Class Project : CIS 600

Advanced Data Structures in Computational Geometry

K. Sivaramakrishna & Sachin More

May 16, 1995

Contents

List of Figures

Chapter 1

Introduction

The project involved studying the advanced data structures used in computational geometry. The following three data structures were considered :

- Fractional Cascading : Fractional cascading is a technique used to speed up searches that involve repeated searching of the same key in distinct ordered lists. This data structure is discussed in Chapter 2.
- Balanced Binary Search Trees : This data structure in Chapter 3.
- Finger Trees : Finger trees are used to maintain an ordered list in which given an object in the list, another object can be located in time proportional to the logarithm of the difference in their rank. This data structure is discussed in Chapter 4.

The rest of the chapters are organized as : They start with the definition of a problem in computational geometry. It is followed by a brief description of the algorithm and the necessary data structures. Next comes basic definitions and operations for the data structure. Finally the complexities of various operations on the data structure are analyzed.

Chapter 2

Fractional Cascading

2.1 Interval Stabbing Problem

The *Interval Stabbing Problem* is defined as :

Given a collection of n intervals on the real line with distinct endpoints, organize them in to a data structure so that the number of them containing an arbitrary query point can be efficiently computed.

The following section describes an algorithm that uses $O(n)$ space and has query time $O(logn)$[1].

2.2 Algorithm and Data Structures

The algorithm stores the intervals in the nodes of a binary search tree as ordered lists and each interval appears exactly once in the tree.

The searching essentially involves traveling down the search path in the tree and searching the ordered lists associated with each node (See figure 2.1).
The algorithm to construct the tree is as follows :

1. Sort the $2n$ end points of the n intervals.
2. Find the median of the end points. Store the value of the median in the root node as the key value.
3. Find out the intervals that *contain* the median.

[1] All logarithms are assumed to have base 2.

4. Keep the sorted left end points of these intervals in one list and right end points in another list. Let's call the lists L and R respectively.
5. Divide the remaining intervals in two sets, $S1$ and $S2$. $S1$ contains intervals that are to the $left$ of the $keyvalue$ and $S2$ contains intervals that are to the $right$ of the $keyvalue$.
6. Repeat the above steps for $S1$ and $S2$.

The algorithm to search the tree is :

1. Compare the query point with the key value in the node.
2. If the key value is greater than the query point search the L list using binary search. All the intervals whose left end-points are less than the key value contain the query points. We then go down the left sub-tree and the repeat the algorithm.
3. If the key value is less than the query point search the R list using binary search. All the intervals whose right end-points are greater than the key value contain the query points. We then go down the right sub-tree and repeat the algorithm.

2.2.1 Space Time Analysis

Since we split the list of intervals in half at each level and no interval is repeated, the height of the tree is $O(logn)$. Since each interval is stored only once in the tree, storage requirement is $O(n)$.

The preprocessing can be carried out in $O(nlogn)$ time. Since we perform a binary search at each level and the traverse only one path down the tree the total complexity of searching is only $O(log^2n)$. The search time can be lowered using a method known as *fractional cascading.*

2.3 Fractional Cascading

Fractional Cascading is used to speed up repeated searches of the same query points in different ordered lists. The the interval stabbing problem discussed above, the solution is found by repeated searches among ordered lists associated with each node on the search path as we move down the search tree.

This method involves migrating a fraction of the portion of each ordered list to the preceding node along the search path. This sample of the points is then merged with the ordered list of the preceding node. The process starts at the end-points of each search path and is carried out repeatedly till the starting node of the search path is reached.

In the searching process, the list associated with the starting node is searched as usual. When the search is over, not only we know the solution points associated with the starting node, we can also find out the solution points associated with the query point in the next node along the search path at only a small additional cost. This cost is less than the cost associated with searching the ordered list anew.

Here is what is done in the interval stabbing problem :

Each leaf node selects every alternate points in L and R and sends them to the parent node. The parent node receives four such lists, L_1 and R_1 from the left child and L_2 and R_2 from the right child. It then merges L_1 and L_2 with its own list L and R_1 and R_2 with R. While merging it keeps track of which point came from which child and the index of that point inside the ordered list of the child. See figure 2.2. Only the left list L from each node is shown in the figure. This process terminates at the root node. This results in each node containing a sample of points from its children, a smaller sample from the grandchildren, and so on.

The search process starts at the root node. The list at the root node is searched as usual. When the search is over, we know the exact position of the query point in the original list of that node. By referring to the extra position values maintained in the node and making one comparison in the list associated with the child node, the exact position in the list associated with the child node can be found out. This process is repeated as one proceeds along the search path.

2.4 Complexity Analysis

The preprocessing time remains $O(nlogn)$. In the searching process, a binary search is done initially followed by a constant cost of one comparison per node on the search path. The height of the search tree is $O(logn)$. Hence the total cost is $O(logn)$.

Without fractional cascading the space requirement is $O(n)$. Each node then sends half the elements from the his list to its parent. The parent in turn sends half of the augmented list list to its parent. At the end of the process each list contributes some extra elements in the lists associated with its parent, grandparent and so on. Number of extra elements contributed by each list form a geometric series series as one moves up the tree. Hence the this extra overhead is linear in magnitude to the original length of the list. So, the space requirement remains $O(n)$.

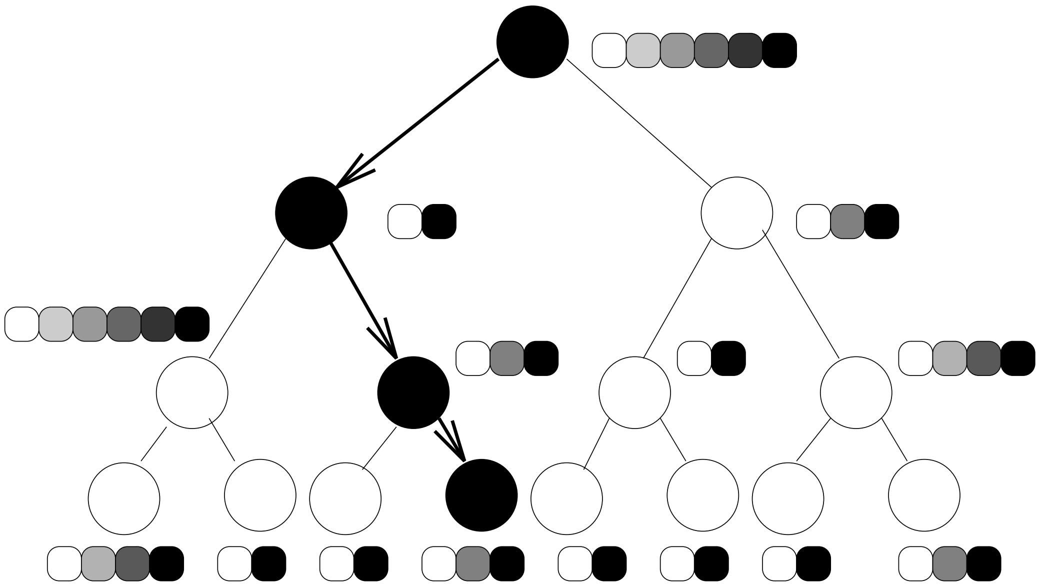


Figure 2.1: Search path in a binary search tree annotated with ordered lists

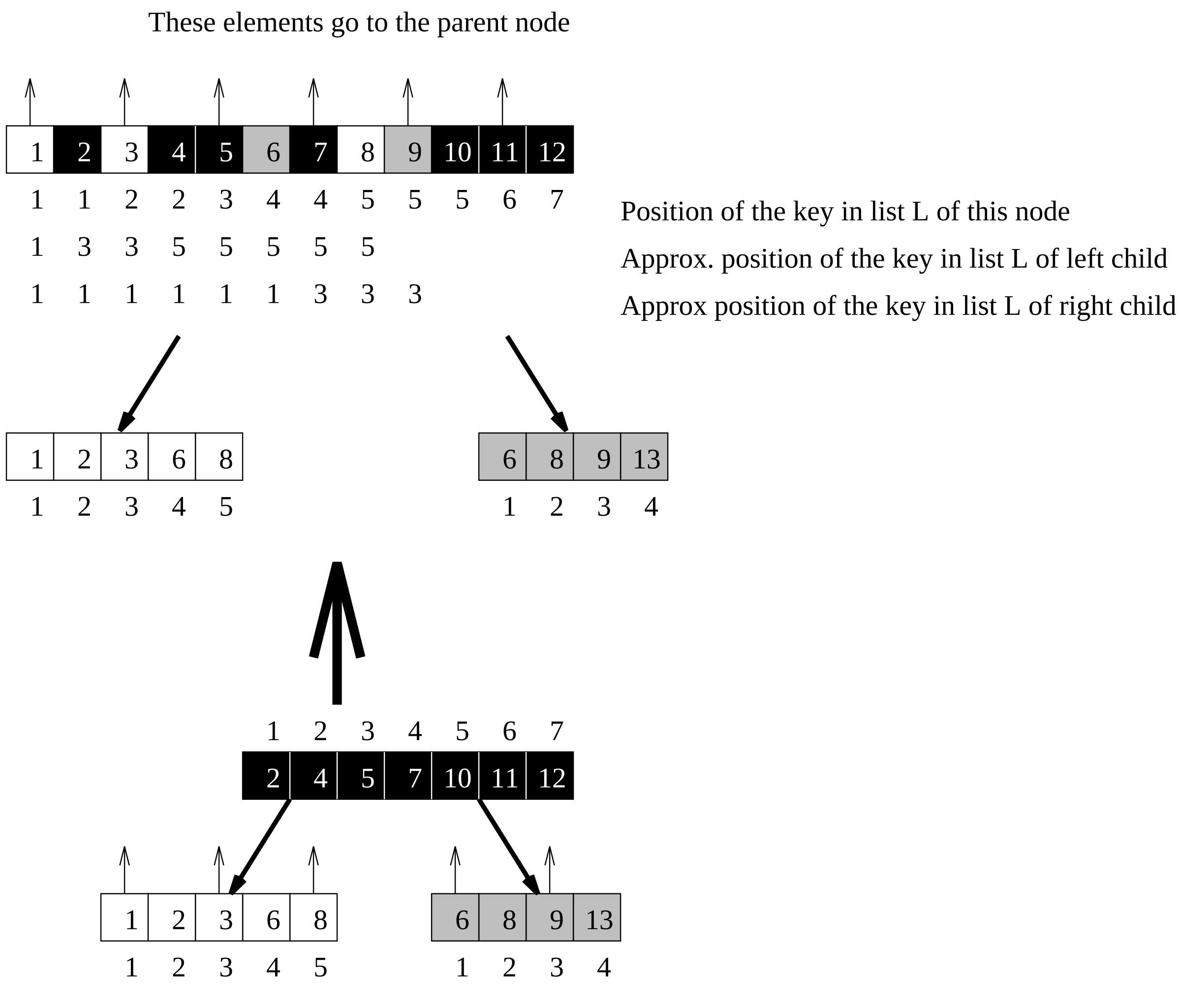


Figure 2.2: Sample List Migration in Fractional Cascading

Chapter 3

Persistent Search Trees

3.1 Planar Point Location Problem

The *Planar Point Location Problem* is defined as :

Given a planar subdivision induced by n line segments and a sequence of points, the polygon containing each point has to be located.

There are several algorithms which solve this problem in $O(logn)$ time and $O(n)$ space, but they are all rather complicated. By using persistent search trees, this problem can be solved in $O(logn)$ time and $O(n)$ space in a very easy manner. First the solution for Planar point location using plane sweep technique is discussed. And then a persistent search tree, which reduces the space requirement for the algorithm is discussed.

3.1.1 The plane sweep technique

In this algorithm a vertical line is drawn from every vertex in the planar subdivision. This splits the plane into vertical slabs. Refer to fig 3.1. Since in a planar subdivision the edges do not cross each other, the line segments within each slab are totally ordered *i.e* given two line segments in a slab, one of the line segments is totally above the other. Associate with every line segment the polygon above it. Now we can locate a query point in two binary searches .

1. Search on x coordinate. This locates the slab in which the query point lies.

2. Search within each slab for the line segment associates with that polygon. (*We can introduce a line segment running from* $(-\infty\ ,\ -\infty)$ *to* $(\infty\ ,\ -\infty)$, *thereby guaranteeing that there is a line segment below every polygon in the slab*). This search gives the face in which the query point lies.

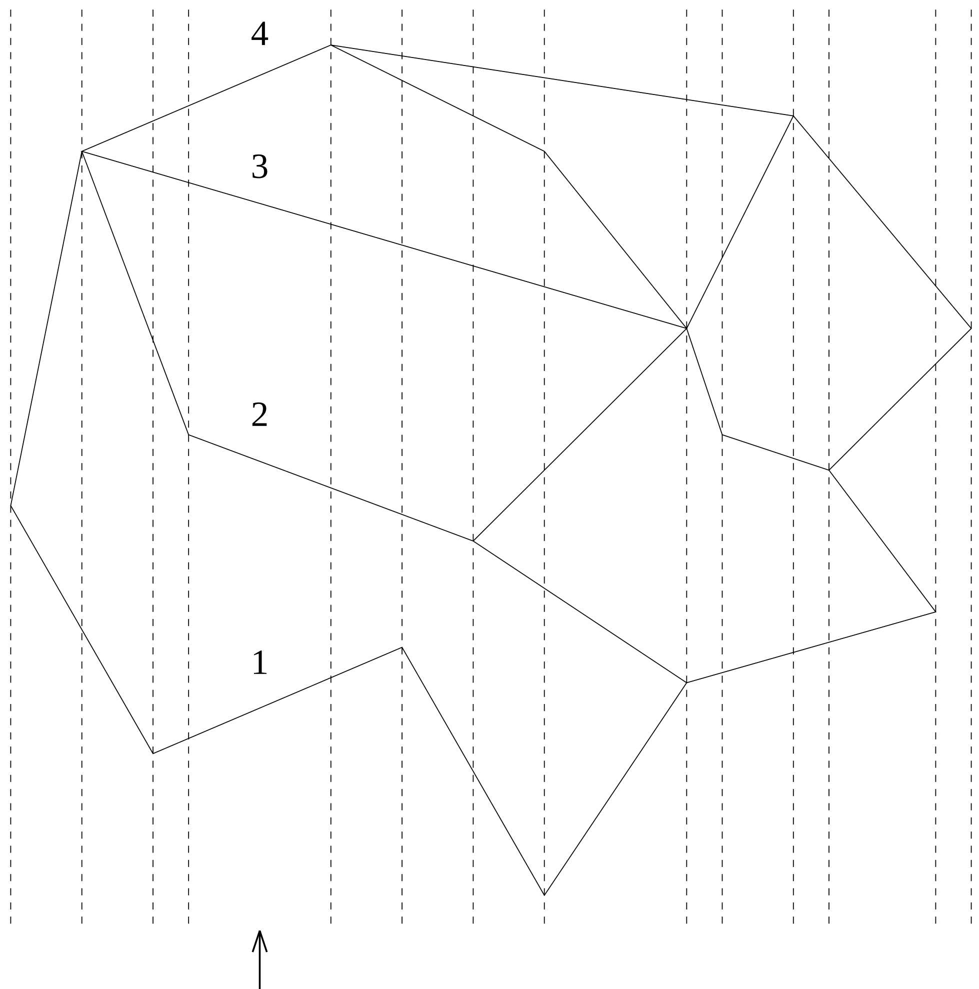


Figure 3.1: The slabs of a planar subdivision

Since both the binary searches are $O(logn)$, the time requirement for the query point location is $O(logn)$. The only way the point location can be done by the binary searches is by somehow storing the order of the line segments within each slab. If separate binary search tree for each slab has to be built, the worst case space requirement can be $O(nm)$.(*Where m is the total number of time stamps considered*).

This space bound can be reduced because the sets of line segments intersecting the adjacent slabs are similar. The x coordinate can be considered as time. As the boundary from one slab to the next is crossed, certain segments are deleted from the set and the other segments are inserted. Over the entire time range there are $2n$ insertions and deletions. (*One for each line segment in the input*).

So now the problem has been reduced to storing a sorted set in a search tree so that query can be done in the past. Insertions are not needed in the past because we know exactly when events occurs before they occur.

3.2 Persistent search trees

This is a kind of search tree which supports queries and updates not only in the present but also in the past versions unlike the normal tree, which allows those operations only in the present.

For the plane sweep technique discussed above a persistent Red-Black can be used. Since no updates are done in the past versions the persistent search tree for this application need not support updates in the past.

A Red-Black tree has the following properties

1. All missing(external) nodes are regarded as black.
2. All paths from root to a missing node contain the same number of black nodes.
3. Any red node, if it has a parent , has a black parent.

3.2.1 *Path copying*

The old version of the tree has to be stored to support querying. One way to do this is store the entire tree for every time stamp. This requires $O(mn)$ space. (Where m is the number of time stamps and n is the number of nodes in the tree)

Another way to do this is store only the nodes which have changed over this time stamp. Any node that has a pointer to the changed node itself has to be copied. Assuming that every node has pointer to its children, copying a node requires the entire path to the node copied. (Since this is a binary search tree nodes are added only at the leaf) Refer to figure 3.2

The effect of this method is to create a set of search trees, one per update, having different roots but sharing common subtrees. Node color in a Red-Black tree is needed only for update operation. Since the tree for the plane sweep technique requires updates only in the present only the present color of the node needs to be stored.

The time and space per insertion or deletion in a persistent Red-Black tree is $log(n)$ since those operations change nodes only along a single path.

If the updates times are arbitrary, another auxiliary structure to facilitate access to appropriate root when searching in the past. An array of pointers to the roots sorted by time can be used for this structure. By doing a binary search the appropriate root can be got in $log(m)$ time. (Where m is the total number of updates)
This increases the query time from $log(n)$ to $log(m)$. (Where n is the total number of nodes) Since typically m is larger than n this increases the query time.

Drawbacks of *Path copying*

The major drawback of the path copying is its non-linear space usage. For every node newly introduced node the entire path has to copied. So the space usage is $O(logn)$ for every new node introduced. (Where n is the number of nodes already there in the tree at the time of introduction of the node).

3.2.2 *No node copying*

In a normal Red-black tree the amortized number of pointer changes is $O(1)$. Hence if the nodes are allowed to become arbitrarily fat then by making an extra pointer in a node each time a new node is added, we can reduce the cost of each update to $O(1)$. Similarly deletion of a node also would be $O(1)$.

Drawbacks of *No node copying*

There is a time penalty because of this approach. Since there can be any number of pointers in each node determining which pointer to follow is not a constant time operation. Choosing a correct pointer would take $O(logm)$ time. Hence the time for an access, insertion, or deletion is $((logn)(logm))$.

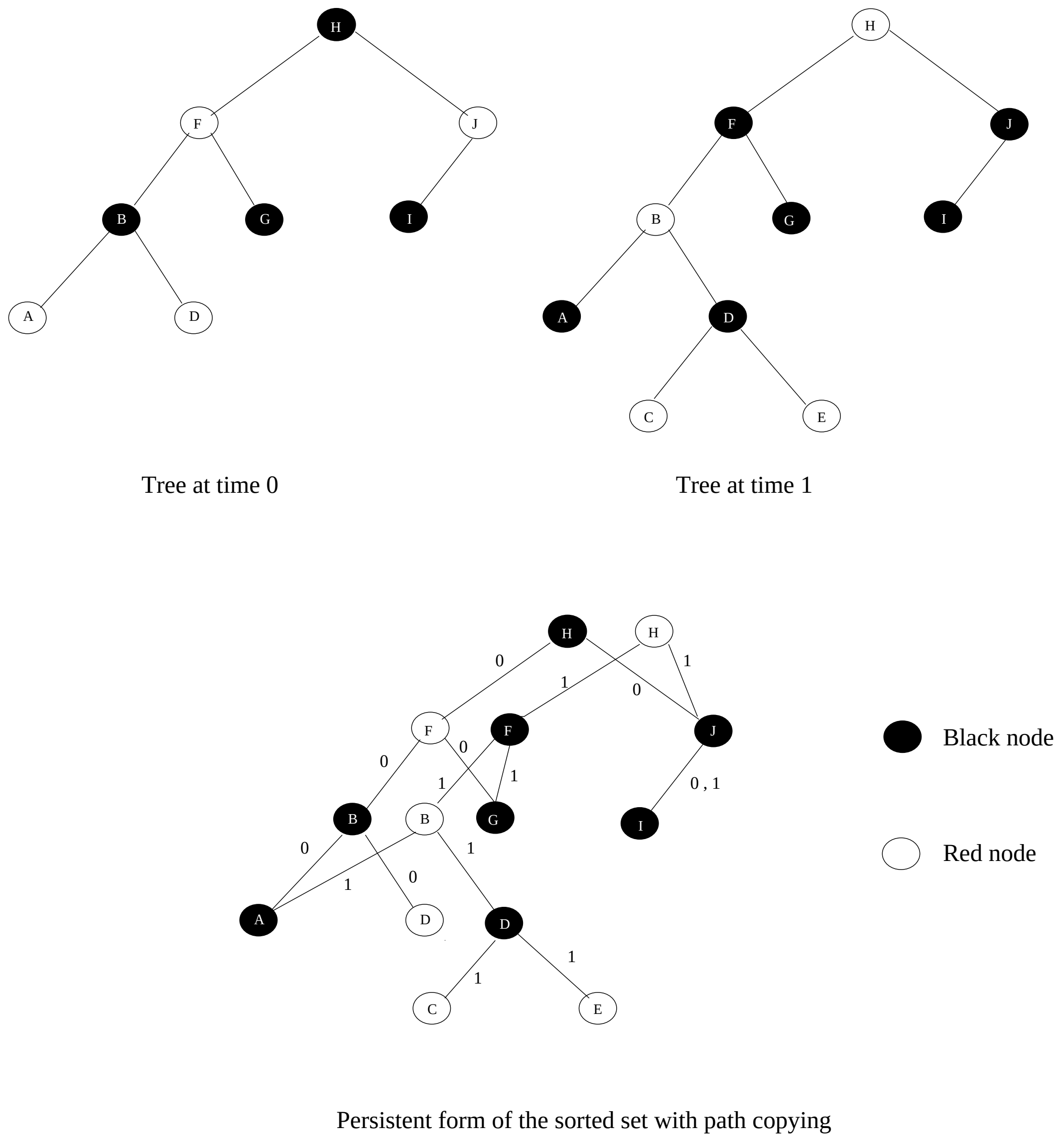


Figure 3.2: Persistent Red-Black tree with *Path Copying*

3.2.3 *Limited node copying*

The above time penalty can be eliminated by introducing limited node copying. Each node should be allowed to hold a maximum of k pointers where k is a small constant. When attempting to add a new pointer, if there is an empty slot the node is not copied. A pointer is made to the new node. But if there is no empty slot the node is copied. Because of there this increases the pointer requirement of its parent. So if the parent does not have an extra pointer it is also copied. So in the worst case the entire path is copied.

Searching doesn't have extra penalty line with *Limited node copying* technique. When searching for a node, first a decision has to be made whether to search a right path of left path. After this by examining the time stamps of extra pointers we can retrieve the required node. Since the number of extra pointers is some fixed small constant. The search time would be $O(logn)$ the extra time to search the pointers goes into the constant of the O notation.

If the update times are arbitrary then an extra array of pointers to the roots of the search tree has to be maintained. This makes the search time $O(logm)$, but the update times are $O(logn)$.

Definitions required for the Analysis of space requirement

The following definitions are needed to obtain the amortized space bound

1) *Live node* : All the nodes that are reachable from the latest root are called *Live nodes.*

2) *Dead node* : All the nodes which are not *Live nodes* are called *Dead nodes.* Live nodes can become dead but not vice versa.

3) *Potential* : The *Potential* of the data structure is

$$= No\ of\ live\ nodes\ -\ \frac{1}{k}(Number\ of\ free\ slots\ in\ live\ nodes).$$

4) *Amortized Space Cost of an update* : This is defined in terms of potential paradigm.

$$Amortized\ Space\ Cost\ of\ an\ update\ =\ Number\ Of\ Nodes\ it\ creates$$
$$+\ Net\ Increase\ in\ the\ potential\ it\ causes.$$

Analysis

1) *Insertion or deletion by storing a new pointer* :

Number of nodes it creates $= 0$.

Number of live nodes increased $= 1$.

Number of free slots it increases $= -1 \; + \; k$. (*Where '-1' is because of the reduction in the number of slots in the parent and 'k' because the free slots in the newly inserted node*).

Hence increase in potential $= 1 \; - \; \frac{1}{k}(-1 \; + \; k)$.

$$= \frac{1}{k}.$$

Hence Amortized space cost $= 0 \; + \; \frac{1}{k}$.

$$= \frac{1}{k}.$$

2) *Insertion or deletion by copying a node* :

Number of nodes it creates $= 1$.

Number of live nodes increased $= 1$.

Number of free slots increased $= -1 \; + k \; + \; k$. (*Where '-1' is because of the reduction in the number of slots in the parent of the new node created and 'k' is the increase in the slots of the newly created nodes*).

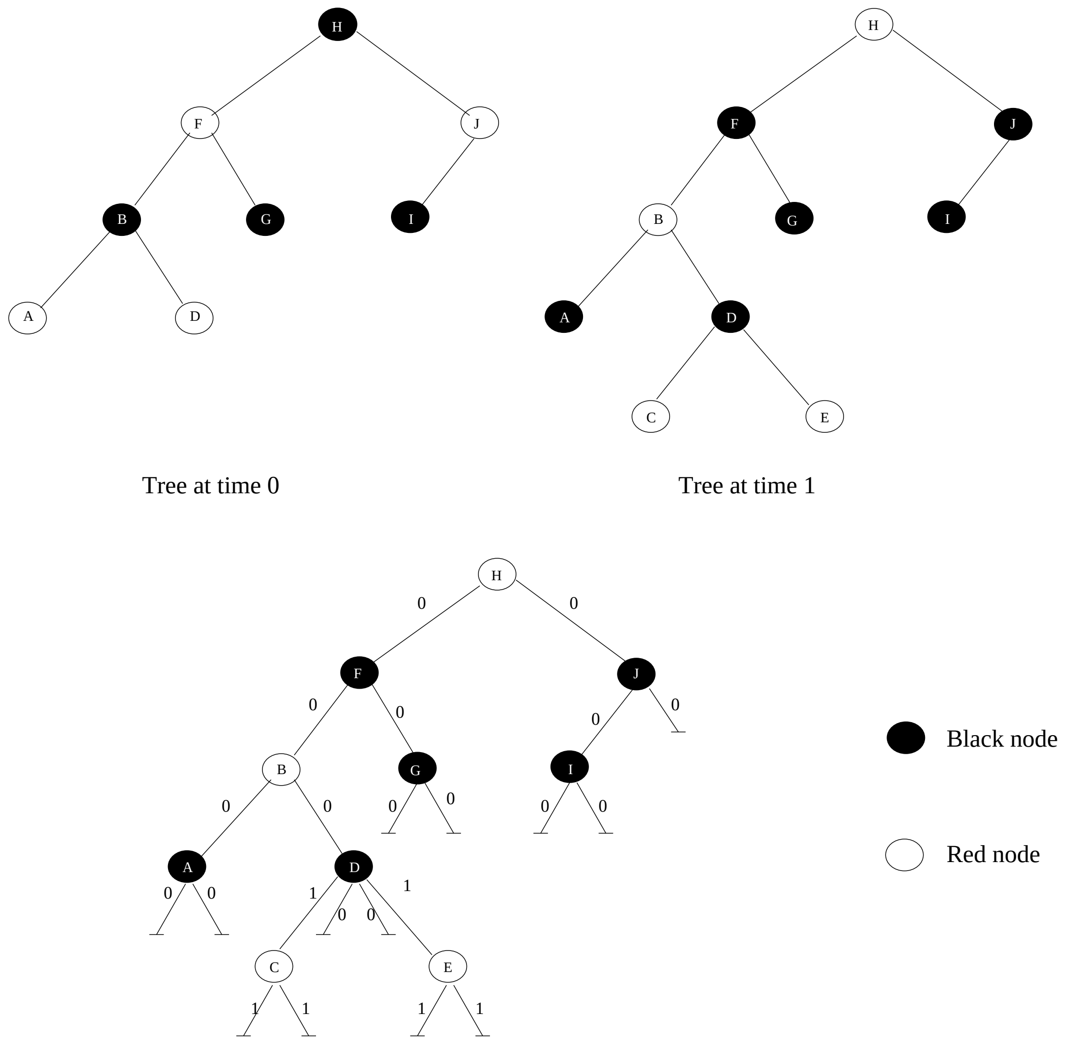
Increase in potential $= 1 \; - \; \frac{1}{k}(\; -1 \; + \; k \; + \; k)$.

$$= \frac{-1}{k} \; + \; 2.$$

Amortized space cost $= \; 1 \; + \; \frac{-1}{k} \; + \; 2$.

$$= 3 \; - \; \frac{1}{k}.$$

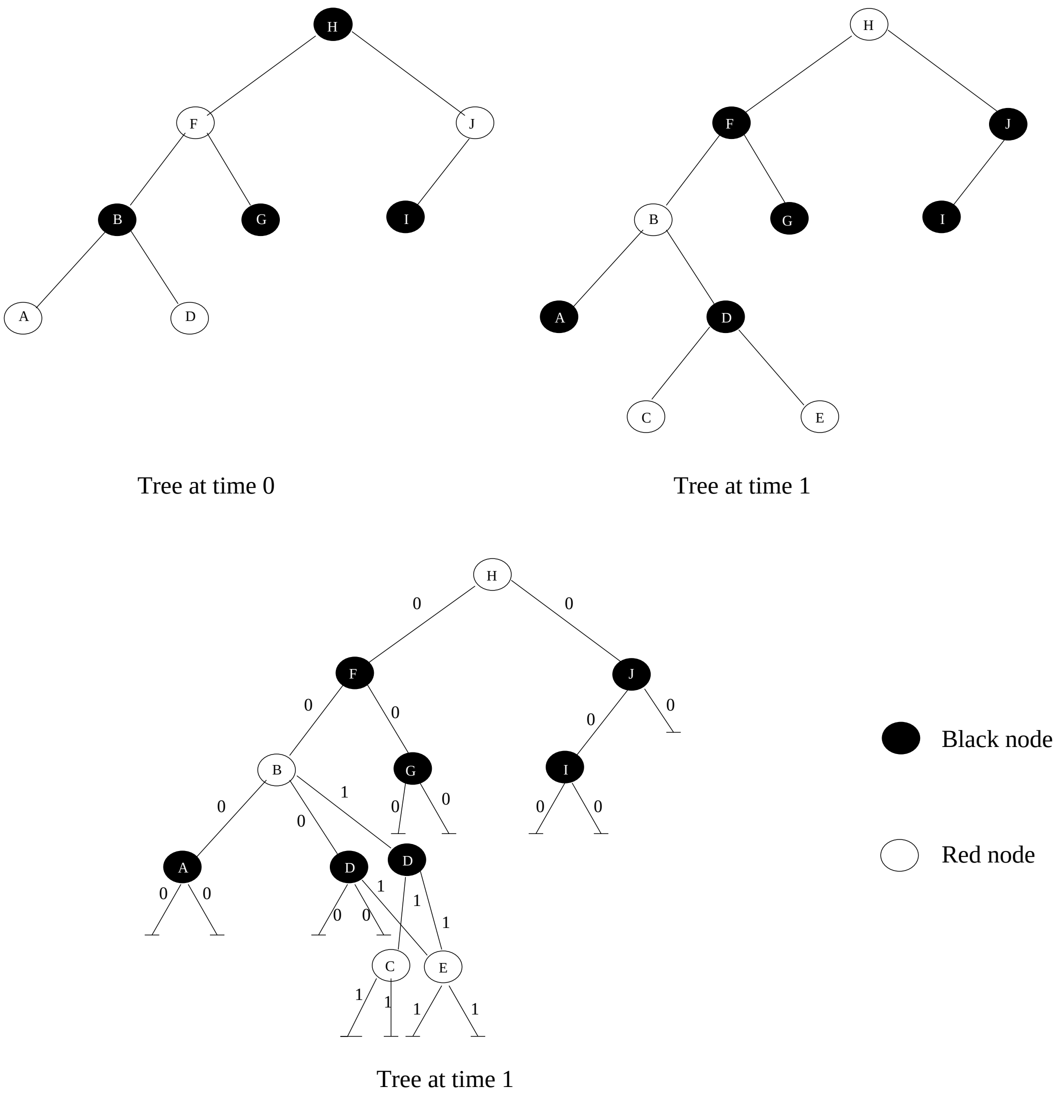
When $k = 1$, The amortized cost of any updation is $O(1)$.



Tree at time 1

Persistent form of the sorted set with no node copying

Figure 3.3: Persistent Red-Black tree with *No Node Copying*



Persistent form of the sorted set with limited node copying

Figure 3.4: Persistent Red-Black tree with *Limited node Copying*

Chapter 4

Finger Trees

4.1 Jordan Sorting

Consider the following problem from computational geometry :

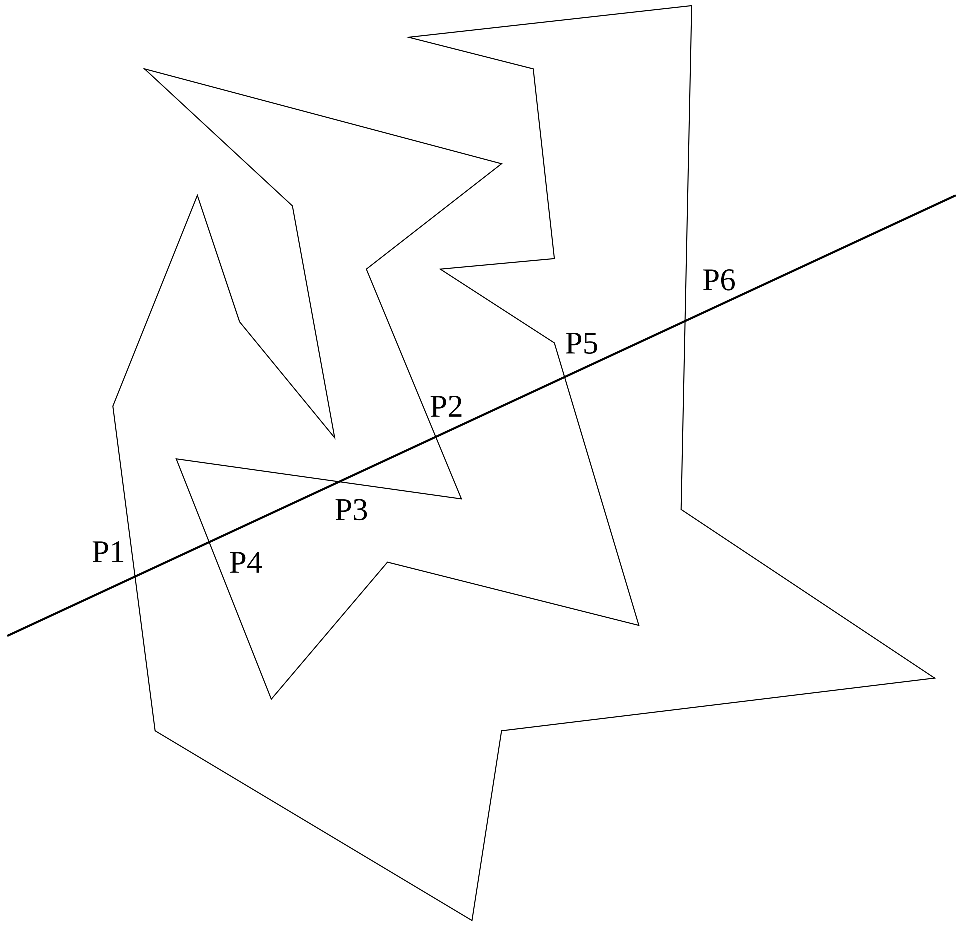
Given a simple polygon and a line intersecting the simple polygon, compute the intersection points sorted in the order they appear on the line. See figure 4.1.

The intersection points $P1$, $P2$, ... are named as they appear along the polygon. A simple algorithm to solve the problem is : Consider each edge of the polygon and compute its intersection point with the line (if one exists). Transform the points in such a way that they lie on the x-axis. Then sort them according to their x-coordinates. Finally, perform the reverse transformation. This method takes $O(nlog\ n)$ time. As we shall see shortly, the problem can be solved in $O(n)$ time.

The problem stated above is an instance of the general problem known as *Jordan Sorting.* Another interesting related problem is : Given two curves A and B, and the their points of intersection along the curve A, find the points of intersection along B.

The *Jordan Sorting* problem is stated as follows :

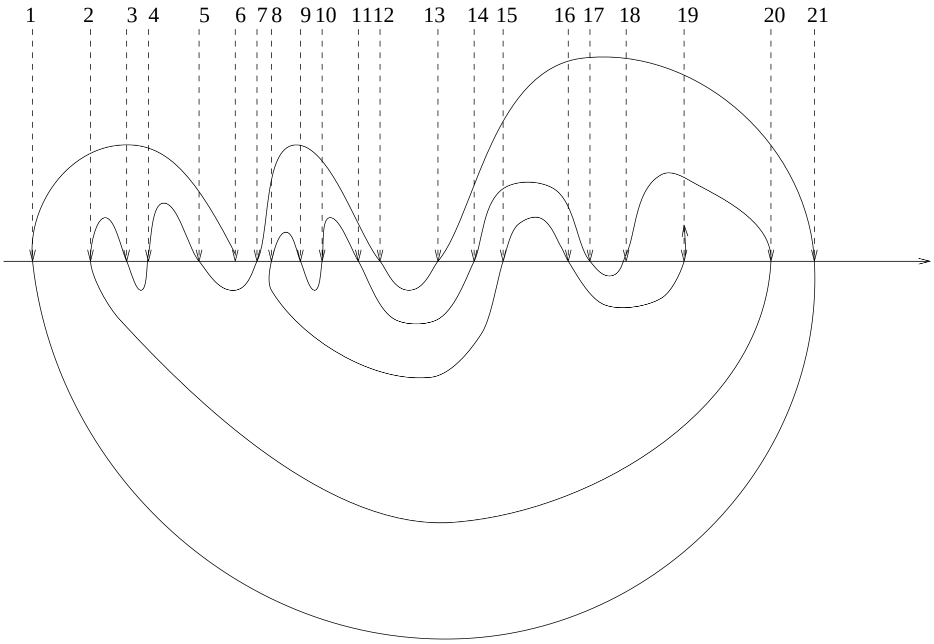
Let C be a Jordan Curve (a non-intersecting curve) that lies in the xy plane. The x axis is not a tangent to this curve at any point on the curve. Let x_1, x_2, ... x_n be the abscissas of the intersection points of C with the x axis, listed in the order they appear along C. The sequence x_1, x_2, ... x_n is called a Jordan Sequence. *Sort the Jordan Sequence.*



The sorted output should be : P1 P4 P3 P2 P5 P6

Figure 4.1: Intersection of a simple polygon and a line

The Jordan Sorting problem is illustrated in figure 4.2. The Jordan Sequence is called *Jordan Permutation* if the sequence $x_1,\ x_2,\ \ldots\ x_n$ is a permutation of integers 1 to n.



Jordan Sequence

X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15	X16	X17	X18	X19	X20	X21
6	1	21	13	12	7	5	4	3	2	20	18	17	14	11	10	9	8	15	16	19

Figure 4.2: A Jordan Curve

The Jordan Sequence can be represented with help of two trees. One tree describes the upper part of the curve and the other tree describes the lower part of the curve. The trees are called *upper tree* and *lower tree* respectively. Consider the upper part of the curve. It consists of a set of arcs. Each arc is identified by its two end-points. We denote an arc as $\{x_j,\ x_{j-1}\}$. Hence for all arcs in upper part of the curve j is *odd* and for all arcs in the lower part of the curve j is *even* (whether j is odd or even for upper or lower part of the curve depends on the way the curve is traversed). The upper part of the curve shown in figure 4.2 consist of the arcs $\{x_1,\ x_2\}$, $\{x_3,\ x_4\}$, $\ldots$ $\{x_{19},\ x_{20}\}$. Similarly the arcs belonging to the lower part of the curve are $\{x_2,\ x_3\}$, $\{x_4,\ x_5\}$, $\ldots$ $\{x_{20},\ x_{21}\}$.

The trees are built using the following rules : Each arc is represented by a node in the tree. An arc that *closes* another arc is an ancestor of the arc in the tree.

An imaginary arc $\{-\infty, \infty\}$ is assumed exist and is placed at the root of the tree. The upper and lower trees for the Jordan Curve in figure 4.1 is shown in figure 4.3.

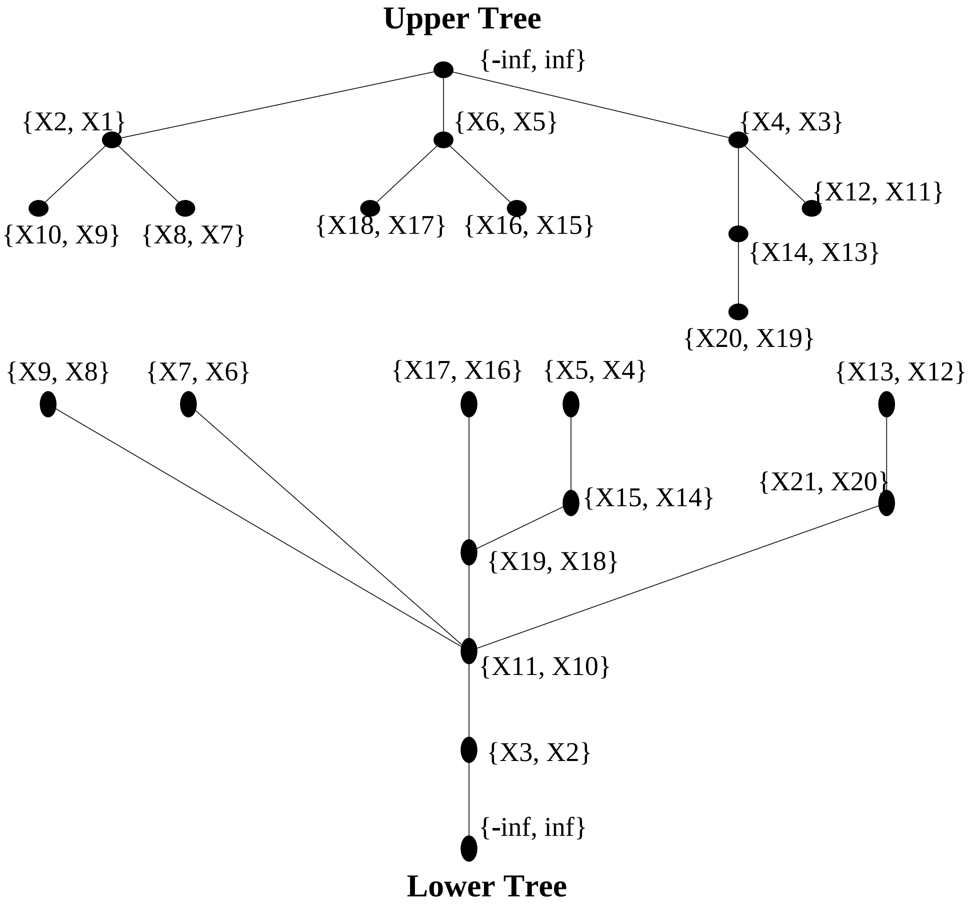


Figure 4.3: Upper and Lower tree for the Jordan Curve in figure 4.2

4.2 Algorithm and Data Structures

Three data structures are maintained by the algorithm : The upper tree, the lower tree and the ordered list of points processed so far.

The algorithm processes each point x_i by adding the interval $\{x_i,\ x_{i-1}\}$ $(i \neq 1)$ in to the appropriate tree. For the Jordan Curve in figure 4.2, $\{x_i,\ x_{i-1}\}$ is inserted in the upper tree if i is even, otherwise it is inserted into the lower tree. The placement of the interval in the tree provides a location of the point x_i into the ordered list and hence can be carried out in $O(1)$ time.

The algorithm proceeds as follows :

1. The ordered list initially contains $(-\infty,\ \infty)$. The upper and lower tree each contains the interval $\{-\infty,\ \infty\}$.

2. If $i = 1$, then just insert the point x_1 in the sorted list between $-\infty$ and ∞.

3. If $i \neq 1$, find the two neighbors r and s of x_{i-1} such that $r < x_{i-1}$ and $s > x_{i-1}$. Let $\{x_{j-1},\ x_j\}$ and $\{x_{k-1},\ x_k\}$ be the intervals containing r and s.

4. If both $\{x_{j-1},\ x_j\}$ and $\{x_{k-1},\ x_k\}$ contain x_i then : $\{x_{j-1},\ x_j\} = \{x_{j-1},\ x_j\} = \{r, s\}$ If this condition is not met then the given sequence is *not* a Jordan Sequence and the algorithm aborts. We insert $\{x_{i-1},\ x_i\}$ as the child of $\{r,\ s\}$. Also x_i is inserted into the ordered list immediately before or after x_{i-1} as appropriate.

5. If one of the pairs $\{x_{j-1},\ x_j\}$ does *not* contain x_i, we move to the parent of $\{x_{j-1},\ x_j\}$. Let's call it P. We partition the children of P in two groups (this operation is called *list splitting*. One group P_1 contains the children that are contained by $\{x_{i-1},\ x_i\}$ and another P_2 contains the rest. We remove P_1 from P. Attach the node containing the $\{x_{i-1},\ x_i\}$ as a child of P. The nodes in P_1 are attached as children of $\{x_{i-1},\ x_i\}$. If while splitting the children of P in P_1 and P_2, a child is found such that one of the end-points of the interval is inside $\{x_{i-1},\ x_i\}$ and the other end-point is outside, then the sequence is *not* a Jordan Sequence and the algorithm is aborted. When splitting the list of children of P, the split occurs at two points as shown in figure 4.4. The splitting shown in the figure takes place when $x_{i-1} < x_i$. The other case, $x_{i-1} > x_i$ is similar. x_i is inserted in the ordered list *after* u_{q-1} (x_i fits *after* x_1 if i is odd and x_1 lies between u_{q-1} and x_i.)

4.2.1 Space and Time Requirements

The input list $x_1, x_2, \ldots x_n$ is stored in a one dimensional array of size n. Given i, we can access x_i in $O(n)$ time. The space needed is $O(n)$.

The ordered list is represented as a doubly-linked list. Since all operations on the ordered list are of the type *access/insert and element before/after a given* *element in the list*, it can be carried out in $O(1)$ time. The space needed is $O(n)$.

The upper and lower trees are *annotated m-ary trees*. Each node contains a sorted list containing the end-points of its own interval as well as the end-points of the intervals stored in its children. See figure 4.5. The annotation list is shown in bold font associated with each node. Each point in the annotated list points to its location in the ordered list. Hence given a location in the annotated list of a node, we can find the point in the ordered list $O(1)$ time. Since each point contributes to two arcs, one on the upper side and one on the lower side, each point x_i occurs exactly four times. Twice in each tree (upper and lower), once in a parent node and once in the child node. Hence the storage of the points in the upper and lower trees needs $O(n)$ space.

To estimate the running time of the algorithm, we need to know the complexity of the *list splitting* operation. The operation is described as :

Given a pair of numbers a and b and a number u_p from an ordered list u_1, u_2, $\ldots$ u_m such that $u_{p-1} < a < u_p$, find a number u_q such that $u_{q-1} < b < u_q$.

If the annotation list is represented as a *circular doubly connected linked list*, the complexity of the above operation is $O(min\{|p-q|,\ m-|p-q|\})$. The overall complexity of the algorithm then turns out to be $O(nlog\ n)$. This is no better than any other fast sorting algorithm like quick sort.

We can improve the running time of the algorithm by changing the implementation of the annotation list.

4.3 Finger Trees

Ordinary binary search tree allow search starting at the root node. The searching time is $O(log\ n)$ where n is the total number of elements in the tree. *Finger Trees* allow the search to start from multiple locations (called *fingers*, hence the name *finger tree*) in the tree. The searching time is $O(log\ d)$ where d is the distance of the query point from the nearest finger. Many different versions of the finger trees exist. We will be studying one such version[1] which is based on *2,4 trees*.

A 2,4 tree is a tree in which all the external nodes have the same height and every interior node has either two, three or four children.

A finger tree is a 2,4 tree in which each external node stores an item ordered list in left-to-right order. Each internal node stores the items with maximum key

[1] called *finger tree* henceforth, though other versions exist

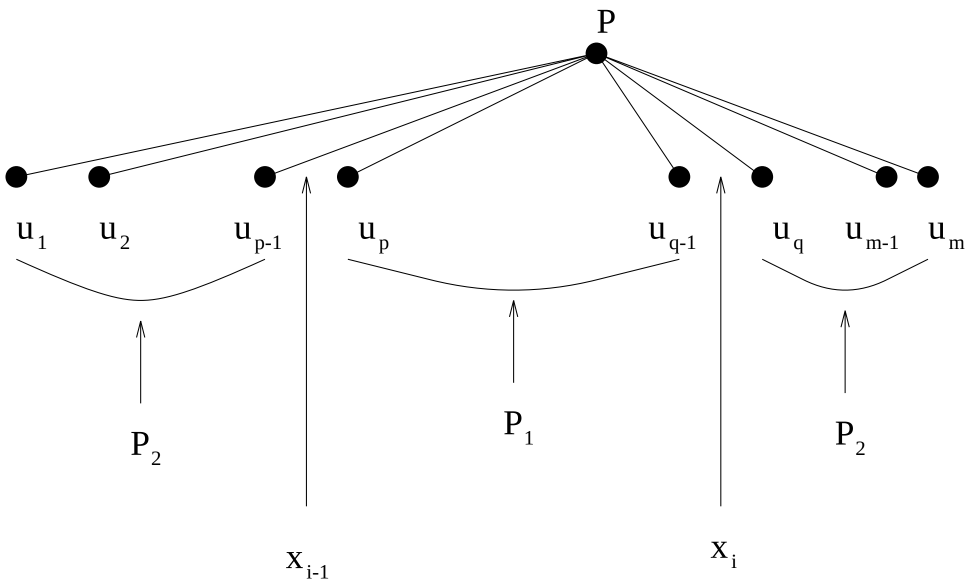


Figure 4.4: Splitting the list of children

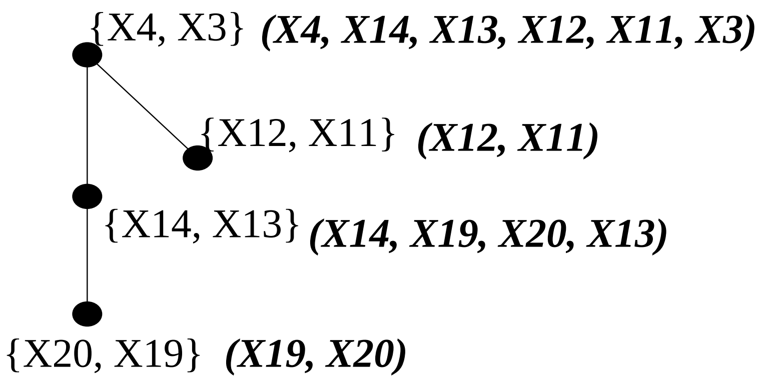


Figure 4.5: Part of the annotated upper tree for the Jordan Curve in figure 4.2

value (called maxvalue*) occurring in each of its its subtrees except the right most subtree. Hence an internal node having d children stores $d-1$ maxvalues inside it, one corresponding to each of its subtrees except the rightmost one. The root node is an exception : it stores maxvalues from all its subtrees. Stored along with each maxvalue is a direct link to the external node in which it is stored. In addition, each node contains links to its left and right neighbor at the same height called* level links*. The level links are circular. The internal nodes also contain links to the parent node as well as to all its children. See figure 4.6.*

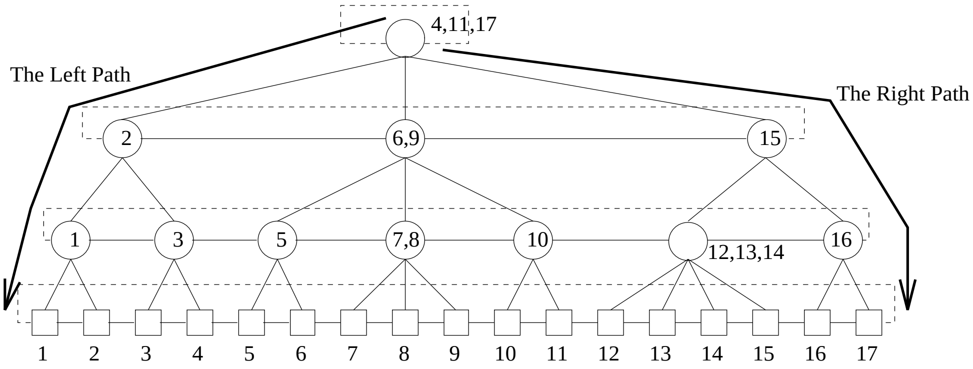


Figure 4.6: A Finger Tree

The *left path* on the tree is defined as the path from the root node which always takes the leftmost child while traversing down the tree.

The *right path* on the tree is defined as the path from the root node which always takes the rightmost child while traversing down the tree.

4.3.1 Searching an item inside a finger tree

Let's consider the following problem : Given an item x in the finger tree and a query point y, find an item z such that z is the smallest item in the tree greater than or equal to y.

1. If $x < y$, travel up the tree from the external node containing x. We stop once we reach :
   - a node e on the right path of the tree. *or*
   - a node e such that the right neighbor of e contains a maxvalue that is greater than or equal to y. *or*

- a node e on the left path of the tree whose left neighbor (who is on the right path because of the circular level links) contains a key less than or equal to y.

2. If e is on the right path of the tree, we travel down the tree, each time choosing that subtree whose maxvalue is greater than or equal to y and its left neighboring subtree's maxvalue is less than y. The rightmost subtree is chosen when all the maxvalues in the node are less than y.

3. If e is not on the left path then we have to search down the right neighbor of e. We need to search down the rightmost child of e if y is greater than *all* the maxvalues in e.

4. Else we search down the left neighbor of e.

5. The case when $y < x$ is treated in symmetric manner. We travel up the tree till we reach the left path of the tree *or* reach a node whose left neighbor contains a maxvalue that is less than or equal to y *or* a node on the right path whose right neighbor contains a maxvalue that is greater than or equal to y.

6. The search proceeds as follows : If e is on the left path of the tree, we travel down the tree from there. If e is not on the right path of the tree, we search the left neighbor's subtree. Else we search down the right neighbor's subtree.

The searching process is illustrated in figures 4.7 and 4.8.

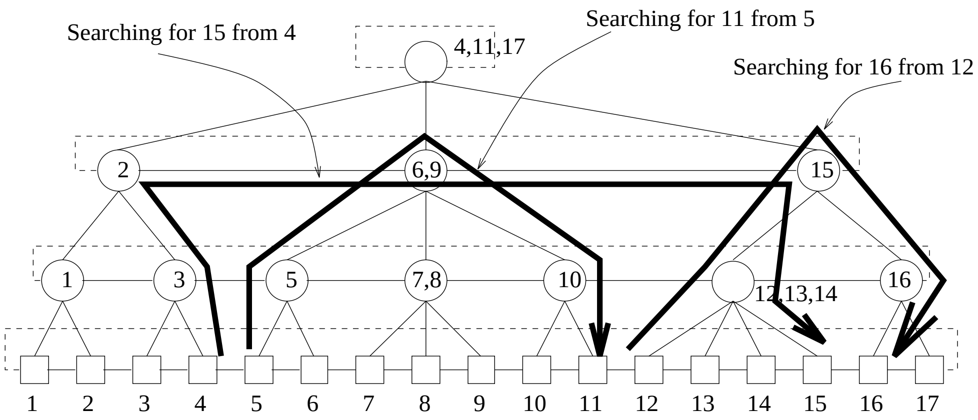


Figure 4.7: Searching in a Finger Tree ($x < y$)

Time Complexity

The time required to search is proportional to the maximum height reached during the search. It can be proved that if n is the total number of items in the tree and d is the difference between the ranks of x and y (the horizontal distance between the external node containing x and z) then $h = O(log\ min\{d,\ n-d\})$.

4.3.2 Inserting and deleting an item inside a finger tree

To insert an item y into the tree next to an item x :

1. Create an external node to hold the item y and attach it to the parent of x (let's call it p), next to x. Also add y inside the list of children maintained in the parent of x. See figure 4.9.
2. If the insertion does not violate the properties of 2, 4 tree, that is it does not result in p having five children, then the insertion process is complete.
3. Otherwise split p in two nodes, p_1 and p_2, with one node having two children and the other having three children. The parent of p is parent of both p_1 and p_2. This process is called *splitting*. See figure 4.10.
4. If this action results in another violation (generation of a node having five children), we repeat the above step by *splitting* the offending node.
5. This process will either terminate when no more violation of 2, 4 tree properties occurs or we reach the root node. If we are at the root node, we split the root node in two nodes, r_1 and r_2 and create a new root node, r. We attach r_1 and r_2 to r as its two children. See figure 4.11.

To delete an item y from the tree :

1. Destroy the external node containing y. Remove y from the list of children of y's parent, p. See figure 4.12.
2. If the deletion does not violate the properties of 2, 4 tree, that is it does not result in p having one child, then the deletion process is complete.
3. Otherwise look for a sibling of p having three or four children. If such a sibling exists, remove a child from that sibling and attach it as a child of p. This process is called *borrowing*. See figure 4.13. The deletion is now complete.
4. If the sibling(s) of $p$have exactly two children, then merge a sibling with two children with p. This process is called *fusing*. See figure 4.14. This may again result in the parent of p violating 2, 4 trees properties. Repeat the above step to correct the situation.

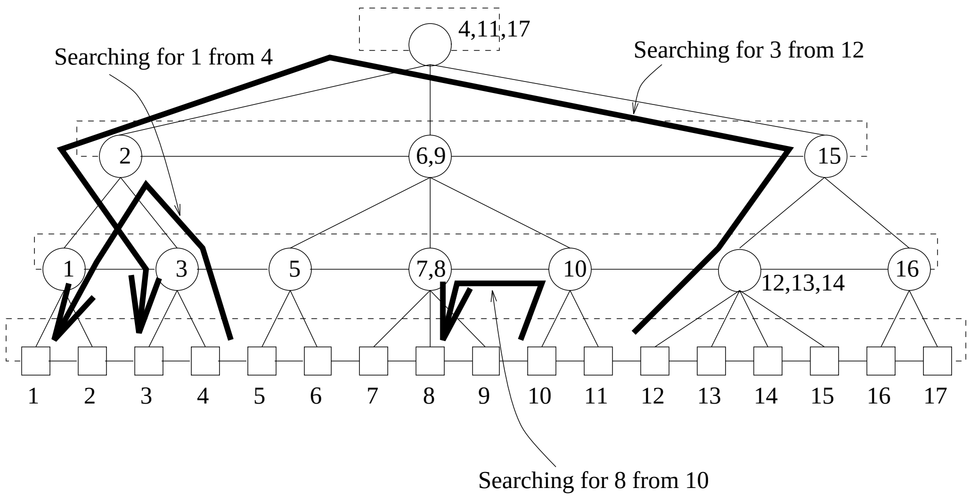


Figure 4.8: Searching in a Finger Tree ($x > y$)

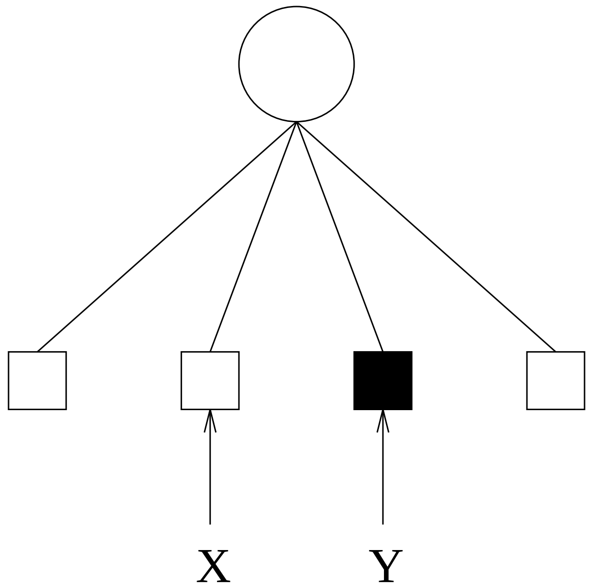


Figure 4.9: Inserting an item in a finger tree

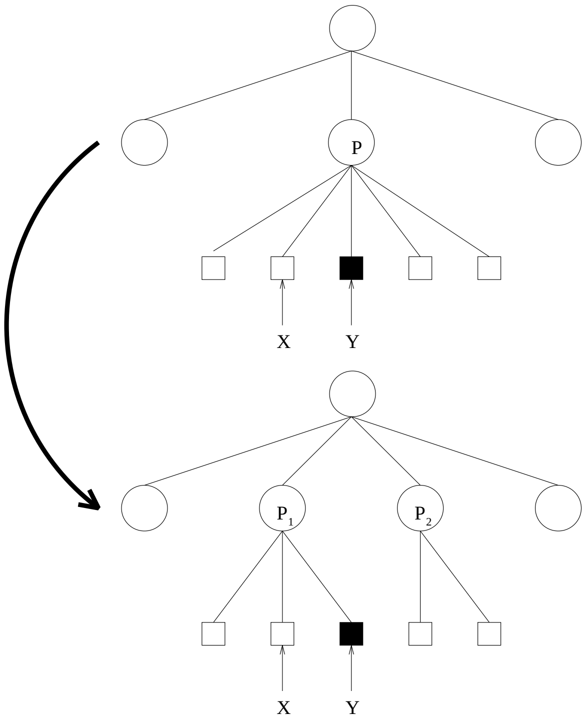


Figure 4.10: Node splitting in a finger tree

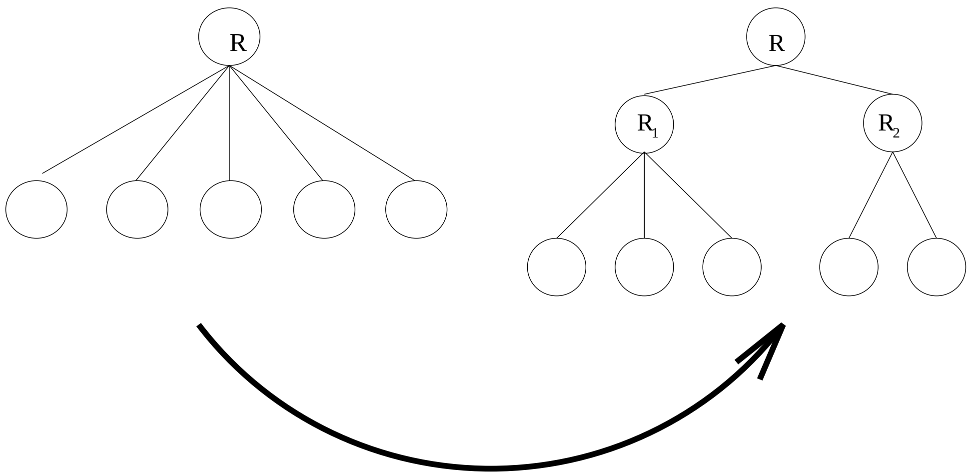


Figure 4.11: Root splitting in a finger tree

5. The fusing process stops when you reach a stage where violation of 2, 4 trees properties does not exist *or* the root node is reached and the root node has one child. If the root node has one child then delete the root node and make the child of the root node the new root node.

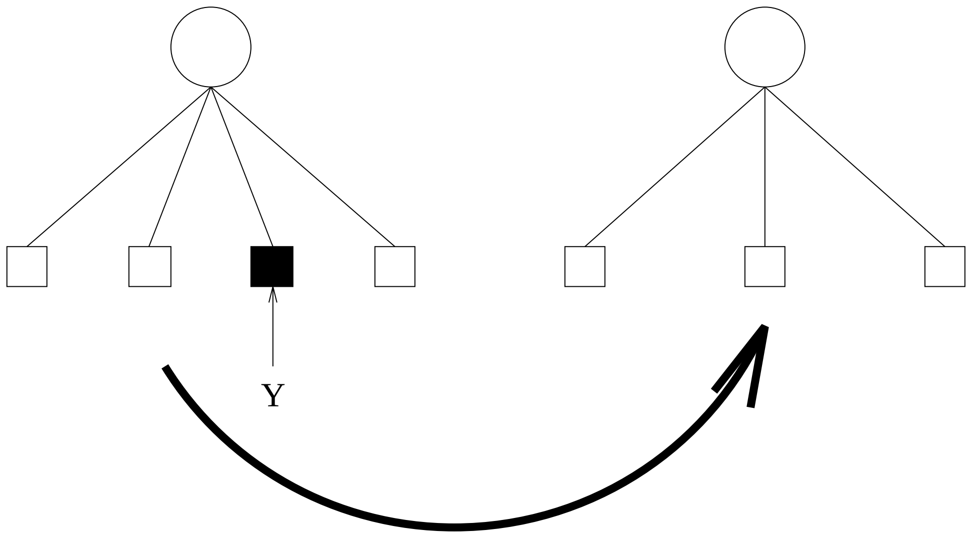


Figure 4.12: Deleting a node in a finger tree

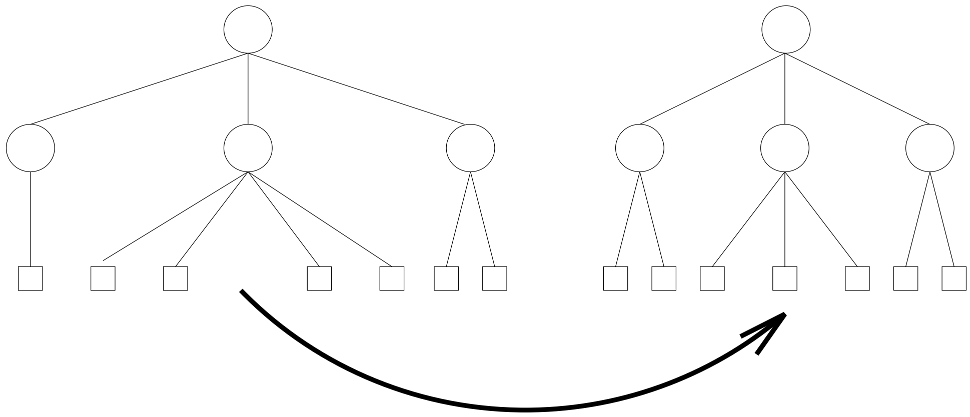


Figure 4.13: Borrowing in a finger tree

Time Complexity

The amortized time bound for the insertion and deletion operation in finger trees is $O(1)$.

4.3.3 List Manipulation inside a Finger Tree

Concatenation

Given two finger trees T_1 and T_2 such that all items in T_1 are less than all items in T_2, we have to create a single tree T such that it represents concatenation of the ordered lists in T_1 and T_2. Note that time taken to perform the same operation on ordered lists in indexed arrays takes $O(n)$ time.

1. Find out the *largest* item in T_1 (called t_{1L}) and the *smallest* item in T_2 (called t_{2S}). Note that this step can be carried out in constant time.

2. t_{1L} is on the right path of T_1 and t_{2S} is on the left path of T_2. Walk up the right path of T_1 and left path of T_2, till we reach the root node of either of the two trees. See figure 4.15.

3. Let the height of the root found be h. Let height of T_1 be h_1 and height of T_2 be h_2. Then, $h = min\{h_1, h_2\}$.

4. If $h = h_1 = h_2$, then we create a new root node R and attach root R_1 of T_1 and root R_2 of T_2 as its two children. See figure 4.16. This completes the concatenation process.

5. If $h_1 < h_2$ then we are at the root R_1 of T_1 and some other non-root node N_2 on the left path of T_2. R_1 is attached to the node N_2 as the leftmost child. This might result in N_2 having five children, thus violating the 2, 4 tree property. *Splitting* (see section 4.3.2 and figure 4.10) is used to correct the situation. See figure 4.17.

6. If $h_1 > h_2$ then we are at the root R_2 of T_2 and some other non-root node N_1 on the right path of T_1. R_2 is attached to the node N_1 as the rightmost child. This might result in N_1 having five children, thus violating the 2, 4 tree property. *Splitting* (see section 4.3.2 and figure 4.10) is used to correct the situation. See figure 4.17.

Splitting

Next we describe the list splitting operation on the finger tree : Given a 2, 4 tree T and an item x in the list, create two trees T_1 and T_2 such that all items less than x are in T_1 and all items greater than x are in T_2. The items equal to x might go to either list depending on the specific implementation.

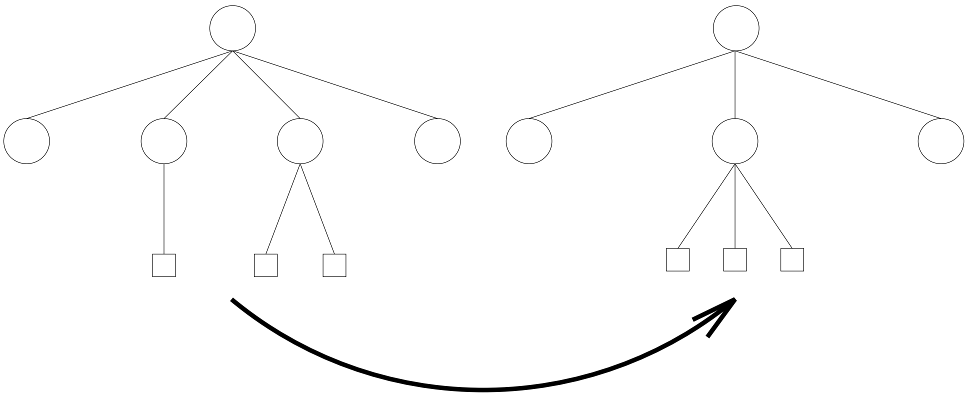


Figure 4.14: Fusing in a finger tree

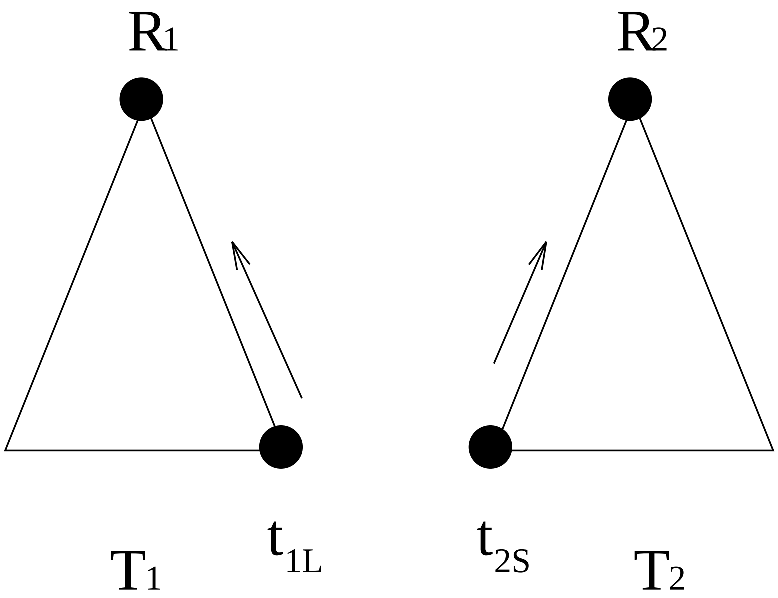


Figure 4.15: Walking up a Finger Tree during Concatenation

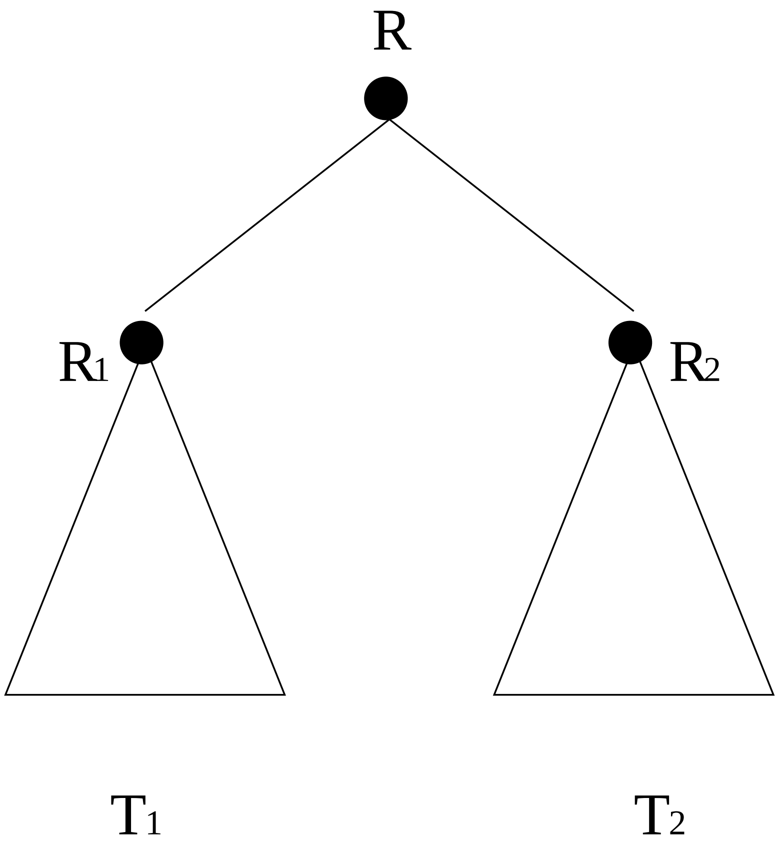


Figure 4.16: Finger Tree Concatenation : $h_1 = h_2$

1. Walk up the tree starting at the node x.
2. At each internal node N, split the the node in N_1 and N_2 such that the subtrees containing items less than x are children of N_1. The rest go to the node N_2.
3. The root is split in two nodes, R_1 and R_2. R_1 is becomes root of T_1 and R_2 becomes root of T_2. See figure 4.18.
4. The above process might result in some 1 nodes[2] along the right path of T_1 and left path of T_2. We traverse down (top down) the left and right path of T_2 and T_1 respectively looking for 1 nodes.
5. At each 1 node found, we apply fusing (see section 4.3.2 and figure 4.14). The process is called *repeated top down fusing.*

Time Complexity

The amortized time bound for the concatenation and splitting operation in finger trees is $O(h)$, where h is height of T.

4.4 Splitting the Family Lists in the Jordan Sorting Problem

Now let's see how we can use finger trees to represent family trees. Recall that, in the algorithm for Jordan Sorting problem (see section 4.1, we maintain two trees, the upper tree and the lower tree. Each node represents a single interval. Each node stores the intervals of its children as satellite data. The intervals are stored in ordered list in the node. This list is called a *Family List.* We will implement the ordered list as a finger tree (see section 4.3).
The operation we want to perform on the family list is call *List Splitting* (see section 4.2.1). Given below is an algorithm for list splitting :

1. We are given a finger tree T and two items x and y ($x \leq y$) in it. We want to split T in T_1 and T_2 such that all the items in T that are greater than x and less than y are in T_1 and the remaining items are in T_2. See figures 4.4 and 4.19.
2. Walk up the tree T from the external nodes containing x and y simultaneously. We stop when we either reach a common node e or two neighboring nodes a and b. Note that either x and y are in the subtree rooted at e or x is in a subtree rooted at a and y is in a subtree rooted at b. There are three case possible (see figure 4.20:

[2] A node having only one child.

- *Wrap around* : b is a left neighbor of a.
- *Single Root* : e exists.
- *Double Root* : b is a right neighbor of a.

3. When we have a wrap around, detach the subtree rooted at a (called T^L). Also detach the subtree rooted at b (called T^R). Call the remaining portion (of T) T^P. Use fusing (see section 4.3.2) on parents of a and b if they become a 1 node.

   - Split T^L around x getting T_1^L containing items less than or equal to x and T_2^L containing items greater than x.
   - Split T^R around y getting T_1^R containing items less than or equal to y and T_2^R containing items greater than y.
   - Concatenate T_1^L with T_2^R to get T_2.
   - Concatenate T_2^L, T^P and T_1^R to get T_1.

   See figure 4.21.

4. If we have a single root :

   - Detach the tree rooted at e (called T^M). Call the remaining tree T^P. Let the height T^M be h.
   - Split T^M in T_1^M and T_2^M such that items less than or equal to x are in T_1^M and rest are in T_2^M.
   - Split T_2^M in T_3^M and T_4^M such that items less than y are in T_3^M and rest are in T_4^M.
   - Concatenate T_1^M and T_4^M (called T_{14}^M.
     - If the height of T_{14}^M is less than or equal to h then extend the tree by adding a node above the root (it becomes a 1 node). Then attach T_{14}^M in the original place of e. Eliminate 1 nodes (if any) by repeated top down fusing (see section 4.3.3.
     - If the height of T_{14}^M is greater than h, then delete the root of T_{14}^M and attach its children to the parent of e (which is now in T^P.

     The resulting tree is T_2.
   - The T_3^M tree is T_1.

   See figure 4.22.

5. If we have a double root :

   - Detach the subtree rooted at a (called T^L and height h_1). Also detach the subtree rooted at b (called T^R and height h_2). Call the remaining portion (of T) T^P. Use fusing (see section 4.3.2) on parents of a and b if they become a 1 node.

- Split T^L around x getting T_1^L containing items less than or equal to x and T_2^L containing items greater than x.
- Split T^R around y getting T_1^R containing items less than or equal to y and T_2^R containing items greater than y.
- To create T_2 :
  - If the height of T_1^L is less than or equal to h_1 then add a new root node to T_1^L. Attach it to the parent of a (in T^P). Eliminate any 1 node(s) formed by repeated fusing.
  - If the height of T_1^L is more than h_1 then delete the root of T_1^L. Attach the children of the deleted root to the parent of a (in T^P). Eliminate any 5 node formed by repeated splitting.
  - If the height of T_2^R is less than or equal to h_2 then add a new root node to T_2^R. Attach it to the parent of b (in T^P). Eliminate any 1 node(s) formed by repeated fusing.
  - If the height of T_2^R is more than h_2 then delete the root of T_2^R. Attach the children of the deleted root to the parent of b (in T^P). Eliminate any 5 node formed by repeated splitting.

  The modified T^P tree is T_2.
- Concatenate T_2^L, T^P and T_1^R to get T_1.

See figure 4.23.

Time Complexity

The amortized time bound for the *family list splitting* operation in finger trees is $O(h)$, where h is height of T.

4.5 Complexity Analysis

Using finger trees, we can implement the family list splitting operation (see section 4.2.1) in $O(min\{log(d), log(n-d)\})$ time[3]. This is an improvement over the $O(n)$ time required if we implement the family lists using double linked lists. It enables us to do Jordan Sorting in $O(n)$ time.

[3] $h = O(min\{log(d), log(n-d)\})$

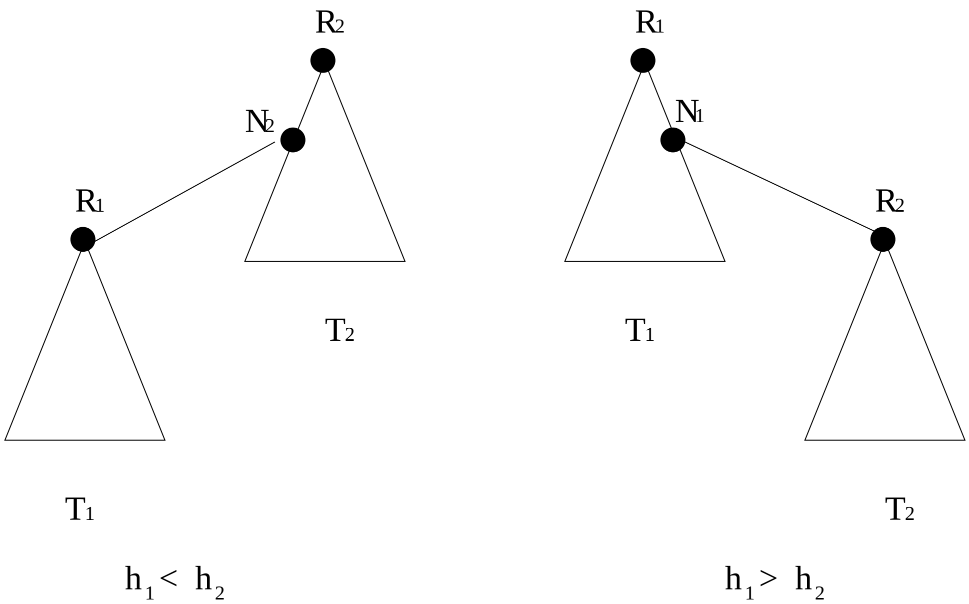


Figure 4.17: Finger Tree Concatenation : $h_1 < h_2$ and $h_1 > h_2$

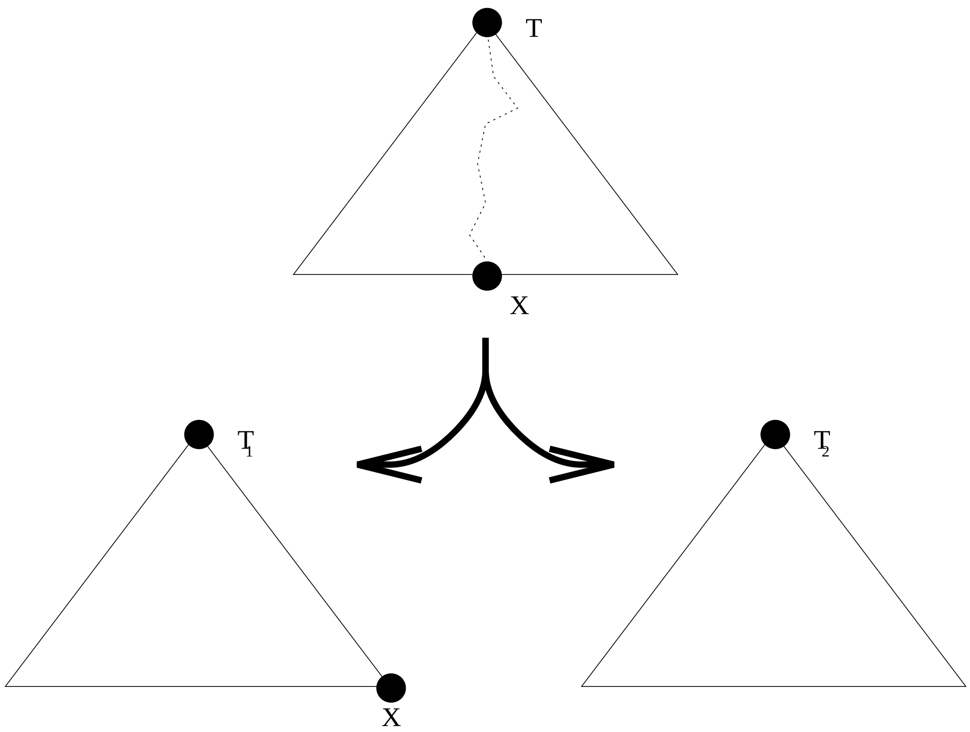


Figure 4.18: Finger Tree Splitting

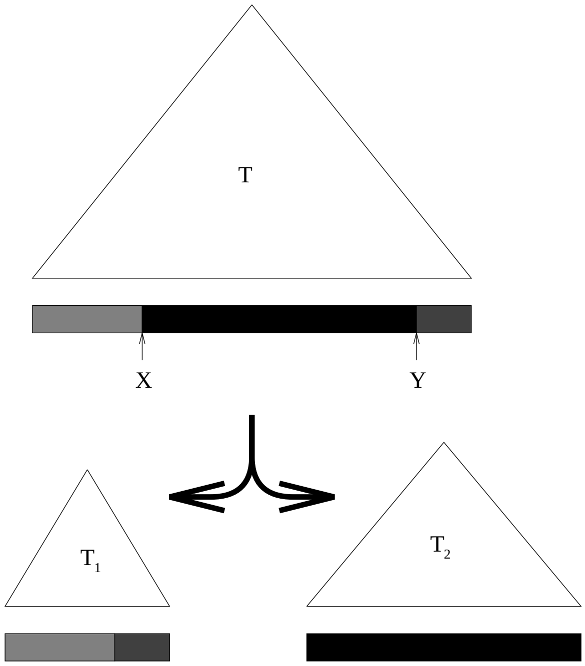


Figure 4.19: Family List Splitting

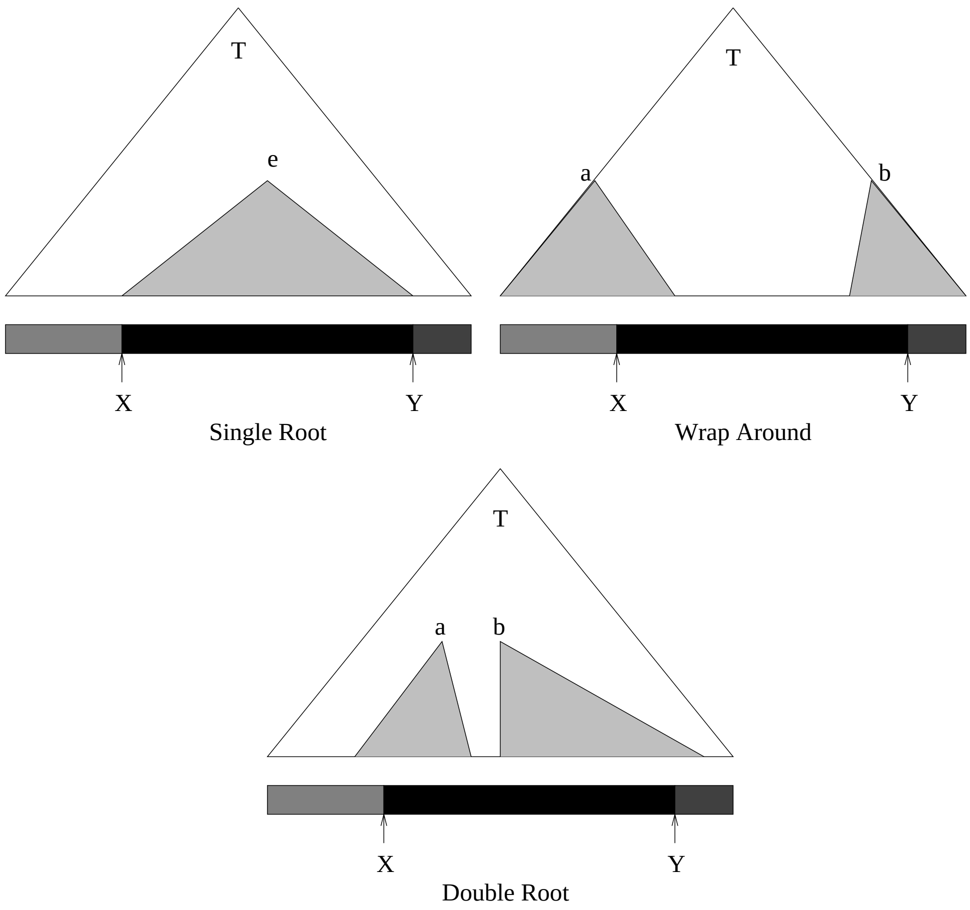


Figure 4.20: Family List Splitting : Three different cases

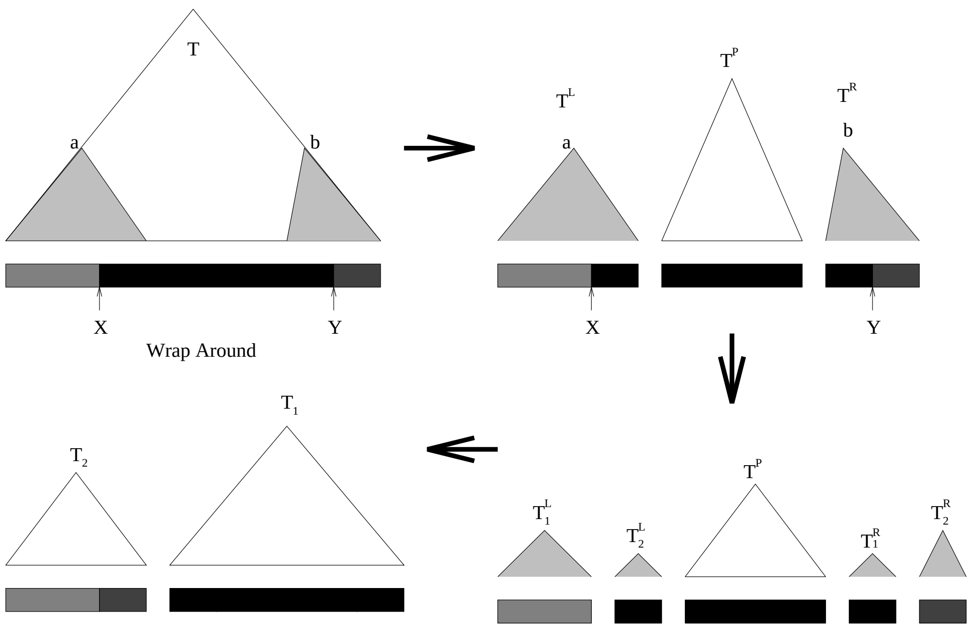


Figure 4.21: Family List Splitting : *Wrap Around* Case

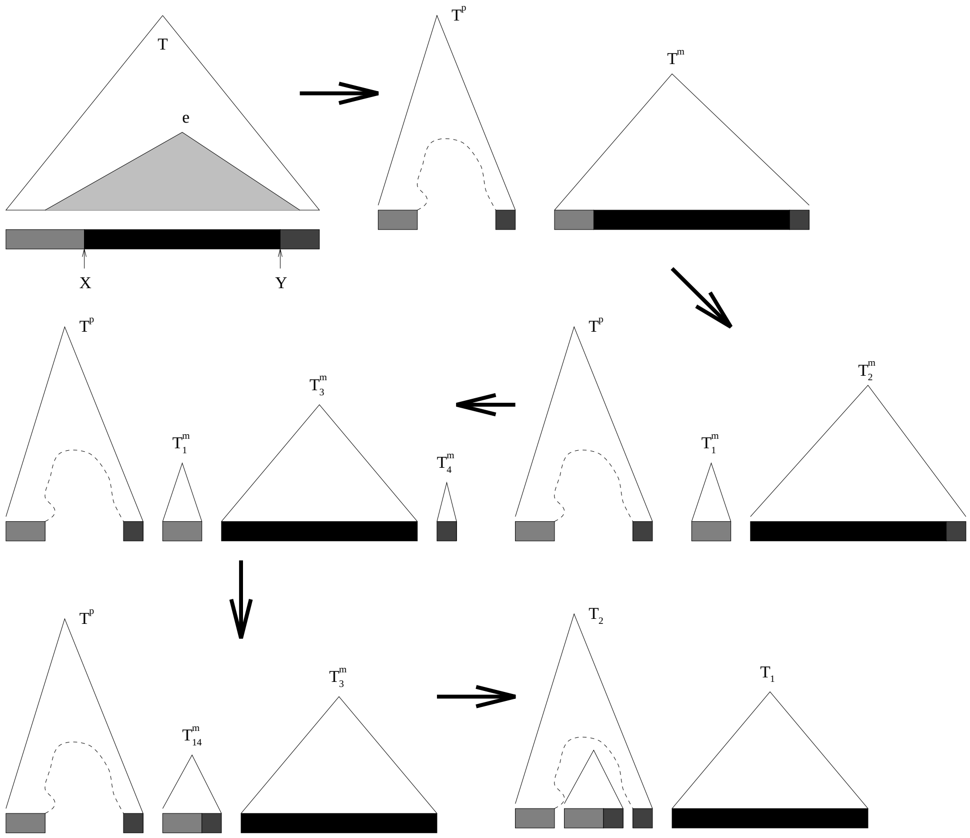


Figure 4.22: Family List Splitting : *Single Root* Case

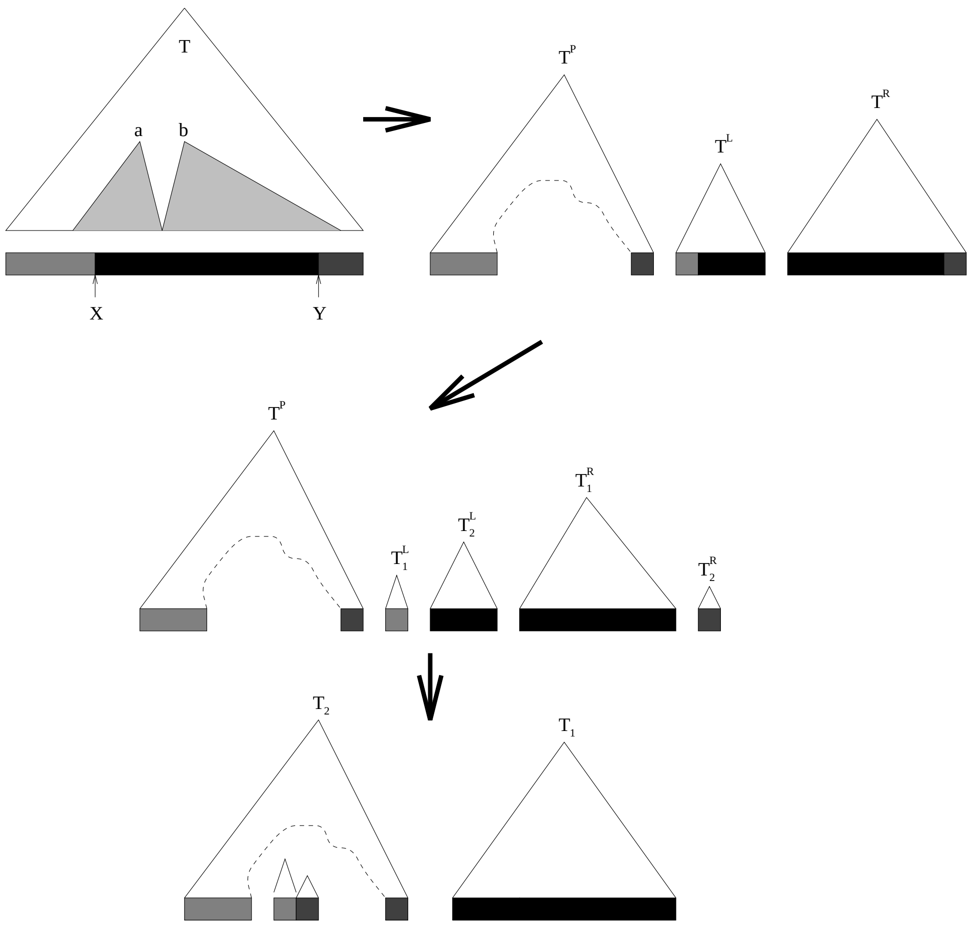


Figure 4.23: Family List Splitting : *Double Root* Case