and cut($k$).




c.) Link/cut tree divided into solid paths separated with dashed edges. The path [$q$,$p$,$e$,$b$,a] has head $q$ and tail a.

Figure 13. Link/cut tree examples. Superscripts are vertex costs.

it storage efficient since there is no need for left and right pointers as is the case with the other two methods. The operations maketree, link, and cut take a constant time of O(1) but each findroot, findcost, and addcost operation takes time proportional to the depth of the input vertex which is O(n). Thus the amortized time of this method for m worst case operations is O(m*n). This time bound is not acceptable for most practical applications and thus a different data structure is needed with a better amortized time efficiency.

Division into Sets of Paths

Another approach to the solution of the link/cut tree problem is by representing the link/cut trees as sets of paths. Each tree is partitioned into a set of vertex disjoint paths. Edges are either solid or dashed. Solid edges connect the vertices of a path and dashed edges connect the paths in a tree. See Figure 13.c. A solid edge is represented by a parent pointer and a dashed edge is represented by a successor pointer. Only one pointer may be used if an extra bit is added to tell whether the pointer is to a parent or to a successor. In this presentation, both parent and successor pointers are used explicitly for simplicity and clarity.

Beside the left, right, and parent pointers, other information is stored in the nodes, namely, cost(x) and mincost(x). The cost(x) is an assigned real value for every vertex and the mincost(x) is the minimum cost of the

descendants of x, including x, in the solid tree containing x. Instead of storing the cost and mincost explicitly, which makes the addcost operation expensive, these values are stored as follows:

$$dcost(x) = cost(x) - mincost(x)$$

$$dmin(x) = mincost(x) \quad \text{if } x \text{ is the root of its solid tree.}$$

or,

$$= mincost(x) - mincost(parent(x)) \quad \text{if } x \text{ is not a solid tree root.}$$

The value $dcost(x) \geq 0$ for any vertex x and $dmin(x) \geq 0$ for any nonroot vertex x. To find mincost(x), move along the solid tree path from the root to x summing dmin along the path. The function cost(x) is found by adding mincost(x) to dcost(x).

Before starting the implementation of the link/cut tree operations, a data structure has to be chosen for the set of paths representing the link/cut tree. There is not a one and only one representation that should be used but the choice of the data structure to be used to represent the paths has a direct effect on the performance of the link/cut tree. One representation may be a list data structure. Using lists to represent paths makes it easy to understand the link/cut tree operations since the virtual tree (a virtual tree is a representation for the actual tree with connected paths that are represented with some kind of a data structure - see Chapter II) will not be different from the actual tree and also lists make the implementation

itself easier. Lists are simple but they have an O(n) time bound in the worst case. Another representation for the paths can be the binary tree. For example, if balanced trees are used, as a type of a binary tree, then the time per path operation is $O(\log_2 n)$, and thus a sequence of m tree operations takes $O(m(\log_2 n)^2)$ time. This result was found by Galil and Naamad (7) and Shiloach (10). This order can be improved to $O(m*\log_2 n)$ time bound if the splay tree is used.

In the following implementation the link/cut tree is divided into a set of paths. A set of path operations is used to be able to complete the implementation. These path operations are defined as follows:

| | |
|---|---|
| Makepath(v) | Create a new path containing the vertex v. |
| Findpath(v) | Return the path containing the vertex v. Each path has an identifier. |
| Findtail(p) | Return the last vertex on the path p. |
| Findpathcost(p) | Return the pair [w,x], where x is the minimum cost of a vertex on the path p. The vertex w is the vertex with cost x that is closer to findtail(p). |
| Addpathcost(p,x) | Add the real value x to the cost of every vertex on the path p. |
| Join(p,v,q) | Add an edge from the tail of p to v |

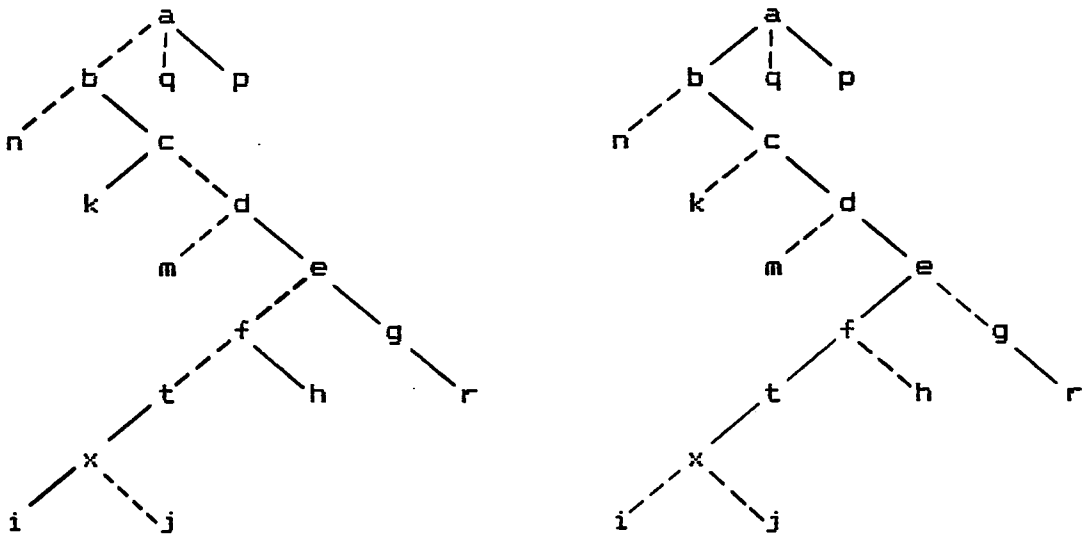
and another edge from v to the head of q. Return the resulting new path.

Split(v) Return the pair [p,q] where p is the part of the path containing v from the head to the vertex just before v, and q is the part from the vertex just after v to the tail of the path containing v. The paths p and q may be empty if v is the head or the tail in its path, respectively.

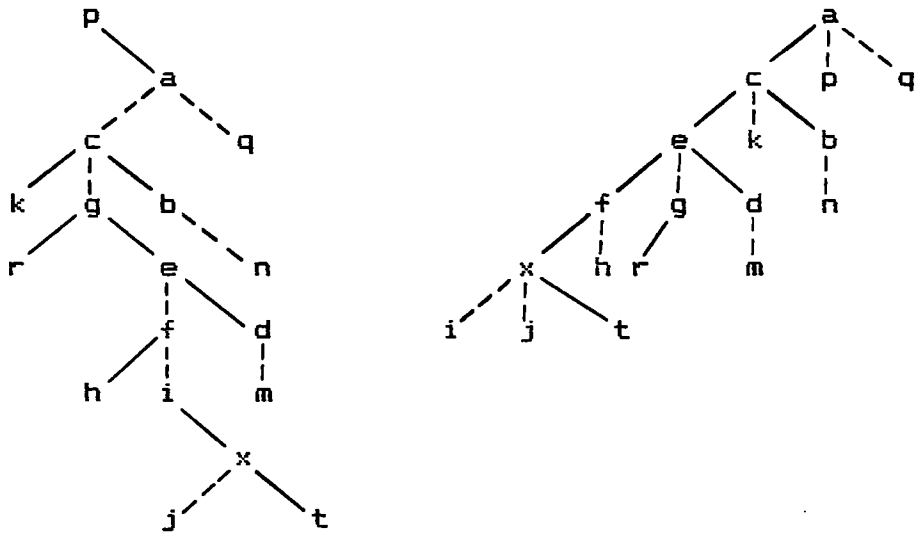
Another operation that is needed to carry out the link/cut tree operations is the expose operation which is defined as follows:

Expose(v) Make the tree path from v to findroot(v) solid by converting dashed edges along the path to solid and solid edges incident to the vertices along the path to dashed. Return the resulting path. See Figure 14.

In an actual tree, no more than one solid edge can enter a vertex but there can be any number of dashed edges entering a vertex at the same time. In the following implementation, the paths are represented as binary trees. Any binary tree variant can be used but here a simple binary tree representation is used and the splaying technique is used on the solid trees to provide a better amortized




a.) Result of expose($x$) on an actual tree.




b.) Result of expose($x$) on a virtual tree.

Figure 14. Expose operation on vertex $x$ demonstrated on both the virtual and actual tree.

efficiency. The following is the implementation of the path operations: (The code is a C language notation and is fully tested.)

The type definition used for the vertices in the implementation below is :

```
typedef struct vertextype {
                float dcost,dmin;
                int vrtx;
                struct vertextype *lt; /* left */
                struct vertextype *rt; /* right */
                struct vertextype *pt; /* parent */
                struct vertextype *sc; /* successor */
} vertex, *vertexptr;


vertexptr makepath(k)
int k;
{
char *malloc();
vertexptr v;

  v=(vertexptr) malloc(sizeof(vertex));
  v->vrtx=k;
  v->pt=NULL;
  v->lt=NULL; v->rt=NULL;
  v->dcost=0; v->dmin=0;
  return(v);
}


vertexptr findpath(v)
vertexptr v;
{
vertexptr splay();
  splay(v);
  return(v);
}


vertexptr findtail(p)
vertexptr p;
{
vertexptr splay();

   while(p->rt!=NULL)  p=p->rt;
   splay(p);
   return(p);
}
```

```
vertexptr findpathcost(p)
vertexptr p;
{
vertexptr splay();

while (p->dcost!=0 || (p->rt!=NULL && p->rt->dmin<=0) ) {
    if(p->rt!=NULL && p->rt->dmin==0)
      p=p->rt;
    else if (p->dcost>0)
      p=p->lt;
  }
  splay(p);
  return(p);
}


addpathcost(p,x)
vertexptr p;
float x;
{
  p->dmin=p->dmin + x;
}


vertexptr  join(p,v,q)
vertexptr p,v,q;
{
  v->lt=p;v->rt=q;

  if(p!=NULL) {
      p->pt=v;
      p->sc=NULL;
  }
  if(q!=NULL)  {
      q->pt=v;
      q->sc=NULL;
  }
  return(v);
}


split(v,l,r)
/* The pair returned by split is [l,r].
   These are put in the argument list */

vertexptr v,*l,*r;
{
vertexptr splay();
  splay(v);
  *l= v->lt;*r=v->rt;
  if(v->lt!=NULL) v->lt->pt=NULL;
  if(v->rt!=NULL) v->rt->pt=NULL;
   v->lt=NULL;v->rt=NULL;
}
```

The following is the implementation of expose:

```
vertexptr expose(v)
vertexptr v;
{
vertexptr p,q,r,w,s,join(),findpath();

  p=NULL;
  while(v!=NULL)  {
    s=findpath(v);
    w=s->sc;
    split(v,&q,&r);
    if (q!=NULL)  {
      q->sc=v;
      q->pt=NULL
    }
    p=join(p,v,r);
    v=w;
  }
  p->sc=NULL;
  return(p);
}
```

There are a few things that should be noted about the above implementation. The first is that the link operation link(v,w) requires v to be a root of one tree and w to be any vertex in any other tree. When the binary tree representation is used, the vertex v is not the root of the solid tree in the virtual tree but it is the vertex at the rightmost position from the root of the solid tree containing v. Notice that w is the parent of v (see definition for link operation) only in the actual tree.

The second thing is that splaying does not have to be used in the implementation. Wherever splaying is used there exists a piece of code that can replace the splaying operation and still be sufficient to carry out the operation. If splaying is used, it will provide a better amortized efficiency if there exist a few vertices that are more frequently involved in the link/cut tree operations

than other vertices.

Finally, in an actual tree there can be no more than one solid edge incident to a vertex. This is represented in the virtual tree by the fact that each vertex has only one successor and one predecessor. In the virtual tree, a binary solid tree represents the solid path in the actual tree.

The following is an implementation of the tree operations using the above path operations. The C language notation is used and the same structure definition for the path operations is also used in the following tree operations:

```
vertexptr maketree(k)
int k;
{
vertexptr v,makepath();
  v=makepath(k);
  v->sc=NULL;
  return(v);
}


vertexptr findroot(v)
vertexptr v;
{
vertexptr r;
vertexptr findtail(),expose();
  r=findtail(expose(v));
  return(r);
}


vertexptr findcost(v)
vertexptr v;
{
vertexptr p,findpathcost(),expose();
  p=findpathcost(expose(v));
  return(p);
}


addcost(v,x)
vertexptr v;
```

```
float x;
{
vertexptr expose();
  addpathcost(expose(v),x);
}
```

link(v,w)

```
vertexptr v,w;
{
vertexptr j;
vertexptr join(),expose();
  j=join(NULL,expose(v),expose(w));
  j->pt=NULL;
  w->sc=j;
}
```

cut(v)

```
vertexptr v;
{
vertexptr expose(),p,q;
  expose(v);
  split(v,&p,&q);
  if (p!=NULL) p->sc=NULL;
  if (q!=NULL) q->sc=NULL;
}
```

The above implementation has an amortized efficiency of $O(\log_2 n)$ per operation. An advantage of using this implementation is that it makes it possible to think about the link/cut tree as an abstract data type (ADT) and the link/cut tree operations as a set of operations defined over this ADT. The set of paths structure is another ADT with a new set of path operations. ADTs, in general, provide a systematic and efficient way in solving large and complex problems. It makes the implementation of the link/cut operations easier to understand since the overall implementation can be done by using different combinations of the path operations.

The expose operation acts like a backbone to this method. The purpose of expose, besides making the path from

the root to the vertex under operation solid, is to convert the solid edge, if any, entering the vertex in question to dashed edge. This is the splice operation. The number of splices executed determines the time efficiency of an operation.

Using Three Pass Splaying
as a Primary Operation

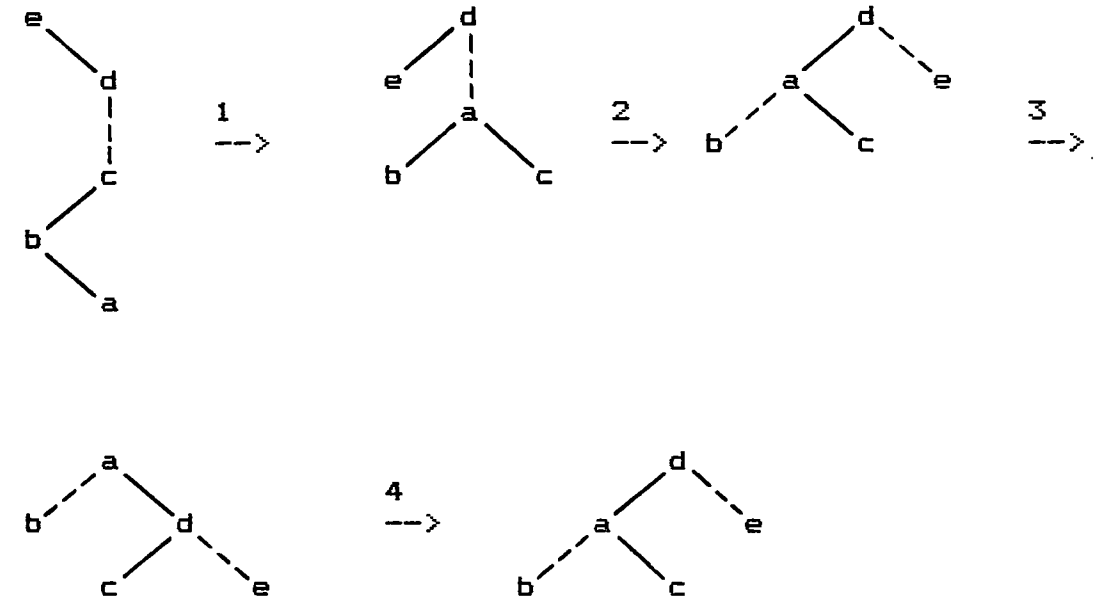
Another implementation that will provide the same order of $O(\log_{\geq}n)$ but using only the splaying technique (no path ADT) may be found easier to implement. What is special about this implementation is that it does not use the expose operation but it uses splicing as part of the splaying operation on the link/cut tree.

The following is an implementation of the link/cut tree operations using the three pass splaying, splay3: (see Figure 15)

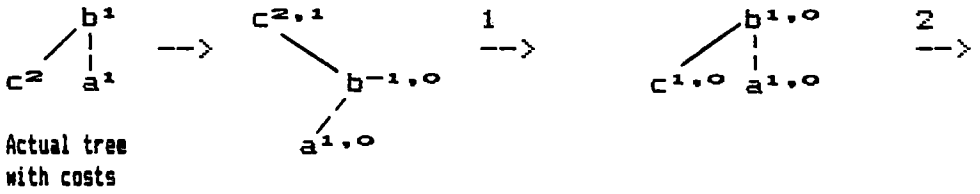
Maketree(v) Create a new tree with the node or vertex v of cost zero.

Findroot(v) Splay3(v), follow right pointers until reaching the right most vertex w, splay3(w). Return w. See Figure 15.a.

Findmin(v) Splay3(v), use the dcost and dmin fields to walk down from v to the last minimum-cost node w *after* v in the same solid subtree, splay at w, and return w. See Figure 15.b. See equations for dcost and dmin in Page 60 and 63.




a.) Findroot(a). 1. Splay at *a* and *d* each in its own solid tree. 2. Splicing operation. 3. Splay at *a*. 4. Splay at *d*, the right most vertex from a.




b.) Findmin(a). 1,2, and 3. Splay3(a). 4. Splay at *b* the vertex with minimum-cost.

Figure 15. Implementation of link/cut tree operations findroot and findmin using three pass splaying.

Addcost(v) Splay3(v), add x to dcost(v), and subtract x from dcost(left(v)) if left(v)!=null. See Figure 16.a.

Link(v,w) Splay3(v), splay3(w) and make v a middle child of w. See Figure 16.b.

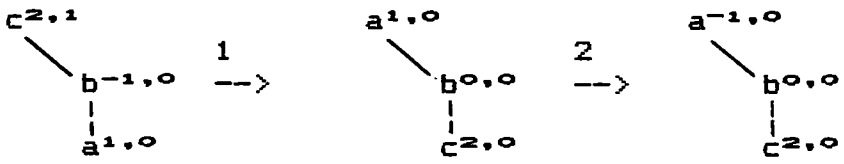
Cut(v) Splay3(v), add dcost(v) to dcost(right(v)), and break the link between v and right(v) by defining p(right(v))=null and right(v)=null. See Figure 16.c.

Findcost(v) Splay3(v) and return the value dcost of v.

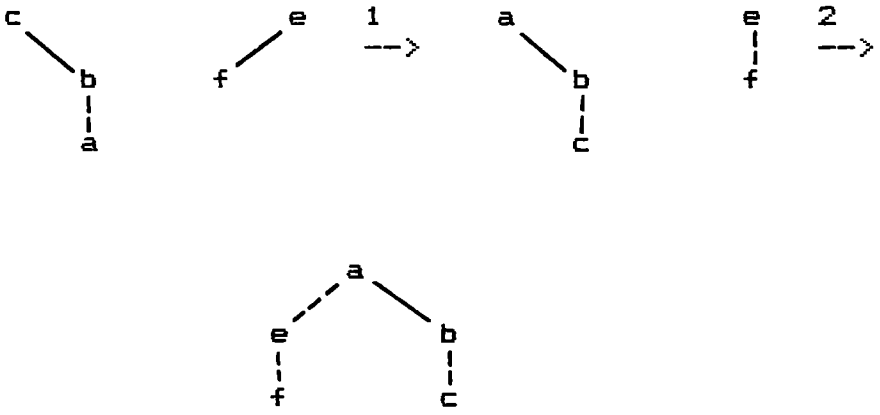
The following is a description of how to walk down the tree the find the minimum-cost when doing the findmin(v) operation. Let p be the root vertex:

```
Repeat

     if right(p)!=NULL and
     (dcost(right(p))-dmin(right(p))+dmin(p))=0   then
            p=right(p);

     if (right(p)=NULL or
     (dcost(right(p))-dmin(right(p))+dmin(p)) >0)
                                   and dmin(p) > 0 then
            p=left(p);

Until both of the above conditions are false;
{The vertex at the new p has the minimum-cost}
```

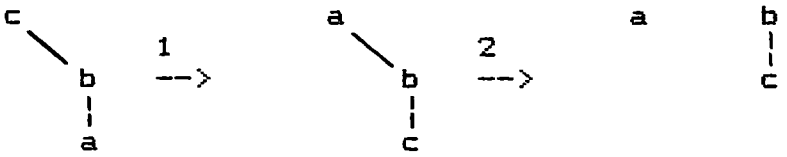
For a sequence of m operations, this technique takes an amortized time bound of $O(m*\log_2 n)$. The splaying operation used in the above implementation differs from the usual splaying operation in the sense that it may involve more than one solid tree. Another important difference also is that splaying here is a three pass operation moving a




a.) Addcost(a,-2). 1. Splay3(a). 2. Add $x$=-2 to dcost(a). If left(a)!= null then subtract $x$ from dcost(left(a)) which is not the case here.




b.) Link($e$,a). 1. Splay3($e$) and splay3(a). 2. Link $e$ to a by adding the dashed edge [$e$, a].




c.) Cut(a). 1. Splay3(a). 2. Delete the solid edge [a,$b$]. The result is two trees rooted at a and $b$.

Figure 16. Implementation of the link/cut tree operations addcost, link, and cut using the three pass splaying operation.

vertex or node all the way to the root of the virtual tree instead of only moving it to the root of its own solid tree. This splaying process was described earlier in Chapter II.

The cost and mincost functions are stored implicitly in the nodes. If cost(x) is the cost of a vertex x and mincost(x) is the minimum cost of any descendant of x in the same solid subtree then the values stored in the vertex or node x are as follows: (see Figure 17)

dcost(x) = cost(x) if x is the root of a solid tree.

or

= cost(x) - cost(parent(x)) if x is not a root of a solid tree.

dmin(x) = cost(x) - mincost(x).

Note: In the above definition dmin(x)$\geq$0 for any node x.

The values of dcost and dmin are effected by the rotation and splicing operations and thus they have to be readjusted after every rotate or splice operation. Let the vertex v have a parent w, and let a and b be the children of v before the rotation, and let b and c be the children of w after the rotation. A primed function denotes values after the rotation. Unprimed functions are the values before the rotation. The formulas that are needed to adjust these values are as follows: (See Figure 18)

dcost'(v)= dcost(v) + dcost(w),
dcost'(w)= -dcost(v),
dcost'(b)= dcost(v) + dcost(b),
dmin'(w) =
Max{0,dmin(b)-dcost'(b), dmin(c)-dcost(c)}
dmin'(v) =
Max{0, dmin(a)-dcost(a), dmin'(w)-dcost'(w)}.

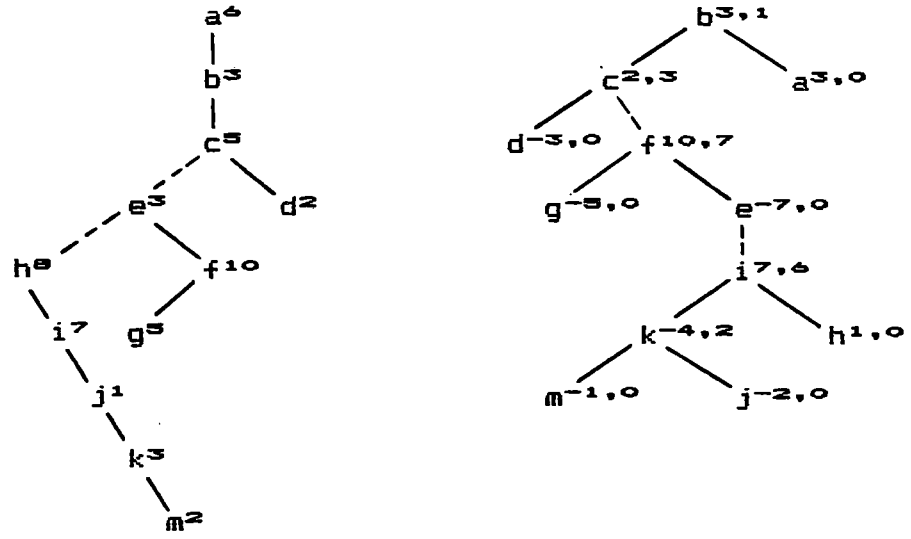



Figure 17. Link/cut tree representation. Actual tree on left with superscript representing vertex costs. Virtual tree on right with superscript representing (dcost,dmin).


Figure 18. The effect of rotation on the values of dcost and dmin of the vertices. The above is a single right rotation in a *solid* tree. Values in brackets are (dcost, dmin). Subtrees and dashed edges are skiped for clarity.

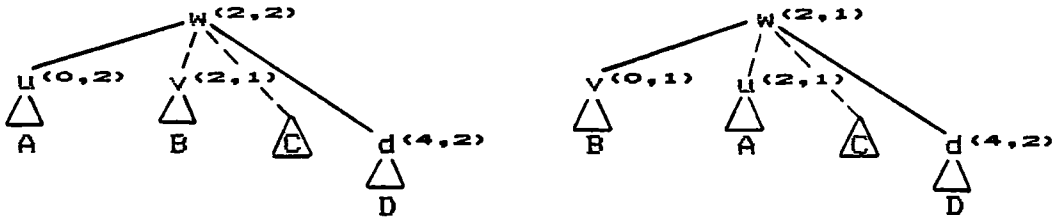



Figure 19. The effect of a splice on the values of dcost and dmin. The vertex *d* is the right(*w*). Values in brackets are (dcost,dmin). The vertices *w*, *u* and *d* are part of one solid tree before the splice operation.

The other restructuring operation that requires modifications to these values is the splicing operation. Splicing occurs in the expose operation when a solid edge entering a vertex is converted into a dashed edge and a dashed edge at the same time is converted into a solid edge. If w is the root of a solid subtree and v is any middle child of w, v becomes the left child of w, and the old left child, if any, becomes a middle child of w. Let u be the old left child of w, and let primed and unprimed functions be the values after and before the splice operation, respectively. The following are the formulas necessary to adjust these values effected by the splice operation: (See Figure 19)

```
dcost'(v)= dcost(v) - dcost(w)
dcost'(u)= dcost(u) + dcost(w)
dmin'(w) = Max
{0,dmin(v)-dcost'(v),dmin(right(w))-dcost(right(w))}.
```

To find the cost of a vertex v, either splay at v and return dcost(v) or walk along the path from the root of the solid tree containing v to v adding up dcost. Note that the value for dmin for a vertex v is :

```
Max  {0,
      dmin(left(v))-dcost(left(v)),
      dmin(right(v)- dcost(right(v))
     }
```

This method also divides a link/cut tree into paths but it has no path operations. Instead, a three pass splaying operation is used to conduct all the link/cut tree operations. If the three pass splaying operation is implemented and understood, then this method for implementing the link/cut tree operations can be

particularly simple. The storage requirements of this method are just like the method with the paths ADT; it requires a left, right, parent and successor pointers.

As may be the case with the other methods, this method has some room for improvement. With little modification to the link/cut tree operations the third pass of the splaying operation can be eliminated providing faster but more complex algorithms for the link/cut tree operations. Another substantial improvement can be achieved if parallel processing or programming is used. Each of the three passes of the splaying operation can be done in parallel. For example, the findroot(v) operation can be done by following parent and successor pointers until the root is reached, return the root then do the splaying for each solid tree in a top-down fashion and in parallel. Then for the second pass, do the splicing in parallel for each vertex on the dashed path from v to the new root of the virtual tree. Finally splay at v to complete the operation.

## Applications

Link/cut tree data structure was specifically designed to provide an efficient way to solve a practical application, namely the problem of finding a maximum flow in a network. Using the link/cut tree, an algorithm for finding a maximum flow can be done providing a time bound of $O(m*n*\log_2 n)$, better than the fastest previously known algorithm by a factor of $\log_2 n$. The maximum flow problem is presented in Chapter V.

Many more applications exist for the link/cut tree data structure. Obviously, it is a powerful tool to solve simple as well as complex problems efficiently.

# CHAPTER V

# THE MAXIMUM FLOW PROBLEM

## Introduction

Even though the link/cut tree has many applications, it was specifically designed for finding maximum flows in networks. In a network where the capacities of the branches are limited, an objective which is usually of interest is to maximize the total amount of flow from an origin to a destination and here is where maximum flow problems arise. Real life maximum flow problems include water, gas or oil through a network of pipelines; the flow of traffic through a road network; and the flow of products through a production line system. In these examples and others the branches may have different capacities, the road network is an example where a branch may have one lane while another branch may have four or more lanes leading to a higher capacity of vehicles.

In this chapter the link/cut tree data structure is used to solve the practical problem of finding a maximum flow. A maximum flow algorithm is presented and an illustration of the solution is shown to make it easy to understand the use of a link/cut tree in the solution of a maximum flow and possibly other useful problems.

## Maximum Flow Problem Definition

The maximum flow problem is that of finding a flow of maximum value from a single source to a single sink or destination in a network. The following are the properties of a flow f on a graph G:

1. The flow from a vertex v to a vertex w is equal to the negative value of the flow in the other direction; $f(v,w) = -f(w,v)$. A flow exists from v to w if $f(v,w)>0$.

2. The flow from a vertex v to a vertex w can not at any time exceed the capacity of the branch from v to w; $f(v,w)\leq cap(v,w)$. An edge [v,w] is said to be saturated if the flow is equal to the capacity of that edge.

3. Every vertex other than the source and the destination has an in flow that is equal to the out flow from that same vertex.

4. The total flow into the network through the source is equal to the total flow going out of the network through the destination vertex.

For a graph G=(V,E), where V and E are the sets of vertices and edges respectively, a cut is defined to be a partitioning of the vertex set V into two parts X and X' where X'= V-X. The set X contains the source (s) and the set X' contains the sink or destination (t). The capacity cap(X,X') of a cut is equal to the total capacities from the vertices in X to the vertices in X'. Among all possible cuts, a minimum cut is a cut with the minimum capacity. Assuming that the capacity and the flow are zero from v to w if there is no edge connecting the two vertices, then a cut has a flow that is equal to the total flow from all vertices in X to all vertices in X'. A theorem called min-cut

theorem states that the capacity of a minimum cut is equal to the maximum flow for a given network.

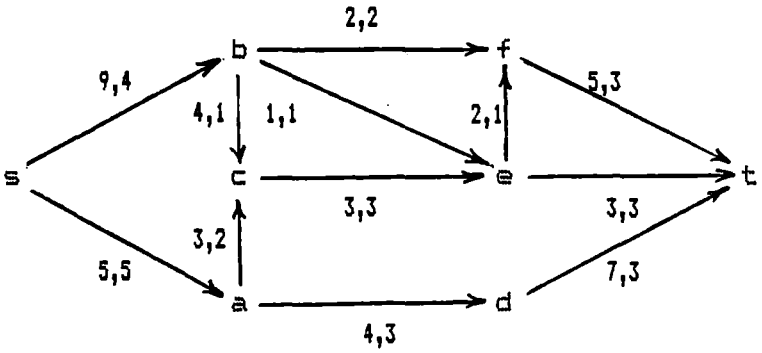
The residual capacity for an edge [v,w] of flow f is defined by res(v,w) = cap(v,w) - f(v,w). The residual capacity is an indication of how much flow can be pushed in the direction [v,w] by increasing the flow in the direction from v to w and decreasing the flow in the opposite direction w to v. Figure 20 shows the residual graph R for a given flow. The residual graph contains the same set of vertices in the flow with edges [v,w] of capacities equal to res(v,w) in the direction v to w and capacities of f(v,w) in the direction w to v. A path P in R from s to t is an augmenting path for f.

## Finding a Maximum Flow

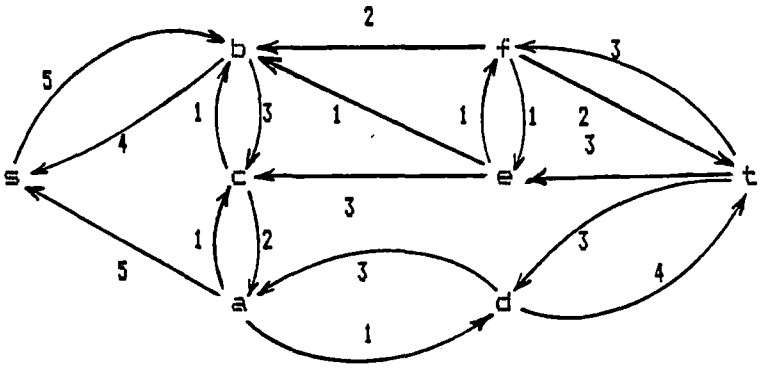
Before introducing the $O(m*n*(\log_2 n))$ solution for the maximum flow problem, it is helpful to present the old methods to give a better feeling of how this may be approached. A method by Ford and Fulkerson (5,6) is considered to be the simplest and is described below. Another method by Dinic (3) resembles the solution using the link/cut tree data structure but does not have the same time bound.

### Augmenting Path Method

Let G be a graph and G' be a subgraph of G such that $f'(v,w) > 0$ in G'. Let $P_i$ be the paths found from s to t in G'. The following is an outline of the algorithm for




a.) Original graph. Each edge has an assigned capacity and flow respectively. The edge [*e,t*] is one out of five saturated edges in the above graph.




b.) Residual graph for the network in (a). The path [*s,b,c,a,d,t*] is an augmenting path with residual capacity of 1.

Figure 20. Residual graph for a given flow.

this method:

1. Let i=1.
2. Find an augmenting path $P_i$ (a path from s to t) of maximum residual capacity in G'. If no path exists from s to t then halt.
3. Let $n_i$ be the minimum flow f'(v,w) of an edge [v,w] along $P_i$.
4. Decrease the flow f'(v,w) for every edge along $P_i$ by $n_i$.
5. Delete all edges [v,w] along $P_i$ if the flow f'(v,w)=0.
6. Increment i by one. Go to step number 2.

When choosing an augmenting path, that path must have the maximum residual capacity, otherwise this method may halt before finding a maximum flow. Table VII shows a trace of the above algorithm when applied to the graph of Figure 20.a starting with the zero flow. When the algorithm halts, a maximum flow is obtained by summing up the units of flow $n_i$ forced along its respective path $P_i$. Each time step number 5 is executed and edge is deleted from G' and thus the algorithm halts after at most m loops where m is the number of edges in the original graph G.

Dinic's Algorithm for Finding a Maximum Flow

Dinic's algorithm uses two new concepts that need to be presented. The first concept is called a blocking flow. A blocking flow f is a flow in G with all paths from s to t containing at least one saturated edge. Having a blocking flow does not necessarily mean having a maximum flow since the flow may be increased by rerouting even when there is a

TABLE VII

A TRACE OF THE AUGMENTING PATH METHOD
FOR FINDING A MAXIMUM FLOW

| Path $P_i$ | $n_i$ | Edge deleted from G |
|---|---|---|
| [s,a,d,t] | 4 | [a,d] |
| [s,b,c,e,t] | 3 | [c,e],[e,t] |
| [s,b,f,t] | 2 | [b,f] |
| [s,b,e,f,t] | 1 | [b,e] |

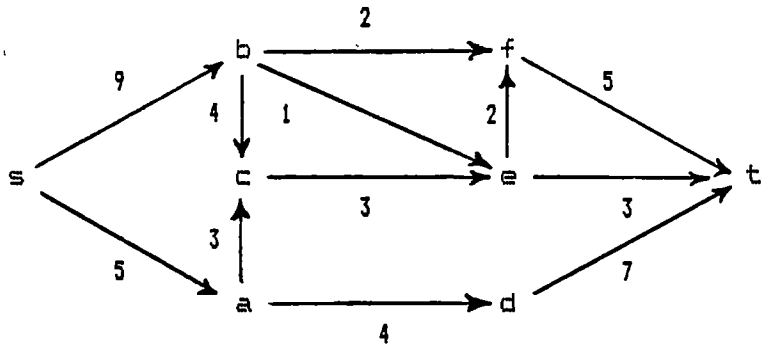
The value for the maximum flow is found by adding all $n_i$. In this example, a maximum flow of ten is obtained. Notice that augmenting paths with maximum residual capacity are choosen first. This gurantees a maximum flow when the algorithm halts. The algorithm halts if no more augmenting paths can be found in G'.

blocking flow. For example, Figure 20.a shows a graph with a blocking flow but not with a maximum flow. The other concept is called the level graph. Let R be the residual graph for f. The level of a vertex v, level(v), is equal to the shortest path from s to v in the residual graph R. The level graph L for f is the subgraph of R such that L contains only the edges [v,w] in R for which level(w)=level(v)+1. The level graph L contains every shortest augmenting path. This can be constructed using breadth-first search.

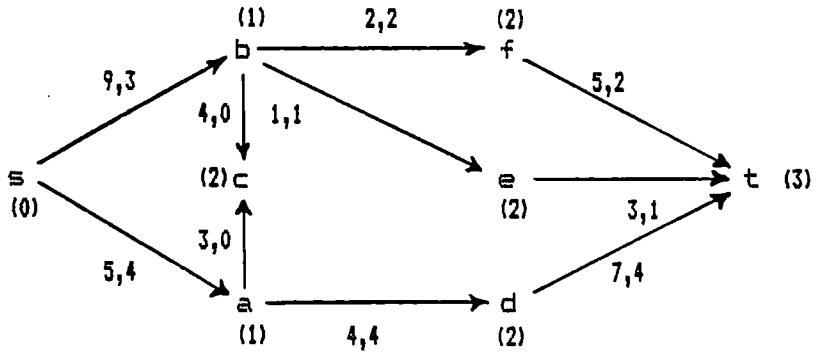
At this point Dinic's algorithm for finding a maximum flow can be presented. A way for finding a blocking flow is needed to complete the following algorithm. The algorithm starts with the zero flow (f(v,w) = 0 for all edges [v,w]) and is outlined below (See Figure 21):

1. Find a blocking flow f' on the level graph for the current flow f.
2. Add the flow in f' to the original flow f such that the new flow f(v,w) = f(v,w) + f'(v,w).
3. If the destination vertex t is in the current level graph then go to step 1, otherwise halt.

If n is the number of vertices in a graph then the above algorithm needs at most n-1 loops to find a maximum flow. The way the augmenting paths are chosen in this algorithm is such that shortest augmenting paths are chosen first. A path length is defined by the number of edges along the path. As suggested by Dinic (3), it is most efficient to augment along paths of same length simultaneously.

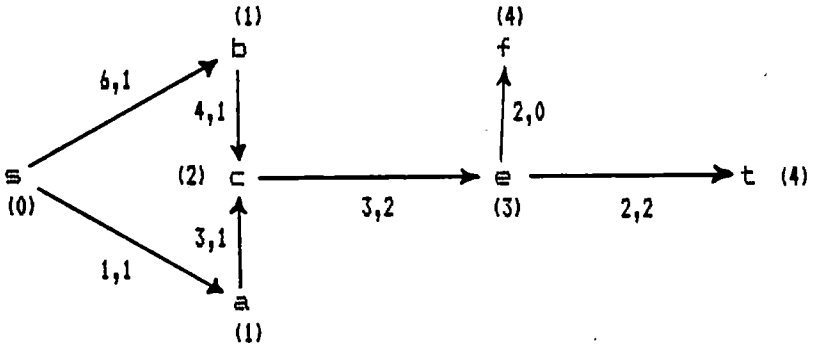



a.) Original graph. Values on edges represent the flow capacities.

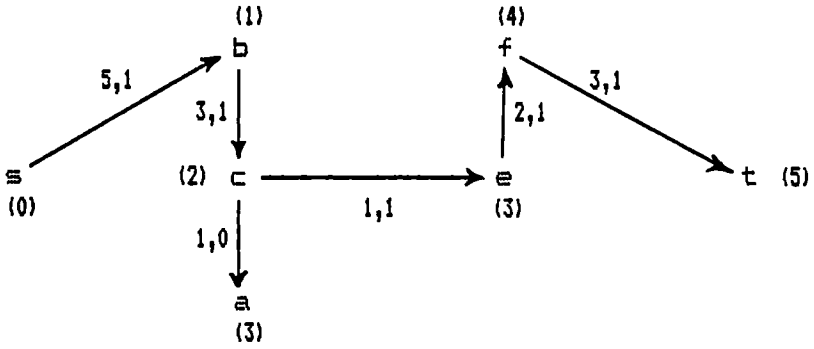



b.) First level graph with a blocking flow. The two values on each edge are the reseidual capacity and the flow, respectively. Values in parentheses are the levels of the vertices.

Figure 21. Dinic's algorithm for finding a maximum flow. There is no fourth level since there is no path from *s* to *t* in the residual graph R after the third level graph.

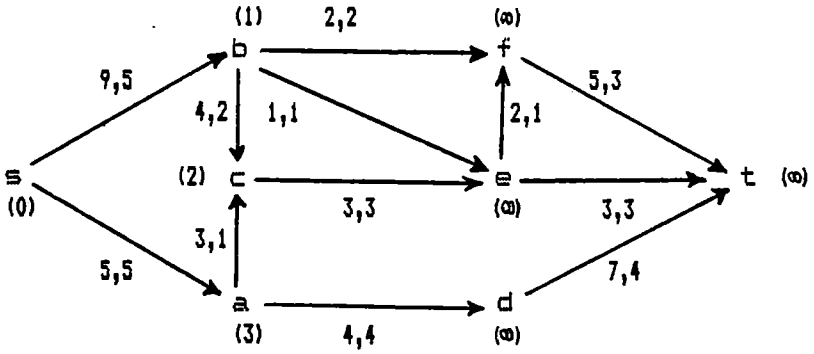



c.) Second level graph with a blocking flow.




d.) Third level graph with blocking flow.

Figure 21. (Continued)




e.) Final flow. Maximum flow is 10 and [*s,a,b,c*],[*d,e,f,t*] is a Minimum. cut.

Figure 21. (Continued)

Finding a Blocking Flow

There are many known methods for finding a blocking flow for an acyclic network. In this section two methods are presented. The first is by Dinic and the second is suggested by Sleator and Tarjan using the link/cut tree data structure.

Dinic's algorithm finds any path from s to t, then saturates at least one edge by pushing enough flow through that path. Afterwards, it deletes all saturated edges and repeats the process until t is not reachable from s. The following is a more detailed outline of this method:

```
Initialize:
    Let path P contain the vertex s; P=[s].
    Let v=s.
    Go to  Advance.


Advance:
    If there is no edge out of v then go to retreat.
    else
         let [v,w] be an edge out of v.
         Add the vertex w to P.
         Let v=w.
    If w=t then go to Augment. else repeat advance.


Augment:
    Let  d  be  the  minimum  of cap(v,w)-f(v,w) for an
    edge [v,w] along the path P.
    Add d to the flow of every edge on P.
    Delete from G all newly saturated edges.
    Go to  Initialize.


Retreat:
    If v=s then halt.
    else
         Let [u,v] be the last edge on p.
         Delete v from P.
         Delete [u,v] from G.
         Let v=u.
         Go to  Advance.
```

Table VIII shows the trace of the above algorithm when applied to the graph in Figure 21.b. Each step of initialize, advance, or retreat takes a constant time and each augment step takes O(n) time. Each augment step deletes at least one edge and each retreat step deletes one edge at a time and thus there are at most m such steps. There are at most m+1 initialize steps all but the first coming after advance steps. Since at most n-1 advance steps are executed before augmenting or retreating, thus there are at most (n-1)m advance steps. Putting all results together, it follows that Dinic's algorithm finds a blocking flow in O(n*m) time. Since it takes n blocking steps to find a maximum flow then the overall process takes $O(n^2m)$ time to find a maximum flow.

The other method for finding a blocking flow resembles the one by Dinic in the sense that it saturates at least one edge along a path at a time. The method uses the link/cut tree data structure to reduce the time needed per edge saturation. The result is an algorithm with a time bound of $O(m*\log_2 n)$ for finding a blocking flow and thus an $O(m*n*\log_2 n)$ for finding a maximum flow where n and m are the number of vertices and edges in the network, respectively.

Using the link/cut tree operations presented in chapter IV an algorithm for finding a blocking flow can be done. The algorithm maintains for each vertex v a current edge [v,parent(v)] on which it may be possible to increase the flow. The costs associated with the vertices are

TABLE VIII

A TRACE OF THE FLOW OF THE ALGORITHM
BY DINIC FOR FINDING A
BLOCKING FLOW

| Path P | d | Edge deleted from G | Operation |
|---|---|---|---|
| [s,b,e,t] | 1 | [b,e] | Advance & Augment |
| [s,b,f,t] | 2 | [b,f] | Advance & Augment |
| [s,a,d,t] | 4 | [a,d] | Advance & Augment |
| [s,b,c] | - | [b,c] | Advance & Retreat |
| [s,b] | - | [s,b] | Retreat |
| [s,a,c] | - | [a,c] | Advance & Retreat |
| [s,a] | - | [s,a] | Retreat |
| [s] | - | --- | halt. |

The minimum capacity $d$ along a path P is added to the flow of every edge in P. For example, after the first step the flow of each edge $[s,b]$, $[b,e]$, and $[e,t]$ is increased by a value of 1. The path P is reset to $[s]$ after every step.

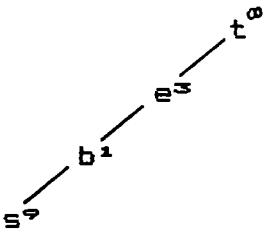
defined as cap(v,parent(v))−f(v,parent(v)) for all vertices other that solid tree roots. If v is a solid tree root then the cost is defined to be *high*, where *high* is a number that is higher than the total capacities of all edges in the graph. The following is the outline for the algorithm using link/cut tree data structure to find a blocking flow, it might be very helpful if this algorithm is compared step by step with Dinic's algorithm:

```
Initialize:
     for all vertices do
          maketree(v)
          addcost(v,high).
     Go to  Advance.


Advance:
     Let v=findroot(s).
     If v=t then Go to  Augment.
     If there is no edge out of v then go to retreat.
     else
          Let [v,w] be an edge out of v.
          Addcost(v,cap(v,w)−high).
          Link(v,w).
          Let p(v)=w.
          if w=t then go to augment.
          else repeat advance.


Augment:
     Let [v,d]=findcost(s).
     Addcost(s,−d).
     Go to Delete.


Delete:
     Cut(v).
     Addcost(v,high).

     Let f(v,p(v))=cap(v,p(v)).
     Delete the edge [v,p(v)] from the graph G.
     Let [v,d]=findcost(s).

     If d=0 then repeat Delete.
     else go to Advance.
```

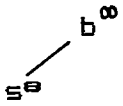
```
Retreat:
     If v=s then Go to Terminate.
     else
          For all edges [u,v] do
               Delete [u,v] from the graph G.
               If p(u) <> v then f(u,v)=0.
               else
                    Cut(u).
                    Let [u,d]=findcost(u).
                    Addcost(u,high-d).
                    Let f(u,v)=cap(u,v) - d.
          Go to Advance.


Terminate:
     For every undeleted edge [u,v] do
          If p(u) <> v then f(u,v)=0
          else
               Cut(u).
               Let [u,d]=findcost(u).
               Addcost(u,high-d).
               Let f(u,v)=cap(u,v) - d.
```

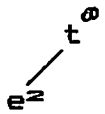
Figure 22 illustrates the effect of the above algorithm when applied to the graph in Figure 21.b. Notice that the abstract version (without the sets of paths representation) of the link/cut tree is used in the figure instead of using paths to represent trees for simplicity. Each tree operation takes an $O(\log_2 n)$ and since there are $O(m)$ tree operations in the algorithm thus the overall algorithm takes $O(m*\log_2 n)$. Using this algorithm to find a maximum flow, a time bound of $O(m*n*\log_2 n)$ can be obtained.

Finally, it is clear that the link/cut tree is a powerful technique that may be found useful in many network algorithms as it is in the maximum flow problem. The link/cut tree data structure was used simply to minimize the time needed to construct a saturated edge in the case of finding a blocking flow and was not meant to change the

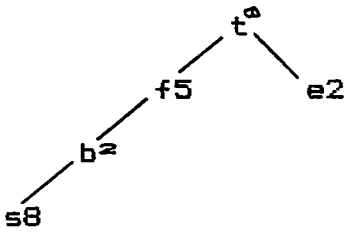



a.) Advancing.

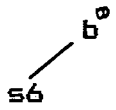


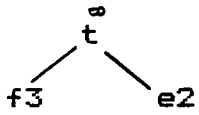




b.) Augment then Delete. Delete $[b,e]$ from G and let its flow be $f[b,e]=cap[b,e]=1$.

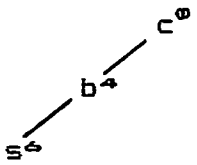


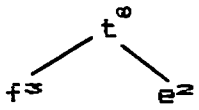

c.) Advancing




d.) Augment and delete. Delete $[b,f]$. Let $f[b,f]=2$.

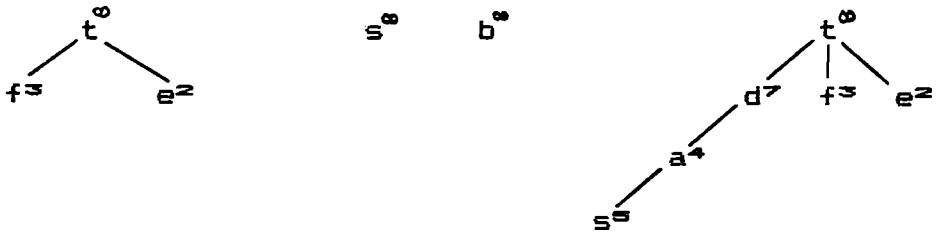



e.) Advancing.

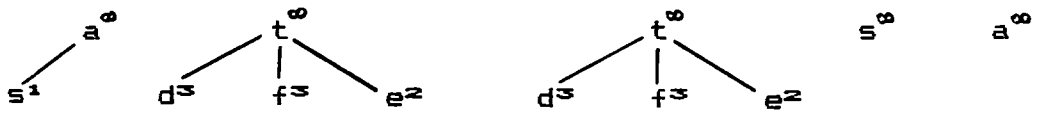
Figure 22. A trace of the algorithm for finding a blocking flow using the link/cut tree data structure. Values on vertices are costs. Any implementation of the link/cut tree operations can be used. Terminating step deletes $[d,t]$, $[f,t]$ and $[e,t]$ and set the flows $f[d,t]=7-3=4$, $f[f,t]=5-3=2$ and $f[e,t]=3-1=2$.


f.) Retreating since there is no edge out of c in G. $[u,v]$ = $[b,c]$, $[a,c]$. $f[b,c]=cap[b,c] - 4 = 0$. The flow $f[a,c]= 0$ since $p(a) <> c$. Delete all $[u,v]$.




g.) Retreating. No Edge out of $b$ in G. $f[s,b] = 9-6 = 3$. Delete $[s,b]$.

h.) Advance.




i.) Augment. Delete the edge $[a,d]$ Let $f[a,d]=4$.

j.) Retreat since there is no edge out of a. $f[s,a]= 5-1= 4$. Delete $[s,a]$.

Figure 22. (Continued)

overall process used by Dinic to find a blocking flow.

# CHAPTER VI

## SUMMARY AND CONCLUSIONS

The main topic investigated and discussed in the previous chapters is the idea of using self-adjusting data structures. Self-adjusting data structures provide ways to achieve amortized efficiency which is the average time per operation on a given data structure over a worst case sequence of operations. Splaying, the main operation used in self-adjusting binary trees, was studied and new methods were presented to perform splaying in a top-down fashion. These new methods along with the original splaying operation by Tarjan were empirically evaluated. Also, the effects of having different frequency distributions of the items in the tree and the effect of having a certain percent of the total number of items in the tree with higher probability of being accessed are shown from the results of the empirical study on the splaying operation.

Two new methods to move a node to the root position are presented in Chapter III. These methods are considered to be an improvement over the original methods, move-to-root and splaying. The first has the exact effect of move-to-root but in a top-down fashion and in a much simpler and more efficient method. The other is similar in effect to top-down splaying but also much simpler and always as

good as the top-down splaying described by Tarjan.

Splaying provides most efficient results when few items in the tree have higher probability of being accessed than other items in the same tree. If this is the case then splaying will be more time efficient in an amortized sense than the balanced tree data structure which is known to provide a time bound of $O(\log_2 n)$ in the worst case. The splay tree data structure also allows the access frequencies of the items in the tree to change dynamically during the accesses which gives the splay tree an advantage over the optimum tree.

In applications where the access frequency of an item decreases after being accessed, splaying is not effective. In such cases an attempt was made to move a node to the leaf level thus giving way to other items in the tree to go up the tree and shorten the access path. Many methods were applied to move a node to the leaf level. These methods include single rotations in all different orders, merging the left and right subtrees, and merging the rightmost path of the left subtree and the leftmost path of the right subtree. Unfortunately, these methods only works when the number of items in the tree is small but for large numbers the results are disastrous. The reason is that it is difficult if not impossible to move a node to the leaf level systematically without extending the total path length of the nodes in a tree. The same idea was discussed in a list form of a self-adjusting data structures and shown to be effective.

Perhaps one of the most important features of splaying is that it allows the use of parallel programming or processing. Parallelism can be applied in the top-down splaying method where restructuring takes place during the searching process. Also, the same idea applies to the new top-down version of move-to-root presented in Chapter III.

The linking and cutting tree problem can be solved by applying the operations maketree, findroot, findcost, addcost, link, and cut. The tree over which these operations are defined is called the link/cut tree. Different ways are available to solve this problem but not all are efficient if amortization is the objective. One method to solve this problem is by assigning parent pointers to each vertex. The implementation is simple but since the order of findroot, findcost, addcost is O(n) in the worst case this method is not time efficient. The other alternative is to divide the link/cut tree into sets of paths and a new set of path operations is defined over the sets of paths. These path operations are used to implement the link/cut tree operations. Splaying proves to be an efficient way to implement the path operations. The use of the path operations can be avoided if a special type of splaying is used directly to apply to link/cut tree operations. This type of splaying is a three pass splaying process that uses splicing as a part of the process. The amortized time bound in either case is $O(\log_2 n)$ per operation and the amortized time bound for a sequence of m operations is $O(m*\log_2 n)$. An implementation of the

link/cut tree operations illustrating the use of splaying in these implementations is presented in Chapter IV.

The link/cut tree data structure was specifically designed for the practical application of solving maximum and minimum flow problems. Using the link/cut tree data structure to solve the maximum flow problem provides an algorithm with a time bound of $O(m*n*\log_2 n)$, where m is the number of tree operations used to find a maximum flow and n is the number of vertices in the network. This time bound is better than the fastest known algorithm by a factor of $\log_2 n$. Beside the applications in network flow algorithms, link/cut tree data structure may be used to find nearest common ancestors and in computing constrained minimum spanning trees.

# BIBLIOGRAPHY


(1) Allen, B. and Munro, I. "Self-Organizing Binary Search Trees." JACM, Vol. 25, No. 4, (October, 1978), pp. 526-535.

(2) Bent, S.W., Sleator, D.D, and Tarjan, R.E. "Biased Search Trees." SIAM J. COMPUT., Vol. 14, No. 3, (August, 1985), pp 545-568.

(3) Dinic, E.A. "Algorithm for solution of a problem of maximum flow in a network with power estimation" Soviet Math. Dokl., No. 11, (1970), pp. 1277-1280.

(4) Feigenbaum, J. and Tarjan, R.E. "Two New Kinds of Biased Search Trees." Bell System Tech. J., Vol. 62, No. 10, (December, 1983), pp. 3139-3158.

(5) Ford, L.R. and Fulkerson D.R. "Maximal flow through a network" Canad. J. Math., No. 8, (1956), pp. 399-404.

(6) Ford, L.R. and Fulkerson D.R. "Flows in Networks" Princeton Univ. Press, Princeton, NJ, (1962).

(7) Galil, Z. and Naamad, A. "An $O(e*v*\log_2 v)$ algorithm for the maximal flow problem." J. Comput. System Sci., Vol. 21 (1980), pp. 203-217.

(8) Guting, H. and Kriegel. H.P "Multidimensional B-tree: An efficient dynamic file structure for Informatik-Fachberichte." In proceedings of the 10th Gesellshaft fur Informatik Annual Conference. Informatik-Fachberichte , Vol. 33. Springer-Verlag, New York, 1980, pp. 375-388.

(9) Mehlhorn, K. "Dynamic binary search trees" SIAM J. Comput., Vol. 8, No. 2 (1979), pp. 175-198.

(10) Shiloach, Y. "An $O(n*I*\log_2 I)$ maximum-flow algorithm" Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford Univ., Stanford, CA, (1978).

(11) Sleator, D.D. and Tarjan, R.E. "Amortized efficiency of list update and paging rules." Commun. ACM, 23,2 (Feb. 1985), pp202-208.

(12) Sleator, D.D. "An O(nm log n) algorithm for maximum network flow" Tech. Rep. STAN-CS-80-831, Computer Science Dept, Stanford Univ., Stanford, CA, (1980).

(13) Sleator, D.D. and Tarjan, R.E. "A Data Structure For Dynamic Trees." J. Comp. Syst. Sci., Vol. 26, (1983), pp 362-391.

(14) Sleator, D.D. and Tarjan, R.E. "Self Adjusting Binary Search Trees." JACM, Vol. 32, No. 3, (July, 1985), pp. 652-686.

(15) Tarjan, R.E. "Data Structures And Network Algorithms" Regional Conference Series in Applied mathematics, pp. 45-111.

# VITA

Ziad Ishaq Rida

Candidate for the Degree of

Master of Science

Thesis: SPLAYING IN THE IMPLEMENTATION OF THE LINK/CUT TREE OPERATIONS USED IN SOLVING THE MAXIMUM FLOW PROBLEM

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Amman, Jordan, July 3, 1963.

Education: Graduated form Terra Santa High School, Amman, Jordan, in June, 1980; received Bachelor of Science degree in Electrical engineering from Oklahoma State University, May, 1984; completed requirements for the Master of Science degree at Oklahoma State University in December 1986.

Professional Experience: Applications programmer, Pan-Arab-Computer Center, Amman, Jordan, May, 1984-August 1984 and May, 1985- August, 1985; graduate teaching assistant, Math Department, Oklahoma State University, Fall 1985; member of ACM, 1984.