Data Structure and Algorithm
handout 4
Trees

Apurba Sarkar

October 13, 2017

1 Trees

Definition 1. *A tree is a finite set of one or more nodes such that (i) there is a specially designated node called the root; (ii) the remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots T_n$ where each of these sets is a tree. $T_1, \ldots T_n$ are called the subtrees of the root.*

Let us look carefully at the definition and note down the important points.

1. the definition is recursive.
2. the set is finite and must have atleast one node for it to be tree.
3. an empty set is not considered to be a tree.
4. the condition that $T_1, \ldots T_n$ be disjoint sets prohibits subtrees from ever connecting together.
5. every node in the tree is root of some subtree.

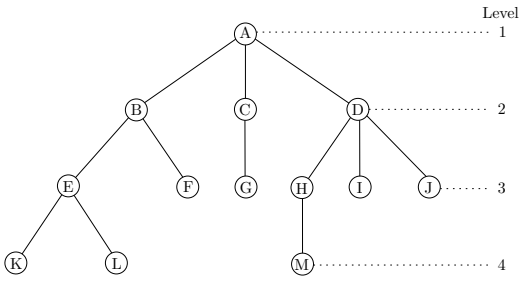


Figure 1: A sample tree

There are few more related terms that one need to know while talking about trees.

Node: A *node* stands for the item of information plus branches to the other other items. Consider the tree in figure. 1. This tree has 13 nodes, each item of data being a single letter for convenience. The root is A and we will normally

draw trees with the root at the top.
Degree of a node: The number of subtrees of a node is called its *degree*. Thus, the *degree* of A is 3, of C is 1 and of F is 0.

Leaf nodes: Nodes that have degree 0 are called *leaf* or *terminal* nodes. K,L,F,G,M,I,J is the set of emphleaf nodes. Alternatively, the other nodes are referred to as *nonterminals*.

Parent and child: The roots of the subtrees of a node, X, are the *children* of X. X is the *parent* of its *children*. Thus, the *children* of D are H, I, J; the parent of D is A.

Siblings: Children of the same *parent* are said to be *siblings*. H, I and J are siblings. We can extend this terminology if we need to so that we can ask for the *grandparent* of M which is D, etc.

Degree of a tree: The degree of a tree is the maximum degree of the nodes in the tree. The tree of figure. 1 has degree 3.

Ancestors: The ancestors of a node are all the nodes along the path from the root to that node. The ancestors of M are A, D and H.

Level of a node: The *level* of a node is defined by initially letting the root be at level one. If a node is at level l, then its children are at level $l+1$. Figure. 1 shows the levels of all nodes in that tree.

Height or depth: The *height or depth* of a tree is defined to be the maximum *level* of any node in the tree.

Forest: A *forest* is a set of $n \geq 0$ disjoint trees. The notion of a forest is very close to that of a tree because if we remove the root of a tree we get a forest. For example, in figure. 1 if we remove A we get a *forest* with three trees.

There are other ways to draw a tree. One useful way is as a list. The tree of figure. 1 could be written as the list

(A(B(E(K,L),F),C(G),D(H(M),I,J)))

The information in the root node comes first followed by a list of the subtrees of that node. Now, how do we represent a tree in memory? If we wish to use linked lists, then a node must have a varying number of fields depending upon the number of branches.

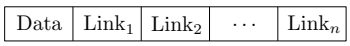


Figure 2: Structure of a typical node of a tree

However, it is often simpler to write algorithms for a data representation where the node size is fixed as shown in figure 2.

2 Binary Trees

A binary tree is an important type of tree structure which appears very often in computer science. It is characterized by the fact that any node can have at most two branches, i.e. there is no node with degree greater than two. For binary trees we distinguish between the subtree on the left and on the right, whereas for trees the order of the subtrees was irrelevant. Also a binary tree may have zero nodes. Thus a binary tree is altogether a different object than a tree. The definition of the binary trees is as follows:

Definition 2. *A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.*

Again few important observations to note from the definition

1. The definition is recursive.
2. Unlike tree binary tree can be empty.
3. Order of the subtree does matter here, i.e., we distinguish between left subtree and right subtree.

Let us spend some more time to distinguish between a binary tree and a tree. First of all there is no tree having zero nodes because from the definition of the the tree it should atleast have one node which is t root of the tree, but there can be an empty binary tree. The two binary trees shown in Fig. 3 are different.

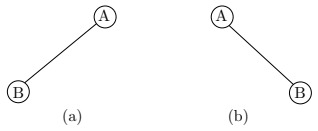


Figure 3: Binary trees

The first one has an empty right subtree whereas the second has an empty left subtree. If they are regarded as trees, then they are the same tree despite the fact that they are drawn slightly differently.

Figure 4 shows two sample binary trees. These two trees are special kinds of binary trees. The first one is a *skewed* binary tree, skewed to the left. There is corresponding one which is skews to the right. The second, the one in the figure 4(b) is called a complete binary tree. The terms that we have defined for trees such as *degree, level, height, leaf, parent, and child* will all apply to binary trees naturally as well.

Before we discuss about the representation of binary tree in memory, let us first prove some relevant fact about binary trees.

Lemma 1. *The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.*

Proof. The proof is by induction on i.
Induction Base: The root is the only node on level $i = 1$. Hence the maximum

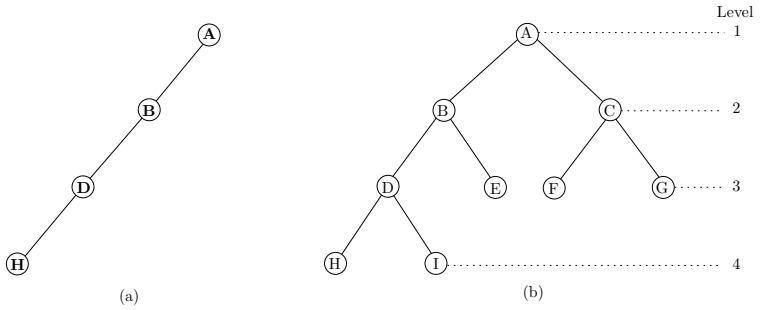


Figure 4: Sample binary trees

number of nodes on level $i = 1$ is $2^0 = 2^{i-1}$.
Induction Hypothesis: Let for all j, $1 \leq j < i$, the maximum number of nodes on level j is 2^{j-1}.
Induction Step: By induction hypothesis, the maximum number of nodes on level $i-1$ is 2^{i-2}. Since each node in a binary tree can have at most two children, the the maximum number of nodes on level i is 2 times the maximum number of nodes on level $i-1$, which is $2^{i-2}.2 = 2^{i-1}$. □

Lemma 2. *The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.*

Proof. By lemma 1, the maximum number of nodes on level $i = 1$ is $2^0 = 2^{i-1}$. So, the total number of nodes in a binary tree of depth k is sum of the maximum number of nodes in all the level i, $1 \leq i \leq k$, which is $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$ □

Lemma 3. *For any nonempty binary tree, T, if n_0 is the number of terminal nodes and n_2, the number of nodes with degree 2, then $n_0 = n_2 + 1$.*

Proof. Let n_1 be the number of nodes of degree 1 and n be the total number of nodes in the binary tree. Since all nodes in T are of degree ≤ 2 we have

$$n = n_0 + n_1 + n_2 \tag{1}$$

If we count the number of branches in a binary tree, we see that every node except for the root has a branch leading into it. If B is the number of branches, then $n = B + 1$. All branches emanate either from a node of degree one or from a node of degree 2. Thus, $B = n_1 + 2n_2$. Hence, we obtain

$$n = 1 + n_1 + 2n_2 \tag{2}$$

Subtracting Eqn. 2 from Eqn. 1 and rearranging the terms we get

$$n_0 = n_2 + 1 \tag{3}$$

□

3 Representation of a Binary Tree

Definition 3. *A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes.*

By lemma 2, this is the maximum number of node such a binary tree can have. Figure 5 shows a full binary tree of depth 4. A very elegant sequential representation for such binary trees results from sequentially numbering the nodes, starting with nodes on level 1, then those on level 2 and so on. Nodes on any level are numbered from left to right (see figure 5). This numbering scheme gives us the definition of a complete binary tree.

Definition 4. *A binary tree with n nodes and of depth k is complete iff its nodes correspond to the nodes which are numbered one to n in the full binary tree of depth k. In other words, A binary tree is said to be complete if it contains the maximum number of nodes in all but last level and nodes in the last level appear from left to right.*

The nodes may now be stored in a one dimensional array, *tree*, with the node umbered i being stored in *tree[i]*. Lemma 4 enables us to easily determine the locations of the parent, left child and right child of any node i in the binary tree.

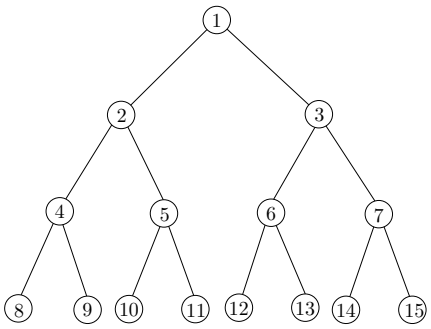


Figure 5: A full binary tree of depth 4 with sequential node numbering

Lemma 4. *If a complete binary tree with n nodes (i.e., depth $= \lfloor \lg 2n \rfloor + 1$) is represented sequentially as in figure 5 then for any node with index i, $1 \leq i \leq n$ we have:*
(i) $parent(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$. When $i = 1, i$ is the root and has no parent.
(ii) $lchild(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
(iii) $rchild(i)$ is at $2i+1$ if $2i+1 \leq n$. If $2i+1 > n$, then i has no right child.

Proof. We prove (ii). (iii) is an immediate consequence of (ii) and the numbering of nodes on the same level from left to right. (i) follows from (ii) and (iii). We prove (ii) by induction on i. For $i = 1$, clearly the left child is at 2 unless $2 > n$ in which case 1 has no left child. Now assume that for all $j, 1 \leq j \leq i$, $lchild(j)$ is at $2j$. Then, the two nodes immediately preceding $lchild(i+1)$ in the representation are the right child of i and the left child of i. The left child of i is at $2i$. Hence, the left child of $i+1$ is at $2i+2 = 2(i+1)$ unless $2(i+1) > n$ in which case $i+1$ has no left child. □

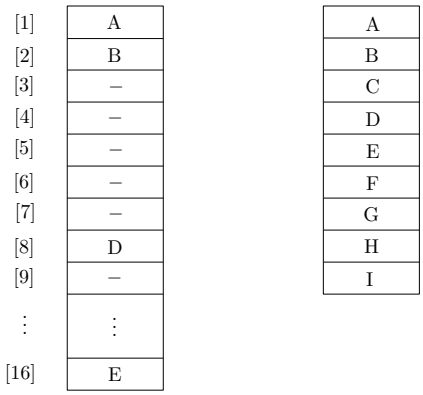
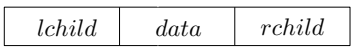


Figure 6: Array representation of tree of (a) figure 4(a) and (b) figure 4(b)

Let us stop for a moment and analyse the space requirement in this kind of sequential representation. This representation can clearly be used for all binary trees though in most cases there will be a lot of unutilized space. For complete binary trees the representation is ideal as no space is wasted. For the skewed tree of figure 4(a), however, less than half the array is utilized. In the worst case a skewed tree of depth k will require $2^k - 1$ spaces. Of these only k will be occupied.

The array representation of the binary tree in figure 4(a) and (b) are shown in figure 6 (a) and (b) respectively. It is clearly evident that in the array representation of the skewed tree of figure 4(a), spaces are wasted (figure 6(a)); whereas since the tree of figure 4(b) is a complete binary tree no spaces are wasted (figure 6(b))

So, it is evident that the while array representation appears to be good for complete binary trees it is wasteful for many other binary trees. In addition, the representation suffers from the general inadequacies of sequential representations. Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in level number of these nodes. These problems can be easily overcome through the use of a linked representation. Each node will now have three fields *lchild*, *data* and *rchild*. The structure of a typical binary tree node and its corresponding structure declaration in C are shown bellow: While this node structure will make it difficult



(a)

```
typedef struct nodetype {
   struct nodetype * lchild;
   int data;
   struct nodetype * rchild;
   } treenode;
```

(b)

Figure 7: (a)A typical node (b) corresponding structure declaration in C

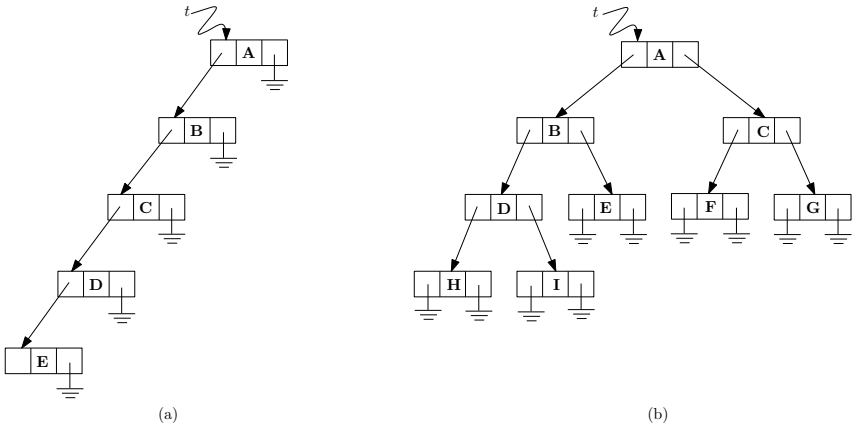


Figure 8: Linked representation of tree of (a) figure 4(a) and (b) figure 4(b)

to determine the parent of a node, we shall see that for most applications, it is adequate. In case it is necessary to be able to determine the parent of random nodes. then a fourth field *parent* may be included. The representation of the binary trees of figure 4 using this node structure is given bellow in figure 8. A tree is referred to by the pointer variable that points to its root.

4 Binary Tree Traversal

There are several operations that we often want to perform on trees and one important operation that arises frequently is the traversal of a tree or visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree. When traversing a binary tree we want to treat each node and its subtrees in the same fashion. If we let L, D, R stand for moving left, printing the data, and moving right when at a node then there are six possible combinations of traversal: LDR, LRD, DLR, DRL, RDL, and RLD. If we adopt the convention that we traverse left before right then only three traversals remain: LDR, LRD and DLR. To these we assign the names inorder, postorder, and preorder because there is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an expression. Consider the binary tree of figure 9. This tree contains an arithmetic expression with binary operators: add(+), multiply(*), divide(/), exponentiation($\wedge$) and variables A, B, C, D, and E. Let us not worry for now how this binary tree was formed, but assume that it is available. We will define three types of traversals and show the results for this tree.

4.1 Inorder Traversal

In this case you move down the tree towards the left until you can go no farther. Then you "visit" the node, move one node to the right and continue same procedure again. If you cannot move to the right, go back one more node up i.e. to the parent. Naturally you can see recursion is involved in this procedure and procedure is listed bellow:

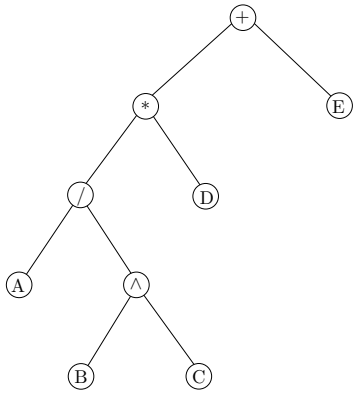


Figure 9: Binary tree with arithmetic operations

```
void inorder(treenode *t)
{
   if(t != NULL)
   {
      inorder(t->lchild);
      printf("%d ",t->data);
      inorder(t->rchild);
   }
}
```

Recursion is an elegant tool for describing this traversal and if you trace it out, the elements get printed in the order A/B ∧ C * D + E.

4.2 Preorder Traversal

In this case you visit a node, traverse left and continue again. When you cannot continue, move right and begin again or move back until you can move right and resume. If you trace out the traversal it would print in *preorder* as +* / A ∧ B C D E which is the *prefix* form of the expression. The recursive procedure of the *preorder* traversal is listed bellow:

```
void preorder(treenode *t)
{
   if(t != NULL)
   {
      printf("%d ",t->data);
      preorder(t->lchild);
      preorder(t->rchild);
   }
}
```

By now you must have guessed what should be the procedure for *postoder* traversal.

4.3 Postorder Traversal

In this case you move down left as far as you can. If you can not move any further then move to right and repeat the procedure. If you can tot also move to the right then visit the node (i.e., print the data) and move back one node up (i.e. to the parent) and repeat. The recursive procedure of the *postorder* traversal is listed bellow:

```
void postorder(treenode *t)
{
   if(t != NULL)
   {
      postorder(t->lchild);
      postorder(t->rchild);
      printf("%d ",t->data);
   }
}
```

If you trace out the traversal it would print in *postorder* as A B C $\wedge$ / D * E + which is the *postfix* form of the expression.

5 Nonrecursive traversal

So far we have written recursive algorithms for tree traversal, these algorithm can also be easily implemented in a nonrecursive way. Let us explain nonrecursive traversal procedure for each of the algorithm.

5.1 Preorder Traversal

The nonrecursive preorder traversal uses a pointer *ptr*, which points to the node currently being scanned. The algorithm also uses a stack to hold the address of nodes for future processing. Initially NULL is pushed onto the stack as sentinel. The pointer *ptr* is initialized to *root* of the tree, i.e. $ptr = root$. Then following steps are repeated until ptr = NULL or, equivalently while $ptr \neq$ NULL.

(a) Proceed down the left-most path rooted at *ptr*, processing (i.e. printing) each node N on the path and pushing each right child $rchild$(N), if any, onto the stack. The traversing ends after a node N with no left child is processed.

(b) Pop and assign to *ptr* the top element on stack. If $ptr \neq$ NULL, then return to Step (a); otherwise exit. A formal presentation of preorder traversal algorithm is shown bellow:

Algorithm 1: PREORDER()

```
   // A binary tree t is in memory.  The algorithm traverse the
      tree in preorder.  An array stack S is used to temporarily
      hold the pointers to nodes.
1  top = 1, S[top] = NULL, ptr = t;
2  while ptr ≠ NULL do
3      print or process ptr → data;
4      if ptr → rchild ≠ NULL then
           // right child exists
5          push(ptr → rchild);
6      if ptr → lchild ≠ NULL then
7          ptr = ptr → lchild;
8      else
9          ptr =pop();
```

5.2 Inorder Traversal

The nonrecursive inorder traversal also uses a pointer *ptr*, which points to the node currently being scanned and a stack which will hold the address of nodes for future processing. With this algorithm, a node is processed only when it is popped from the stack. Initially NULL is pushed on to the stack. A pointer *ptr* is initialized to *root* of the tree, i.e. $ptr = root$. Then following steps are repeated until NULL is popped from the stack.

(a) Proceed down the left-most path rooted at *ptr*, pushing each node N onto the stack and stopping when a node N with no left child is pushed onto stack.

(b) Pop and process(i.e., print) the nodes on stack until

  (i) a node N with right child *rchild*(N)is popped in which case *ptr* set to *rchild*(N) i.e. $ptr = rchild(N)$ and return to Step(a) or

  (ii) if NULL is popped, then exit

It is to be noted that a node is processed only when it is popped from the stack. A formal presentation of Inorder traversal algorithm is shown bellow:

Algorithm 2: INORDER()

```
   // A binary tree t is in memory. The algorithm traverse the
      tree in inorder. An array stack S is used to temporarily
      hold the pointers to nodes.
 1 top = 1, S[top] = NULL, ptr = t;
 2 while ptr ≠ NULL do
 3 |   push(ptr);
 4 |   ptr = ptr → lchild;
 5 ptr =pop();
 6 while ptr ≠ NULL do
 7 |   print or process ptr → data;
 8 |   if ptr → rchild ≠ NULL then
   |   |   // right child exists
 9 |   |   ptr = ptr → rchild;
10 |   |   Go to Step 2;
11 |   ptr =pop();
```

5.3 Posorder Traversal

The postorder traversal algorithm is somewhat complicated than the preceding two algorithms, because we may have to save nodes in two different situations. We distinguish between the two cases by pushing either N or its negative, −N onto stack. Again a pointer variable, *ptr* is used which points to the node currently being scanned and a stack which will hold the address of nodes for future processing. Initially, the pointer, *ptr* is initialized to *root* of the tree, i.e. $ptr = root$. Then following steps are repeated until NULL is popped from the stack.

(a) Proceed down the left-most path rooted at *ptr*, At each node N of the path, push N onto the stack and, if node N has a right child, push $-rchild$(N) also onto stack.

(b) Pop and process(i.e., print) the positive nodes on stack. If NULL is popped, then exit. If a negative node is popped, that is, if $ptr = -$N for some node N, set $ptr =$ N and return to step (a).

It is to be noted that a node is processed only when it is popped from the stack and it is positive. A formal presentation of postorder traversal algorithm is shown bellow:

Algorithm 3: POSTORDER()

```
   // A binary tree t is in memory. The algorithm traverse the
      tree in postorder. An array stack S is used to
      temporarily hold the pointers to nodes.
 1 top = 1, S[top] = NULL, ptr = t;
 2 while ptr ≠ NULL do
 3 |   push(ptr);
 4 |   if ptr->rchild ≠ NULL then
 5 |   |   push(-ptr->rchild);
 6 |   ptr = ptr->lchild;
 7 ptr = pop();
 8 while ptr > 0 do
   |   // ptr is positive
 9 |   print or process ptr->data;
10 |   ptr = pop();
11 if ptr < 0 then
   |   // ptr is negative
12 |   ptr = -ptr;
13 |   Go to Step 2;
```

6 Threaded Binary Trees

If we look carefully at the linked representation of any binary tree, we notice that there are more NULL links than actual pointers. As we saw before, there are $n + 1$ NULL links and $2n$ total links. A clever way to make use of these NULL links has been devised by A. J. Perlis and C. Thornton. Their idea is to replace the NULL links by pointers, called threads, to other nodes in the tree. If the $rchild(p)$ is NULL, we will replace it by a pointer to the node which would be printed after p when traversing the tree in inorder. A NULL $lchild$ link at node p is replaced by a pointer to the node which immediately precedes the node p in inorder. Figure shows the binary tree of figure 10 with its new threads drawn in as dashed lines. The tree t has 9 nodes and 10 NULL links

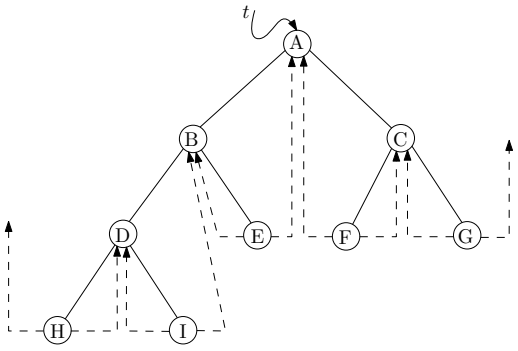


Figure 10: Threaded binary tree

which have been replaced by threads. If we traverse t in inorder the nodes will be

visited in the order H D I B E A F C G. For example node E has a predecessor thread which points to B and a successor thread which points to A. In the memory representation we must be able to distinguish between threads and normal pointers. This is done by adding two boolean fields to the structure, *lthread* and *rthread*. If for a node p, $p \rightarrow lthread = true$, then $p \rightarrow lchild$ contains a thread and otherwise it contains a pointer to the *lchild*. Similarly, if $p \rightarrow rthread = true$ then $p \rightarrow rchild$ contains a thread and otherwise it contains a pointer to the *rchild*. This node structure is now given by the following C declaration.

```
typedef enum { false, true } bool;
typedef struct threadedtreepointer
{
   bool lthread;
   bool rthread;
   int data;
   struct threadedtreepointer *lchild;
   struct threadedtreepointer *rchild;
} threadpointer;
```

In figure 10 we see that two threads have been left dangling in $lchild(H)$ and $rchild(G)$. In order that we leave no loose threads we will assume a head node for all threaded binary trees. Then the complete memory representation for the tree of figure 10. is shown in figure 12. The tree t is the left subtree of the head node. We assume that an empty binary tree is represented by its head node as in figure 11. This assumption will permit easy algorithm design. Now that we

Figure 11: Empty threaded binary tree

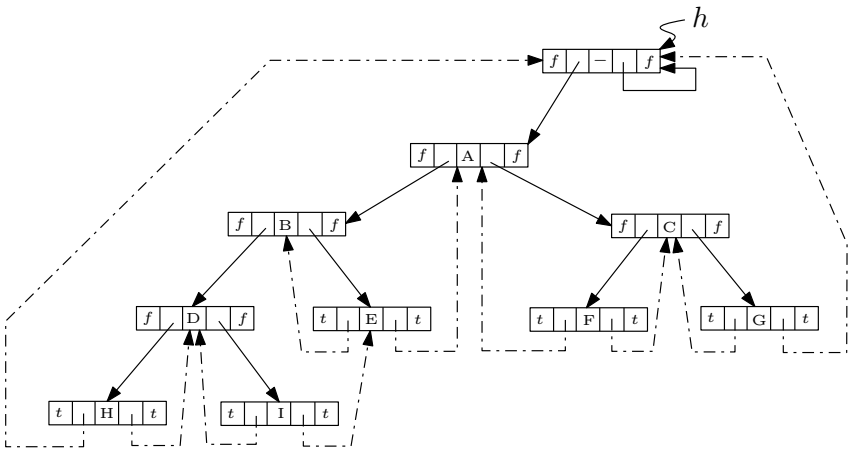


Figure 12: Memory representation of threaded binary tree

have made use of the old null links we will see that the algorithm for inorder

traversal is simplified. First, we observe that for any node x in a binary tree, if $x \rightarrow rthread = true$, then the inorder successor of x is $rchild(x)$ by definition of threads. Otherwise the inorder successor of x is obtained by following a path of the left child links from the right child of x until a node with $lthread = true$ is reached. The procedure *insuc()* finds the inorder successor of any node x in a threaded binary tree.

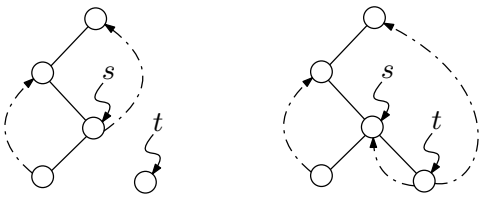
```
threadedpointer * insuc(threadedpointer *t)
{
   threadedpointer *temp;
   temp = t->rchild;
   if(t -> rthread == false )
   {
      while(temp->lthread != false)
           temp = temp->lchild;
   }
   return temp;
}
```

The interesting thing to note about procedure *insuc()* is that it is now possible to find the inorder successor of an arbitrary node in a threaded binary tree without using an additional stack. If we wish to list in inorder all the nodes in a threaded binary tree, then we can make repeated calls to the procedure *insuc()*. Since the tree is the left subtree of the head node and because of the choice of $x \rightarrow rthread = false$ for the head node, the inorder sequence of nodes for tree t is obtained by the procedure *tinoder()*.

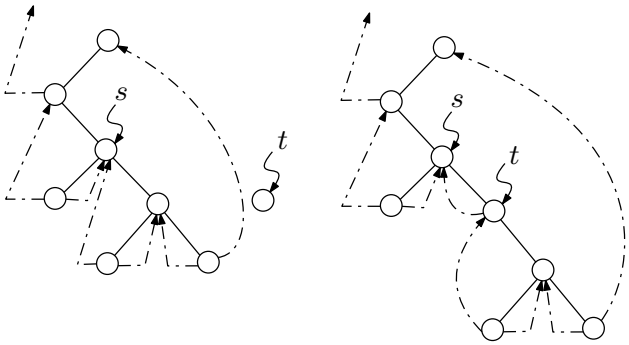
```
void tinorder(threadedpointer *t)
{
   threadedpointer *temp;
   temp = t;
   repeat{
      temp = insuc(temp);
      if(temp != t)
         printf("%d ", temp->data)
   }until(temp == t);
}
```

We have seen how to use the threads of a threaded binary tree for inorder traversal. These threads also simplify the algorithms for preorder and postorder traversal. Before closing this section let us see how to make insertions into a threaded tree. This will give us a procedure for growing threaded trees. We shall study only the case of inserting a node t as the right child of a node s. The case of insertion of a left child is given as an exercise. If s has an empty right subtree, then the insertion is simple and diagrammed in figure 13(a). If the right subtree of s is non-empty, then this right subtree is made the right subtree of t after insertion. When this is done, t becomes the inorder predecessor of a node which has a $lthread = true$ and consequently there is a thread which has to be updated to point to t. The node containing this thread was previously the inorder successor of s. Figure 13(b) illustrates the insertion for this case. In both cases s is the inorder predecessor of t. The details are spelled out in procedure *insertright*.

```
void insertright(threadedpointer *s, threadedpointer *t)
// insert node t as the right child of s.
{
   threadedpointer *temp;
   t->rchild = s->rchild;
   t->rthread = s->rthread;
   t->lchild = s;
   t->lthread = true;
   s->rchild = t;
   s->rthread = false;
   if(t->rthread = false)
   {
      temp = insuc(t);
      temp->lchild = t;
   }
}
```



(a)



(b)

Figure 13: Insertion of a node in threaded binary tree

7 Binary Search Trees

Definition 5. *A binary search tree T is a binary tree which is either empty or each node in the tree contains an identifier and:*
(i) all identifiers in the left subtree of T are less (numerically or alphabetically) than the identifier in the root node of T;

(ii) all identifiers in the right subtree of T are greater than the identifier in the root node of T;
(iii) the left and right subtrees of T are also binary search trees.

For a given set of identifiers several binary search trees are possible. Figure 14 shows two possible binary search trees for set of identifiers $\{15, 20, 25, 30, 35, \}$. To determine whether an identifier x is present in a binary search tree, x is compared with the root. If x is less than the identifier in the root, then the search continues in the left subtree; if x equals the identifier in the root, the search terminates successfully; otherwise the search continues in the right subtree. This is formalized in algorithm *search()*.

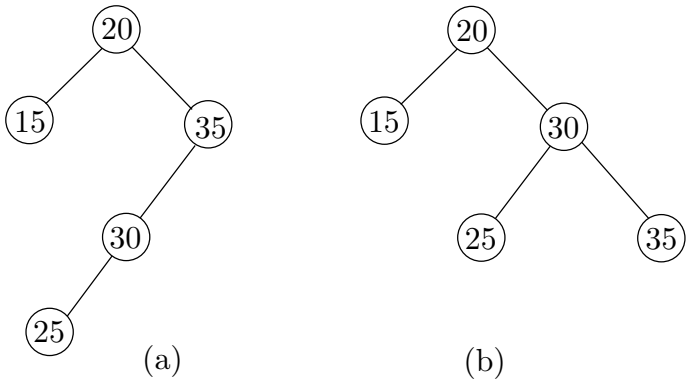


Figure 14: Two possible binary search trees

```
typedef enum { false, true } bool;
treepointer * search(treepointer *t, identifier x)
// search the binary search tree T for x. Each node has fields lchild,
    ident, rchild. Return NULL if x is not in T. Otherwise, return the
    pointer ret such that ret->ident = x.
{
   treepointer *ret = t;
   bool found = false;
   while(ret != NULL && !found)
   {
      if( x < ret->ident )
         ret = ret->lchild;
      else if(x > ret->ident )
         ret = ret->rchild;
      else
         found = true;
   }
   return ret;
}
```

Consider the binary search tree corresponds to using algorithm SEARCH on the binary search tree of figure 15. While this tree is a full binary tree, it need not be optimal over all binary search trees for this set of identifiers when the identifiers are searched for with different probabilities. In order to determine an optimal binary search tree for a given static set of identifiers, we must first

decide on a cost measure for search trees. In searching for an identifier at level k using the procedure *search*, k iterations of the *while* loop are made. Since this loop determines the cost of the search, it is reasonable to use the level number of a node as its cost. Consider the two search trees of figure 14. As identifiers are

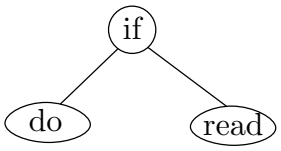


Figure 15: Binary search tree corresponding to identifiers {do, if, read}

encountered a match is searched for in the tree. The second binary tree requires at most three comparisons to decide whether there is a match. The first binary tree may require four comparisons, since any identifier which numerically comes after 20 but precedes 30 will test four nodes. Thus, as far as worst case search time is concerned, this makes the second binary tree more desirable than the first. To search for an identifier in the first tree takes one comparison for the 20, two for each of 15 and 35, three for 30 and four for 25. Assuming each is searched for with equal probability, the average number of comparisons for a successful search is 2.4. For the second binary search tree this amount is 2.2. Thus, the second tree has a better average behavior, too.

In evaluating binary search trees, it is useful to add a special "square" node at every place where there is a null link. Doing this to the trees of figure 14 yields the trees of figure 16. Remember that every binary tree with n nodes has $n + 1$ null links and hence it will have $n + 1$ square nodes. We shall call these nodes external nodes because they are not part of the original tree. The remaining nodes will be called internal nodes. Each time a binary search tree is examined for an identifier which is not in the tree, the search terminates at an external node. Since all such searches represent unsuccessful searches, external nodes will also be referred to as *failure* nodes. A binary tree with external nodes added is an extended binary tree. Figure 16 shows the extended binary trees corresponding to the search trees of figure 14. We define the external path length of a binary tree to be the sum over all external nodes of the lengths of the paths from the root to those nodes. Analogously, the internal path length is defined to be the sum over all internal nodes of the lengths of the paths from the root to those nodes. For the tree of figure 14(a) we obtain its internal path length, L_I, to be:

$$L_I = 0 + 1 + 1 + 2 + 3 = 7$$

Its external path length, L_E is

$$L_E = 2 + 2 + 4 + 4 + 3 + 2 = 17$$

The internal path length and external path length of a binary tree is related by the formula $L_E = L_I + 2n$, n being the total number of internal nodes. Before we prove this formula let us revisit the idea of internal and external path length. Informally, a node's path length is the number of links (or branches) required

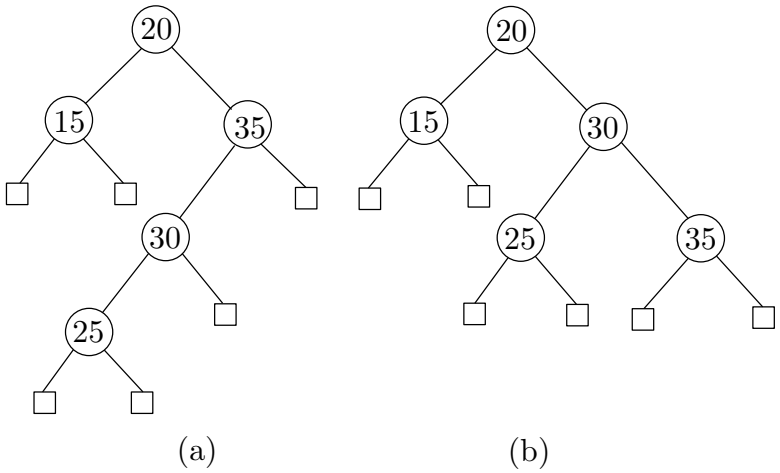


Figure 16: Extended binary trees corresponding to search trees in figure 14

to get back to the root. The root has path length zero and the maximum path length in a tree is called the tree's height. The sum of the path lengths of a tree's internal nodes is called the internal path length and the sum of the path lengths of a tree's external nodes is called the external path length.

Consider a full binary tree A with N_A nodes, internal path length $L_{I,A}$ and external path length $L_{E,A}$. If we make this tree the left subtree of a root node (figure 17), then each path length, internal and external, in the left subtree increases by one:

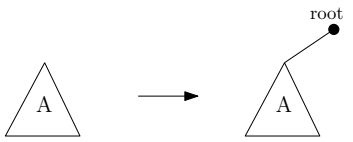


Figure 17: The tree A is made left subtree of the root

$$
\begin{aligned}
N &= N_A + 1 \\
I &= I_A + 1 \\
E &= E_A \\
L_I &= L_{I,A} + I_A \\
L_E &= L_{E,A} + E_A = L_{E,A} + I_A + 1
\end{aligned}
$$

Now add another tree, B, as the right subtree of the root node (figure 18):

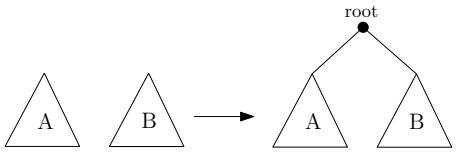


Figure 18: Tree B is added as right subtree

$$
\begin{aligned}
N &= N_A + N_B + 1 \\
I &= I_A + I_B + 1 \\
E &= E_A + E_B \\
L_I &= (L_{I,A} + I_A) + (L_{I,B} + I_B) \\
&= (L_{I,A} + L_{I,B}) + I - 1 \\
L_E &= (L_{E,A} + E_A) + (L_{E,B} + E_B) \\
&= (L_{E,A} + L_{E,B}) + E \\
&= (L_{E,A} + L_{E,B}) + I + 1
\end{aligned}
$$

Lemma 5. *If L_E and L_I be the external and internal path length respectively of a binary three with n nodes then $L_E = L_I + 2n$.*

Proof. The proof will be by induction on I, the number of internal nodes. The basis step is when $I = 0$. That means the root is an external node - it has no children:

The internal path length, the external path length, and the number of internal nodes all equal zero so the equation applies and the basis step is established.

For the inductive step, assume that the equation is true for all numbers of internal nodes up to and including I. We want to show that it is true for $I + 1$ internal nodes.

Consider that the tree with $I + 1$ internal nodes (figure 18) comprises a root and its two subtrees (which may be empty):

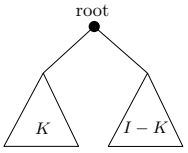


Figure 19: Tree with $I + 1$ internal nodes

If the number of internal nodes in the left subtree is K, then the number in the right subtree is $I - K$, so that the total number of internal nodes, including the root, is $I + 1$. The inductive assumption is:

$$
\begin{aligned}
L_E(K) &= L_I(K) + 2K \\
L_E(I - K) &= L_I(I - K) + 2(I - K)
\end{aligned}
$$

We will apply the identities we established above. Write the internal path length

of the new tree in terms of the internal path lengths of the two subtrees:

$$L_I(I+1) = L_I(K) + L_I(I-K) + (I+1) - 1$$
$$L_I(K) + L_I(I-K) = L_I(I+1) - I$$

Do the same for the external path lengths and substitute in the inductive assumption:

$$L_E(I+1) = L_E(K) + L_E(I-K) + (I+1) + 1$$
$$L_E(I+1) = [L_I(K) + 2k] + [L_I(I-K) + 2(I-K)] + I + 2$$
$$L_E(I+1) = [L_I(K) + L_I(I-K)] + 3I + 2$$
$$L_E(I+1) = [L_I(I+1) - I] + 3I + 2$$
$$L_E(I+1) = L_I(I+1) + 2(I+1)$$

This is the result we required to complete the proof. □

It is evident from the Lemma 5, that a binary trees with maximum L_E also have maximum L_I. Now the most pertinent question at this point is what would be the maximum and minimum values for L_I over all possible binary trees with n internal nodes? The worst case, clearly, is when the tree is skewed (i.e. when the depth of the tree is n). In this case,

$$L_I = \sum_{i=0}^{n-1} i = n(n-1)/2$$

To obtain trees with minimal L_I, we must have as many internal nodes as close to the root as possible. We can have at most 2 nodes at distance 1, 4 nodes at distance 2, and in genera the smallest value for L_I would be

$$0 + 2.1 + 4.2 + 8.3 + \ldots$$

This can simplified as

$$\sum_{1 \le k \le n} \lfloor \lg k \rfloor = O(n \lg n)$$

This explanation tells us that tree with minimal internal path length would be a complete binary tree defined in definition 4.

To see how the ideas of internal and external path lengths can be used in applications let us look at the following simple problem: We are given a set of $n+1$ positive weights $q_1, q_2, \ldots, q_{n+1}$. Exactly one of these weights is to be associated with each of the $n+1$ external nodes in a binary tree with n internal nodes. The weighted external path length of such a binary tree is defined to be $\sum_{1 \le i \le n+1} q_i k_i$ where k_i is the distance from the root node to the external node with weight q_i. The problem is to determine a binary tree with minimal weighted external path length. Note that here no information is contained within internal nodes. For example, suppose $n = 3$ and we are given the four weights: $q_1 = 15, q_2 = 2, q_3 = 4$ and $q_4 = 5$. Two possible trees would be as shown bellow in figure 20.

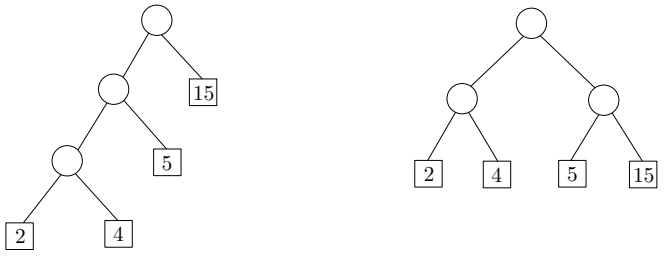


Figure 20: Two possible weighted external path length

The weighted external path length of the tree in figure 20(a) is $2 \cdot 3 + 4 \cdot 3 + 5 \cdot 2 + 15 \cdot 1 = 43$ and that of the tree in figure 20(b) is $2 \cdot 2 + 4 \cdot 2 + 5 \cdot 2 + 15 \cdot 3 = 52$

Binary trees with minimal weighted external path length find application in several areas. One application is to determine an optimal merge pattern for $n+1$ runs using a 2-way merge. If we have four runs $R_1 - R_4$ with q_i being the number of records in run R_i , $1 \leq i \leq 4$, then the skewed binary tree above defines the following merge pattern: merge R_2 and R_3; merge the result of this with R_4 and finally merge with R_1. Since two runs with n and m records each can be merged in time $O(n + m)$, the merge time following the pattern of the above skewed tree is proportional to $(q_2 + q_3) + \{(q_2 + q_3) + q_4\} + \{q_1 + q_2 + q_3 + q_4\}$. This is just the weighted external path length of the tree. In general, if the external node for run R_i is at a distance k_i from the root of the merge tree, then the cost of the merge will be proportional to $\sum q_i k_i$ which is the weighted external path length.

Another application of binary trees with minimal external path length is to obtain an optimal set of codes for messages $M_1, M_2, \ldots, M_{n+1}$. Each code is a binary string which will be used for transmission of the corresponding message. At the receiving end the code will be decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages. The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a 0 as a left branch and a 1 as a right branch, then the decode tree corresponds to codes $000, 001, 01$ and 1 for messages M_1, M_2, M_3 and M_4 respectively (shown in figure 21). These codes are called Huffman codes. The cost of decoding a

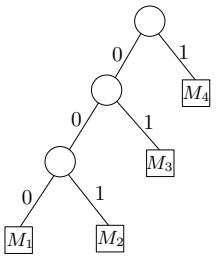


Figure 21: A decode tree

code word is proportional to the number of bits in the code. This number is

equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\sum_{1 \le i \le n+1} q_i d_i$ where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length!

A very nice solution to the problem of finding a binary tree with minimum weighted external path length has been given by D. Huffman. The following type declaration for a node is assumed:

```
typedef struct treepointer
{
   struct treepointer *lchild;
   int weight;
   struct treepointer *rchild;
} treenode;
```

The algorithm *huffman* makes use of a list l of extended binary trees. Each node in a tree has three fields: *weight, lchild* and *rchild*. Initially, all trees in l have only one node. For each tree this node is an external node, and its weight is one of the provided q_i's. During the course of the algorithm, for any tree in l with root node t and depth greater than 1, *weight(t)* is the sum of weights of all external nodes in t. Algorithm *huffman* uses the subalgorithms *least* and *insert*. *least* determines a tree in l with minimum *weight* and removes it from l. *insert* adds a new tree to the list l.

```
void huffman(treenode **l, int n)
// l is a list of n single node binary trees.
{
   treenode *t;
   int i;
   for(i = 1; i < n; i++)
   {
      new(&t);
      t->lchild = least(l);
      t->rchild = least(l);
      t->weight = t->lchild->weight + t->rchild->weight;
      insert(l,t);
   }
}
```

We illustrate the way this algorithm works by an example. Suppose we have the weights $q_1 = 2, q_2 = 3, q_3 = 5, q_4 = 7, q_5 = 9$ and $q_6 = 13$. Then the sequence of trees we would get is shown in figure. 22

The weighted external path length of the tree is $2 \cdot 4 + 3 \cdot 4 + 5 \cdot 3 + 23 \cdot 2 + 7 \cdot 2 + 9 \cdot 2 = 93$. In comparison, the best complete binary tree has weighted path length 95.

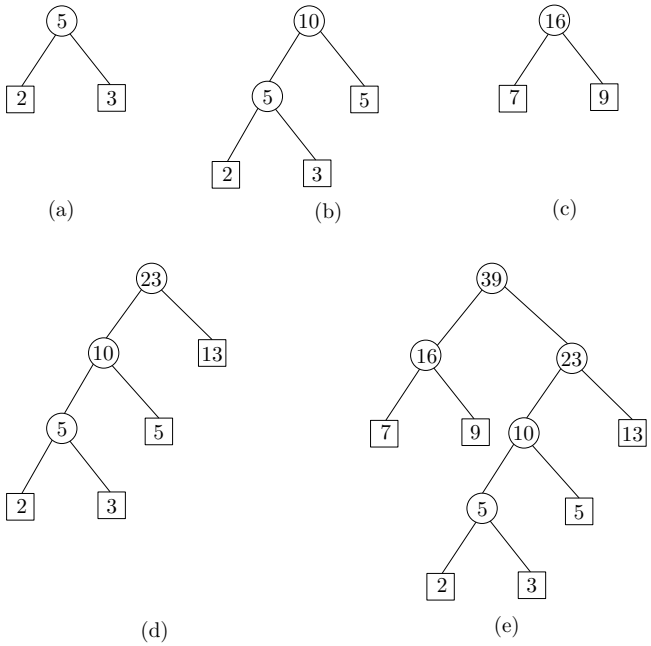


Figure 22: Steps of Huffman algorithm

8 Inserting into a Binary Search tree

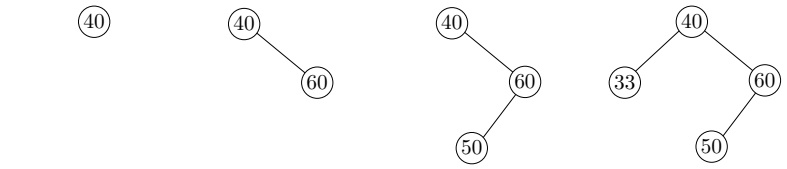
An identifier x may be inserted into a binary search tree t by using the search algorithm of the previous section to determine the failure node corresponding to x. This gives the position in t where the insertion is to be made. Following procedure *insbst* summarizes the steps.

```
typedef enum { false, true } bool;
void insbst(treepointer *t, treepointer *s, identifier x)
// search the binary search tree t for the node pointed by s such that
    s->ident = x. If x is not already in the tree then it is inserted
    at the appropriate point.
{
   treepointer *trail; //trail pointer trails ret
   treepointer *ret;
   bool found;
   ret = t;   trail = NULL; found = false
   while(ret != NULL && !found)
   {
      if( x < ret->ident )
      {
         trail = ret;        //save ret
         ret = ret->lchild;
      }
      else if(x > ret->ident )
      {
         trail = ret;
         ret = ret->rchild;
```

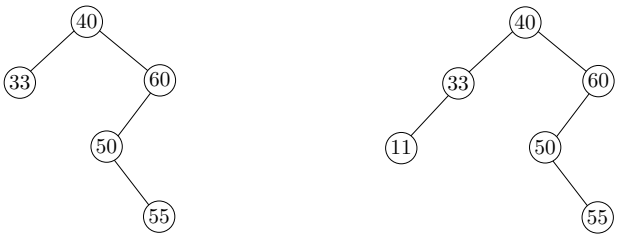
```
      }
      else
         found = true;
   }
   if(!found)  // there is no node with ident x so insert it
   {
      new(ret);
      ret->ident = x;
      ret->lchild = NULL;
      ret -> rchild = NULL;
      if(t = NULL)
         t = ret;
      else if( x < trail->ident)
         trail->lchild = ret;
      else
         trail->rchild = ret;
   }
}
```

Figure 23 shows the binary search tree obtained by entering the following six numbers in order into an initially empty binary search tree.

$$40, 60, 50, 33, 55, 11$$



(a) $x = 40$ (b) $x = 60$ (b) $x = 50$ (b) $x = 33$



(a) $x = 55$ (a) $x = 11$

Figure 23: Insertion in a Binary Search Tree

9 Deleting in a Binary search tree

Suppose there is a binary search tree t and suppose an *item* is given. We now discuss an algorithm which deletes an *item* from the tree t. The deletion algorithm first finds out the location of the node N that contains the *item* and also the location of the parent node P(N). Primarily there are three cases to consider while deleting and *item*.

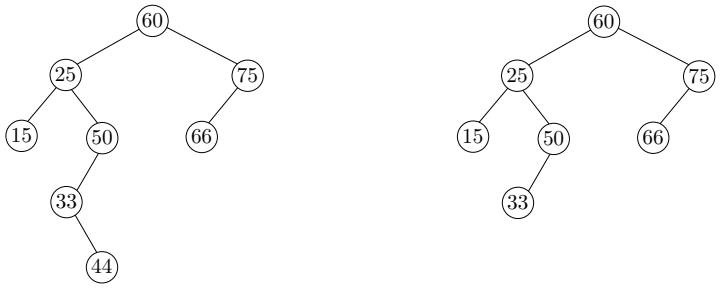
1. N has no children. In this case, the nod N is deleted from t by simply replacing the location of N in the parent node P(N) by the *null* pointer.

2. N has exactly one child. The node N in this case is deleted from t by simply replacing the location of N in P(N) by the location of the only child of N.

3. N has two children. In this case inorder successor of N needs to be found out. Let S(N) denote the inorder successor of N. In oder to delete N from t, first S(N) is deleted from t and then node N is replaced by the node S(N).

It is to be noted that the third case is much more complicated and involved. Let us discuss each of the cases with an example.

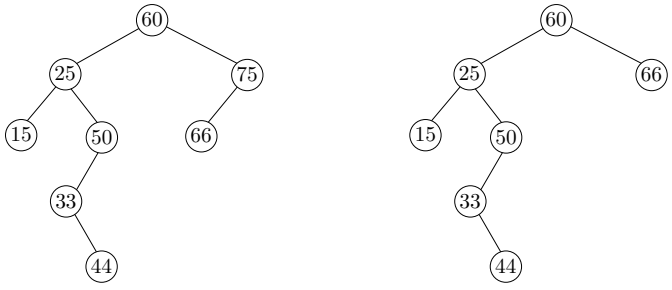
Case1: N has no children. Consider the tree in figure. 24. Suppose the node with value 44 is to be deleted, so N= 44. Figure. 24(a) and (b) shows the tree before and after deletion of 44 respectively.



(a) Before deletion (b) After deletion

Figure 24: Deletion of 44 from t

Case2: N has only one child. Consider the tree in figure. 25. Suppose the node with value 75 is to be deleted, so N= 75. Figure. 25(a) and (b) shows the tree before and after deletion of 44 respectively.

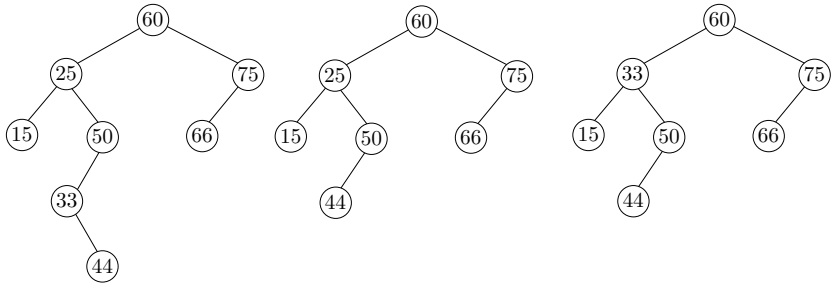


(a) Before deletion (a) After deletion

Figure 25: Deletion of 75 from t

Case3: N has two children. Consider the tree in figure. 26. Suppose the node with value 25 is to be deleted, so N= 25. To delete the node with 25, first its

inorder successor which is 33 in this case needs to be deleted from t. Note that the node with value 33 has right child, so, it is deleted using Case2. Finally, node 25 is replaced with 33 to complete the deletion procedure. Figure. 26(a) represents the original tree, figure. 26(b) shows the tree after deletion of the inorder successor (33) and figure. 26(c) shows the final tree after deletion of the node with value 25. Let us now write down the procedures for deletion.



(a) Before deletion (b) Temporary deletion of 33 (c) After deletion of 25

Figure 26: Deletion of 25 from t

Procedure *caseA* takes care of Case1 and Case2 i.e. when the node to be deleted does not have two children. On the other hand procedure *caseB* deals with the case when the node to be deleted does have two children (i.e. Case3). There are many subcases in the deletion procedures which reflect the fact that N may be a left child, right child or the root. Also, in Case2, N may have a left child or a right child. It is to be noted that the inorder successor of N can be found by moving to the right child of N and then repeatedly to the left until meeting a node with an empty left subtree.

```
procedure caseA(treenode *t, treenode *loc, treenode *left, treenode
    *right, treenode *par)
// This procedure deletes the node N at location loc where N does not
    have two children. The pointer par points to the parent of N, or
    else par = NULL which indicates that N is the root. The pointer
    child bellow points to the only child of N, or else child = null
    indicates N has no children.
{
   treenode* child;
   if(left == NULL && right == NULL) // initialize child
      child = NULL;
   else if(left != NULL)
      child = left;
   else
      child = right;
   if(par != NULL)
      if( loc == par->lchild ) // N is left child of par
         par->lchild = child;
      else (loc == par->rchild ) // N is right child of par
         par->rchild = child;
   else
      t = child;
}
```

```
procedure caseB(treenode *t, treenode *loc, treenode *left, treenode
    *right, treenode *par)
// This procedure deletes the node N at location loc where N has two
    children. The pointer par points to the parent of N, or else par =
    NULL which indicates that N is the root. The pointer insuc bellow
    points to the inorder successor of N and parsuc points to the
    parent of insuc.
{
   treenode *insuc, *parsuc, *ptr, *save;
   save = loc;
   ptr = loc->rchild; // finds insuc and parsuc of N
   while( ptr->lchild != NULL )
      save = ptr;
      ptr = ptr->lchild;
   suc = ptr; parsuc = save;
   //call caseA to delete insuc
   caseA(t, ptr->lchild, ptr->rchild, insuc, parsuc);
   //replace node N by its inorder successor
   if(par != NULL)
      if(par->lchild == loc)
         par->lchild = insuc;
      else
         par->rchild = insuc;
   else
      t = insuc;
   insuc->lchild = loc->lchild;
   insuc->rchild = loc->rchild;
}
```

Let us now formally present the deletion algorithm using procedures *caseA* and *caseB* as stated above.

```
procedure deleteBST(treenode *t, treenode *left, treenode *right, int
    item)
// A binary search tree t is in memory and an item of information is
    given. This algorithm deletes the node with value item
{  treenode *par; //trail pointer trails ret
   treenode *loc;
   loc = t;   par = NULL;
   while(loc != NULL)
   {  if( item < loc->data )
      {  par = loc;       //save loc
         loc = loc->lchild;
      }
      else if(item > loc->data )
      {  par = loc;
         loc = loc->rchild;
      }
      else
         break;          //found the item
   }
   if(loc == NULL)
      print("item not in the tree");
```

```
      exit;
   if(loc->lchild != NULL && loc->rchild != NULL)
      // node has two children so call CaseB
      caseB(t, loc, ptr->lchild, ptr->rchild, par);

   else
      // node has no or one child so call CaseA
      caseA(t, loc, ptr->lchild, ptr->rchild, par);
   free(loc);
}
```

10 AVL Trees

We have discussed binary search tree data structure in the last section. Although binary search tree is used for searching the worst case search time depends on the structure or shape of the tree. For consider the elements A, B, C, ..., Z to be inserted in a binary search tree. The result of this insertion would be a skewed tree (skewed towards right). The disadvantage of such binary search trees is that the worst case search time would be $O(n)$. Therefore there arises the need to maintain the binary search tree to be of balanced height. By doing so we can guaranty that the worst case search time would be $O(\lg n)$.

Adelson-Velskii and Landis in 1962 introduced a binary tree structure that is balanced with respect to the heights of subtrees. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in $O(\lg n)$ time if the tree has n nodes in it. At the same time a new identifier can be entered or deleted from such a tree in time $O(\lg n)$. The resulting tree remains height balanced. The tree structure introduced by them is given the name AVL-tree. As with binary trees it is natural to define AVL trees recursively.

Definition 6. *An empty tree is height balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height balanced iff (i) T_L and T_R are height balanced and (ii) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.*

Definition 7. *The balance factor, $BF(T)$, of a node T in a binary tree is defined to be $h_L - h_R$ where h_L and h_R are the heights of the left and right subtrees of T. For any node T in an AVL tree $BF(T) = -1, 0$ or $+1$.*

The definition of a height balanced binary tree requires that every subtree also be height balanced.

10.1 Insertion into a AVL tree

The process of inserting an element into an AVL tree is same as the process of inserting an element into a binary search tree as we have discussed earlier. However, after the insertion of an element if the balance factor, $BF(T)$, of the tree T is affected so as to render the tree T unbalanced we use the technique called *Rotations* to restore the balance of the search tree. For example consider the AVL tree in figure 27, inserting the element E into the tree makes it unbalanced since in that case $BF(C)$ would become -2.

To perform rotation it is necessary to identify a specific node A whose balance factor is neither 0, 1 or −1 and which is the nearest ancestor to the inserted node on the path from the inserted node to the root. This implies that all the node in the from the inserted node to A will have their balance factors to be either 0, 1 or −1. The rebalancing rotations are classified to as LL, RR, LR and RL as illustrated bellow, based on the position of the inserted node with reference to A.

1. LL *rotation:* inserted node is in the left subtree of the left subtree of A
2. RR *rotation:* inserted node is in the right subtree of the right subtree of A
3. LR *rotation:* inserted node is in the right subtree of the left subtree of A
4. RL *rotation:* inserted node is in the left subtree of the right subtree of A

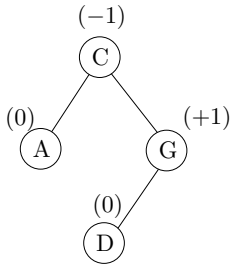
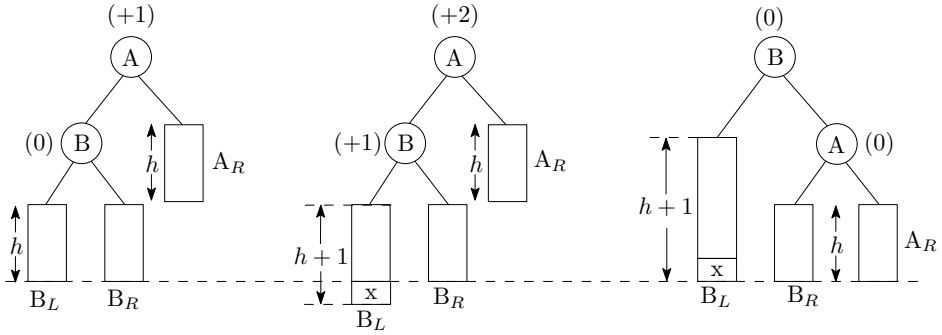


Figure 27: An AVL tree

Each of the rotation are explained with an example.

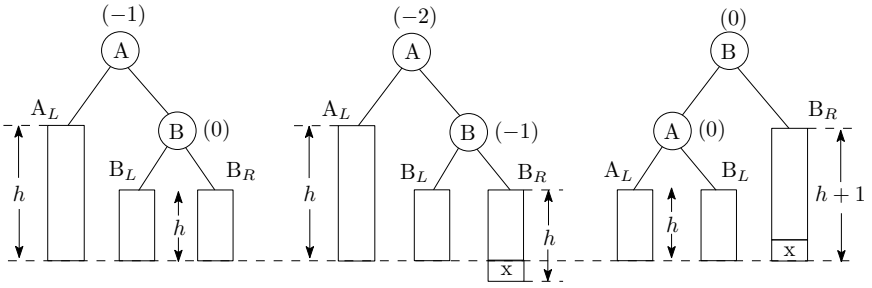
LL Rotation:
The new element x is inserted in the left subtree of left subtree of A, the closest ancestor of the node whose balance factor becomes +2 after insertion. To rebalance the search tree, it is rotated so as to allow B to be the root with B_L and A to be its left subtree and right children respectively and B_R and A_R to be the left subtree and right subtree of A. This is explained in the figure. 28.



(a) Balanced tree (b) Unbalanced tree (c) Rebalanced tree

Figure 28: LL Rotation

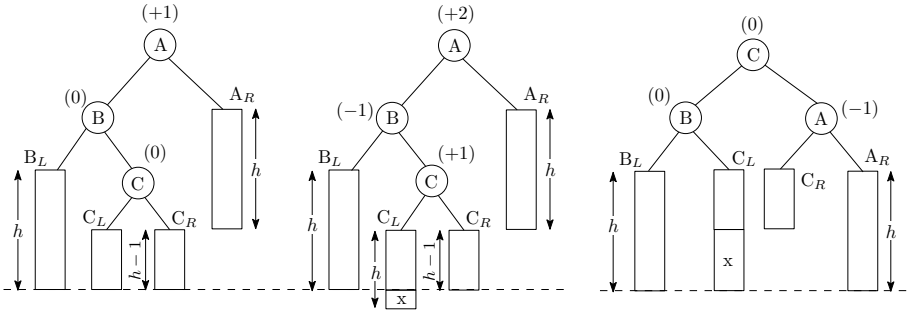
RR Rotation:
In this case the new element x is inserted in the right subtree of right subtree of A. The rebalancing operation pushes B up to become the root with A as its left child and B_R as its right subtree and A_L and B_L as the left subtree and right subtree of A. This is explained in the figure. 29.



(a) Balanced tree (b) Unbalanced tree (c) Rebalanced tree

Figure 29: RR Rotation

LR and RL Rotation:
The balancing methodology for LR and RL rotations are similar but mirror images of one another. Hence, we illustrate LR here and RL is left as an exercise for the reader. Figure. 30 illustrates the LR rotation.



(a) Balanced tree (b) Unbalanced tree (c) Rebalanced tree

Figure 30: LR Rotation

It is to be noted that in all the rotations discussed above, the height of the subtree involved in the rotation is the same after rebalancing has been carried out on the subtree in question. It is not necessary to examine the remaining portion of the tree. The only nodes whose balance factor may change are those in the subtree that is rotated.

Let us now try to find out the answer to the following question. how much time does it take to make an insertion in the AVL tree? An analysis of the algorithm reveals that if h is the height of the tree before insertion, then the time to insert a new item is $O(h)$. This is the same as for unbalanced binary

search trees, though the overhead is significantly greater now. In the case of binary search trees, however, if there were n nodes in the tree, then h could in the worst case be n and the worst case insertion time was $O(n)$. In the case of AVL trees, however, h can be at most $O(logn)$ and so the worst case insertion time is $O(logn)$. To see this, let N_h be the minimum number of nodes in a height balanced tree of height h. In the worst case, the height of one of the subtrees will be $h-1$ and of the other $h-2$. Both these subtrees are also height balanced. Hence, $N_h = N_{h-1} + N_{h-2} + 1$ and $N_0 = 0, N_1 = 1$ and $N_2 = 2$. Note the similarity between this recursive definition for N_h and that for the Fibonacci numbers $F_n = F_{n-1} + F_{n-2}, F_0 = 0 and F_1 = 1$. In fact, we can show that $N_h = F_{h+2} - 1$ for $h \geq 0$. From Fibonacci number theory it is known that $F_h \approx \phi^h/\sqrt{5} - 1$ where $\phi = (1+\sqrt{5})/2$. Hence $N_h \approx \phi^{h+2}/\sqrt{5} - 1$. this means that if there are n nodes in the tree then its height, h, is at most $\log \phi(\sqrt{5}(n+1)) - 2$. The worst case insertion time for a height balanced tree with n nodes is, therefore, $O \log n$

Let us now spell out the the detail procedure to insert a node in the AVL tree in pseudocode. We need the following type definition for a node.

```
typedef struct treepointer
{
   struct treepointer *lchild;
   int data;
   struct treepointer *rchild;
   int bf;
} treenode;
```

With this type definition of a node, the procedure *avlinsert* look as follows: procedure deleteBST(treenode *t, treenode *left, treenode *right, int item)

```
typedef enum { false, true } bool;
procedure avlinser(treenode *t, int x)
// the item x is inserted into the AVL tree with root t. Each node is
    assumed to have an identifier field data, left and right child
    fields and a field bf for balance factor.
{  treenode *a, *b, *c, *f, *p, *q, *cl, *cr,*y;
   bool found, unbalanced;
   int d;
   if(t == NULL)
   {
      new(&y); y->data = x; t = y; t->bf = 0;
      t->lchild = NULL; t->rchild = NULL;
   }
   else
   //locate insertion point for x, a keeps track of most recent node
       with balance factor +1 or -1 and f is the parent of a. q follows
       p through the tree.
   {  f = NULL; a = t; p = t; q = NULL; found = false;
      while(p != NULL && !found)
      {
         if(p->bf != 0)
         {
```

```
      a = p; f = q;
    }
    if( x < p->data )
    {
      q = p; p = p->lchild;
    }
    else if(x > p->data )
    {
      q = p; p = p->rchild;
    }
    else
    {
      y = p; found = true;
    }
  }
}
if(!found)
{ //insert and rebalance. x is not in the t and may be inserted as
    the appropriate child of q.
  new(&y); y->data = x; y->lchild = NULL; y->rchild = NULL; y->bf =
      0;
  if(x < q->data)
    q->lchild = y;
  else
    q->rchild = y;
  //now adjust balance factors of nodes on the path from a to q.
      note that by definition of a all the nodes on this path must
      have balance factors of 0 and so will change to +1 or -1; d=+1
      implies x is inserted in the left subtree of a and d =-1
      implies x is inserted in the right subtree of a.
  if( x > a->data )
  {
    p = a->rchild; b = p; d = -1; // b stores right child
  }
  else
  {
    p = a->lchild; b = p; d = +1; // b stores left child
  }
  while(p != y)
  {
    if(x > p->data)
    {
      p->bf = -1;
      p = p->rchild
    }
    else
    {
      p->bf = +1;
      p = p->lchild;
    }
  }
  unbalanced = true;
  if(a->bf == 0) //tree is still balanced;
  {
```

```
            a->bf = d; unbalanced = false;
         }
         if(a->bf + d == 0) //tree is still balanced;
         {
            a->bf = 0; unbalanced = false;
         }
         if(unbalanced)
         {
            if( d == +1)    //left imbalance
            {
               if(b->bf == +1) // LL rotation
               {
                  a->lchild = b->rchild; b->rchild = a;
                  a->bf = 0; b->bf = 0;
               }
               else
               {
                  c = b->rchild;
                  b->rchild = c->lchild;
                  a->lchild = c->rchild;
                  c->lchild = b;
                  c->rchild = a;
                  switch ( c->bf)
                  {
                     case +1:      //LR(b)
                        a->bf = -1; b->bf = 0;
                     case -1:   //LR(c)
                        b->bf = +1; a->bf = 0;
                     case 0:
                        b->bf = 0; a->bf = 0;
                  }
                  c->bf = 0; b = c; // b is the new root
               }
            }
            else
            {
               // code for right imbalance
            }
            // subtree with root has been rebalanced
            if(f = NULL)
               t = b;
            else
               if(a == f->lchild)
                  f->lchild = b;
               else if(a == f->rchild)
                  f->rchild = b;
         }     // end of if(unbalanced)
      }        // if(!found)
   } //end
```