8 Trees

At the end of this chapter you will be able to understand the concept of trees and their application as data structures.

SCOPE

8.1 Introduction

What do we see when we look at the hierarchical structure of an organization, or for that matter, a family tree or a table of contents. It looks somewhat as follows:

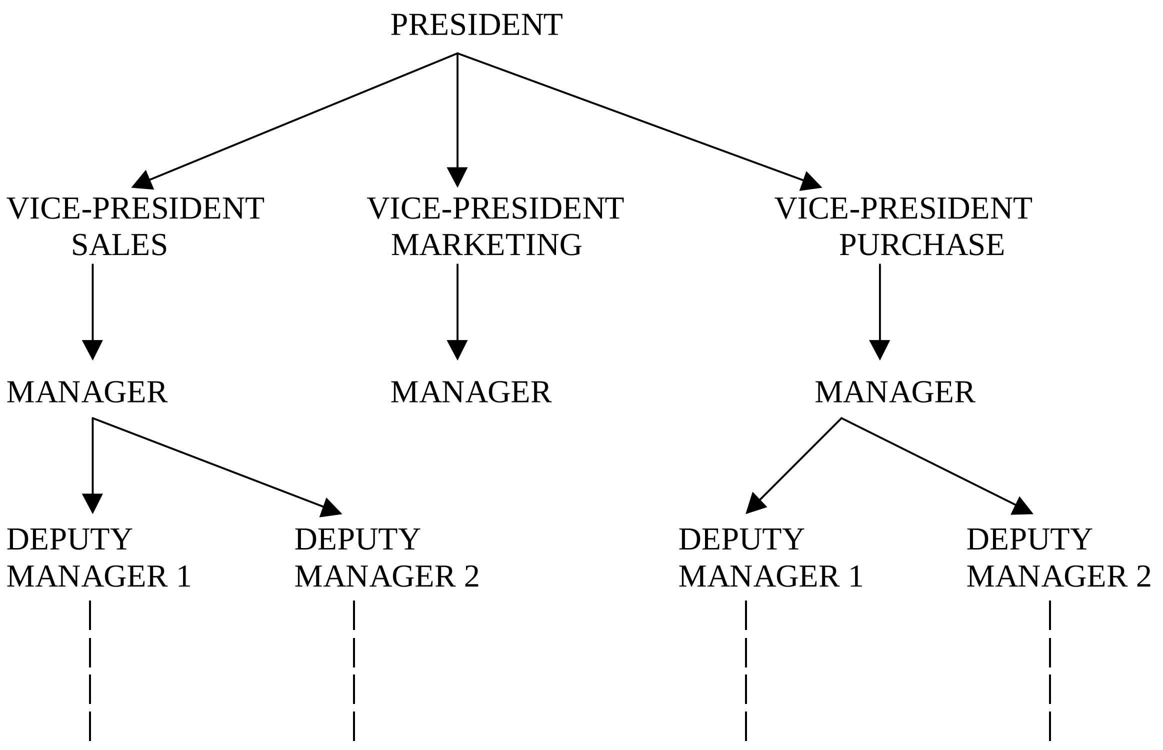


Fig. 8.1
Hierarchical structure of an organization

Can you site some familiarity in the above picture? Doesn't it look like a tree? Yes, it is a tree. But an inverted one. The President is the root. The arrows connecting President to the Vice-President's of Sales, Marketing and Purchase are branches of that tree, and so on. As you can see, a tree with its root, branches and leaves can be used to represent relationships in an organization or a family. The same concept of trees can be used in computing problems to depict relationship among number of records.

So far, we have come across mainly linear type of data structures: arrays, lists, stacks queues, etc. In this chapter, we introduce a non-linear data structure called a tree. This structure is used to represent data containing a hierarchical relationship between elements of a record.

Linked lists have advantages of flexibility over the contiguous representation of data, but they have one weakness. They are sequential lists, i.e., they are arranged in such a manner that to search for a key, one has to move through the list one node at a time, and therefore searching through a linked list is always a sequential search. As we all know very well, a sequential search is usually very slow in comparison to binary search. Let us do a revision:

1. The sequential search is of the order of `O(n)`.
2. The binary search is of the order of `O(logn)`.

What we need to do therefore is to find a method for rearranging the nodes of a linked list so that our search reduces to `O(log n)` instead of `O(n)`. Trees, implemented by using pointers and linked lists can do this.

Consider the following family tree.

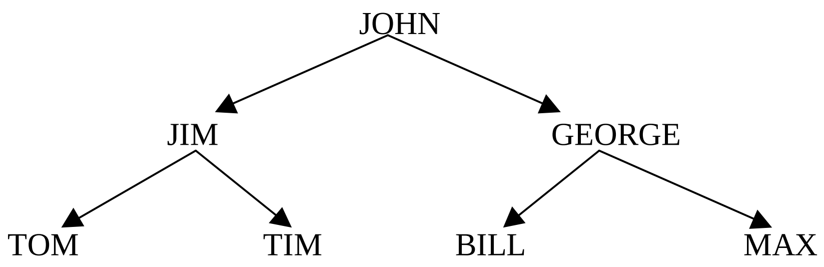


Fig. 8.2
A family tree

Let's get conversant with some terminologies. A tree consists of nodes, each node being named by the name of a person in the family tree. The tree originates from a unique node, named John. Such a node of a tree is called its root. Every node has a number of children nodes (may be zero). For example, in the above family tree, John has two children i.e. two nodes named Jim and George respectively. The nodes with no children like Tom are called leaves of the tree.

Armed with the above knowledge, let us define the tree formally. A tree can be formally defined as a finite set T of one or more nodes such that,

a) There is one specially designated node called the root of the tree; and
b) The remaining nodes (excluding the root) if any, are partitioned into m>0 disjoint sets T_1, T_2, ----, T_m, and each of this sets in turn is a tree. The trees T_1, T_2, ----, T_m are called subtrees of the root.

To understand the above definitions clearly, let us consider an example of a tree as given below:

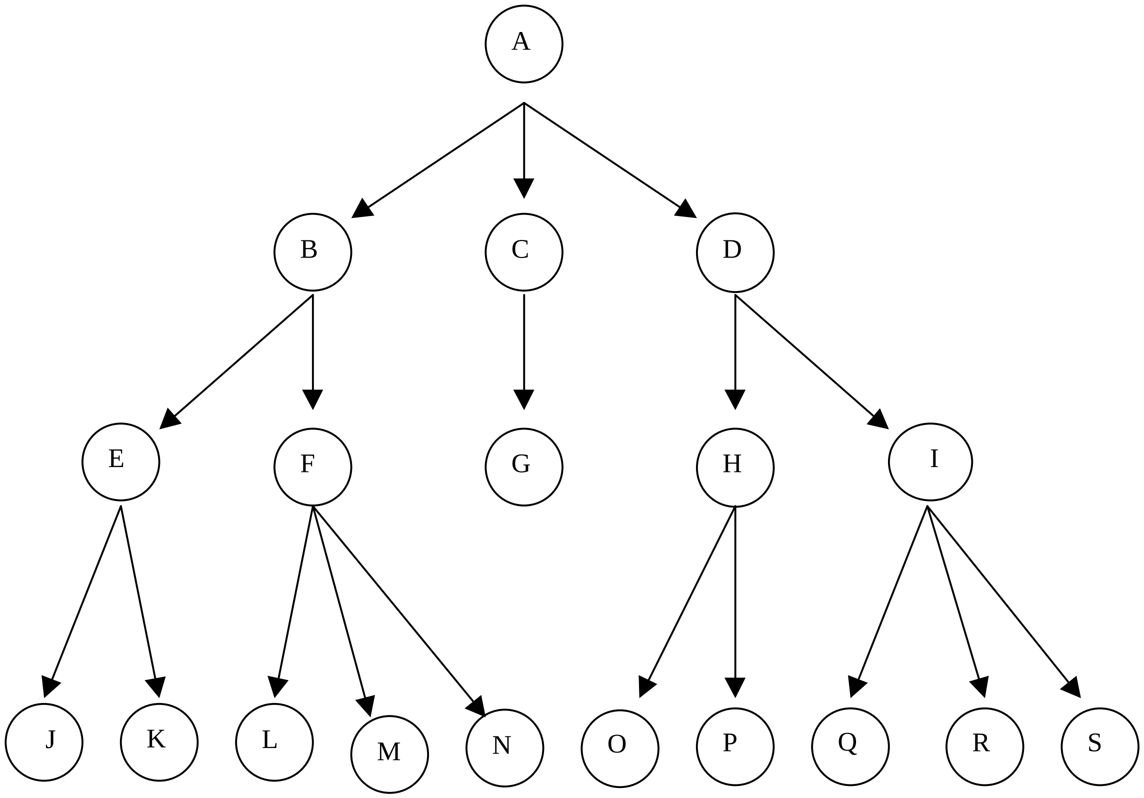


Fig. 8.3
An example of a tree

`A` is the root of the tree. The tree has 3 subtrees, whose roots are `B, C` and `D`. The subtrees are not connected together in any way because they have to be disjoint. Every item in a tree is the root of some subtree of the whole. For example, I is the root of a subtree of `D` which itself has 3 subtrees with roots `Q,R` and `S`.

Remember that a tree can be drawn in any direction, not necessarily with the root at the top and growing downwards. It is just a matter of convention to draw the root at the top and the tree as inverted.

One node is joined to another node by means of an edge or a branch. The number of nodes pointed to by a node by edges is called the degree of that node. For example, the degree of node `A` is 3, that of `B` is 2 and that of node L is just 0. A node of degree 0 is called a leaf or terminal node.

If a node n has a subtree with root m then m is called a child of n and n is the parent of m. In Fig. 8.3 `C` is the child of `A` and `A` is the parent of `C`.

The level of a node in a tree T is defined by saying that the root is at level zero, and other nodes have a level that is one higher than the level its parent has. For example, in the Fig. 8.3, level of `A` is 0, that of `B, C` and `D` is 1 and similarly level of `J, K` and `L` is 3.

We can observe that:

1. In a tree every node except the root has exactly one parent.
2. A tree with `n`-nodes has exactly `n-1` branches.

Two trees are said to be similar if they have the same structure i.e. same shape. The trees are said to be copies if they have the same contents at corresponding nodes. Consider the following figure:

$$\begin{tikzcd}
A \arrow[dr] & & \\
& B \arrow[dl] \arrow[dr] & \\
C & & D
\end{tikzcd}
\quad
\begin{tikzcd}
& & E \arrow[dl] \\
& F \arrow[dl] \arrow[dr] & \\
G & & H
\end{tikzcd}
\quad
\begin{tikzcd}
A \arrow[dr] & & \\
& B \arrow[dl] \arrow[dr] & \\
C & & D
\end{tikzcd}
\quad
\begin{tikzcd}
E \arrow[dr] & & \\
& F \arrow[dl] \arrow[dr] & \\
G & & H
\end{tikzcd}$$

(a) (b) (c) (d)

Fig. 8.4
Illustration of similar and copy trees

In Fig. 8.4, (a), (c) are (d) similar because they have the same structure. However, they are not copies because they don’t have same elements at corresponding nodes. Trees (a) and (c) are copies since they have same elements at corresponding nodes.

Line drawn from a node to its child is called an edge, and a sequence of consecutive edges is called a path. A path ending in a terminal node or a leaf is called a branch.

The depth (or height) of a tree is the maximum number of nodes in a branch of a tree. This is 1 more than the largest level number of T. In Fig. 8.3, height of the tree is 4.

8.2 Binary Trees

A binary tree T is a finite (may be empty) collection of elements. When the binary tree is not empty, it has a root element and the remaining elements (if any) are partitioned into 2 binary trees, which are called the right and left subtrees of T.

The difference between a binary tree and a tree are :

- A binary tree can be empty, whereas a tree cannot.
- Each element in a binary tree has exactly 2 subtrees (one or both of them may be empty). Each element in a tree can have any number of subtrees.

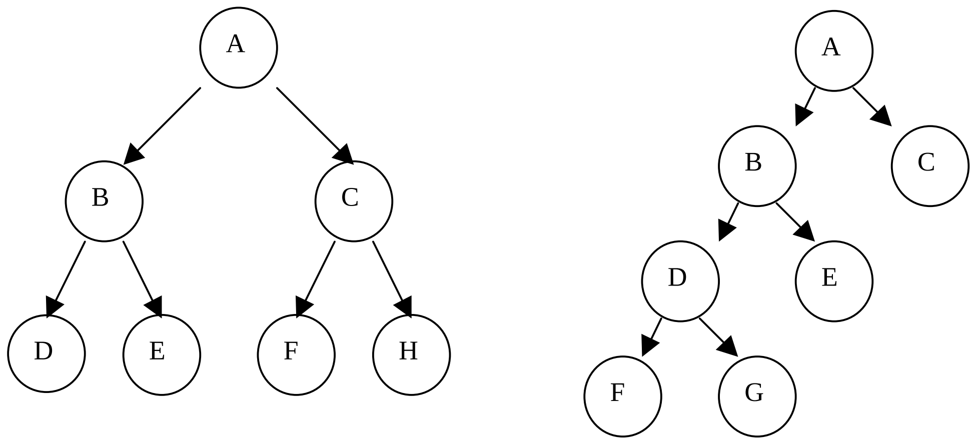


Fig. 8.5
Binary Trees structures

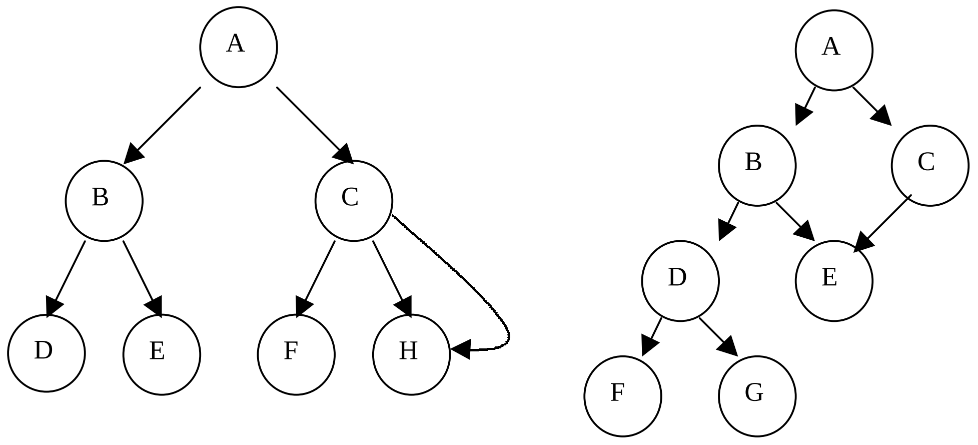


Fig. 8.6
Non Binary Trees structures

8.2.1 Properties of Binary Trees

1. Every Binary tree with `n` elements, where `n>0`, has exactly `n-1` edges.

2. A binary tree of height `h`, where `h>=0`, has at least `h` and at most `2h-1` elements in it.

3. The height of a binary tree that contains n elements, where `n>=0`, is at most n and at least `[log 2 (n+1)].`

A binary tree of height `h` that contains exactly `2h-1` elements is called a full binary tree. Consider the Fig. 8.7.

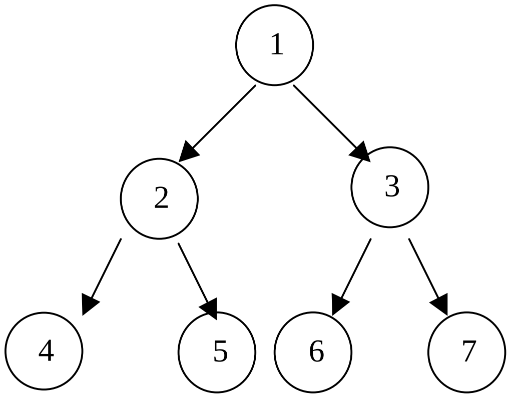


Fig. 8.7
Full Binary tree of height 3

A full binary tree is a special case of a complete binary tree.

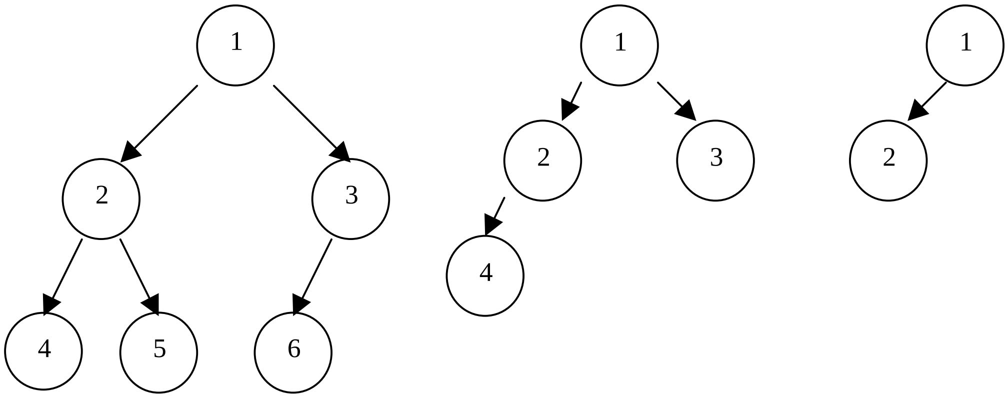


Fig 8.8
Complete binary trees

8.2.2 ADT of binary trees

Value Definition: The nodes of a tree can contain any value that they are defined to contain. The value can be a single unit of information or a complex record.

There are various operations that can be performed on a binary tree. Some basic sets of operations are as follows:

1. `create`

Function	: Initializes the tree to an empty state.
Precondition	: None.
Postcondition	: An empty tree is created.

2. `destroy`

Function	: Destroys all elements, leaving an empty tree.
Precondition	: Tree exists.
Postcondition	: Tree is empty.

3. `retrieve`

Function	: Searches the tree for an element and return a copy.
Precondition	: Tree exists.
Postcondition	: Answer as ‘yes’ or ‘no’; Original tree remains unchanged.

4. `insert`

Function	: Adds an element to the binary tree.
Precondition	: Tree exists and does not contain a node with the same key value as element.
Postcondition	: Original tree with new element added.

5. `modify`

Function	: Replaces existing tree element with new.
Precondition	: Tree exists.
Postcondition	: Original tree with replaced element.

6. `delete`

Function	: Delete an element from the tree.
Precondition	: Tree exists.
Postcondition	: Original tree with an element deleted.

```
7. traversal
```

Function	: Visiting each node of the tree as per the traversal order.
Precondition	: Tree exists.
Postcondition	: Original tree remains unchanged; elements are given as output.

Traversal Techniques

Traversal means, passing through the tree, visiting each of its nodes once. We may simply like to print the contents of each node as we visit it or we may like to process it in some other way.

We know from our experience that nodes of a linear list are visited in the order of first to last. But, the basic structure of a tree is different than that of a linear list. There is no such ‘defined’ order for the nodes of a tree. Different cases may involve different orderings for traversal. There are 3 basic ways of traversal, as follows

1. Preorder.
2. Inorder.
3. Postorder.

In all the above 3 methods, nothing needs to be done to traverse an empty binary tree. Traversing a binary tree involves visiting the root and traversing its left and right subtrees. The only difference among the methods is the order in which these three operations are performed.

To traverse a binary tree in preorder, we perform the following 3 operations:

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

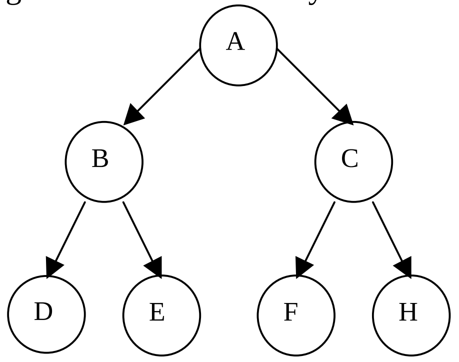
To traverse a binary tree in inorder, we perform the following 3 operations:

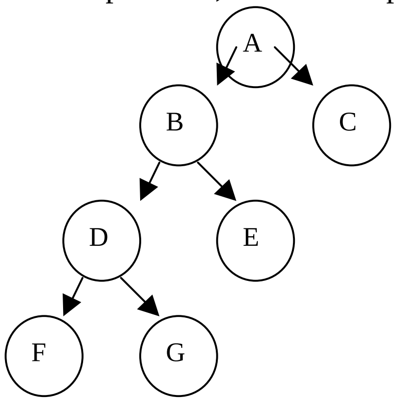
1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

To traverse a binary tree in postorder, we perform the following 3 operations:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Fig. 8.9 illustrates 2 binary trees and their traversals in preorder, inorder and post order.





Preorder	: ABDECFH	Preorder	: ABDFGEC
Inorder	: DBEAFCH	Inorder	: FDGBEAC
Postorder	: DEBFHCA	Postorder	: FGDEBCA

Fig. 8.9
Binary trees and their traversals

We will try to implement these traversal methods in our next section.

8.3 Binary Search Trees

A binary search tree is a binary tree that is either empty or in which each node contains a key that satisfies the conditions:

1. All keys (if any) in the left subtree of the root precede the key in the root.
2. The key in the root precedes all keys (if any) in its right subtree.
3. The left and right subtrees of the root are again search trees.

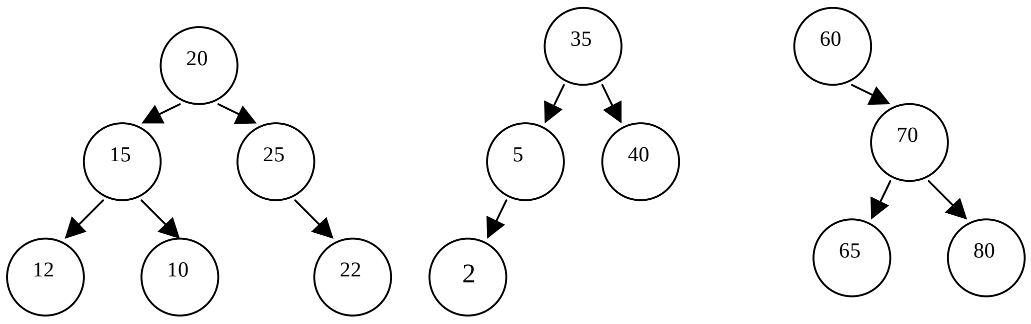


Fig. 8.10
Binary Search Trees.

8.3.1 Implementation

After getting the basic terminology and concepts about trees clear, let us try to implement trees. Trees can be implemented both as an array structure as well as a linked structure. Recall our discussion on relative merits and demerits of these two implementations. We will start with linked version of the trees. Array structure that is very useful in some situations will be studied later.

8.3.1.1 Linked Implementation.

In a linked list format, a tree structure can be defined as follows.

```
struct node
{
      int info;
      struct node *left_child;
      struct node *right_child;
}
```

where,

`info` is the information part of the node,

`*left_child` is a pointer to left child of the node, and

`*right_child` is a pointer to right child of the node.

Therefore, a node can be pictorially represented as follows:

node

*left_child	info	*right_child

Now, let us try to create a tree. Remember, initially root points to NULL, i.e. tree is empty.

```
      struct node *root = NULL
```

Before starting to write the actual code for the program, let us go through an algorithm for the program.

Algorithm

Step 1. Start.

Step 2. Declare the node structure.

Step 3. Accept the new element to be placed in the tree.

Step 4. If root points to NULL, then make the new element the root node.

Step 5. If the new value is smaller than the current node, proceed with the left side of the tree.

Step 6. If the left side is empty, make the new node the left child of the current node.

Step 7. If the new value is bigger than the current node, proceed with the right side of the tree.

Step 8. If the right side is empty, make the new node the right child of the current node.

Step 9. If the new value is equal to the current node, it means that the new value is a duplicate. This is not allowed, therefore display an appropriate message.

Step 10. If the left and right sides are not empty, continue from step 5.

Step 11. Print the tree in inorder.

Step 12. Stop.

Program: To create a tree, and print it in inorder format.

```
/*This program creates a tree and if it exists, inserts a new element at
a new place*/


#include<stdio.h>
#include<stdlib.h>

struct link_list                    /* structure of a node */
{
      int info;
      struct link_list *left_child;
      struct link_list *right_child;
};

typedef struct link_list node;

main()                              /* main starts here */
{

      node *root;             /* pointer to the root node */
      node *temp;             /* temporary node */
      node *p;                      /* traversing pointer */

      node *pre;        // pointer to keep track of previous //
                              // location of p //
      int flag=1;
      root = NULL;


      while(flag==1)
      {
```

```
system("clear");
temp = (node *)malloc(sizeof(node));

printf("\nenter the element:");     //accepting the//
                                    //value for the //
scanf("%d",&temp->info);            //info part//
fflush(stdin);

temp->left_child = NULL;
temp->right_child = NULL;

if( root==NULL)               //if it is the first//
                              //node//
{
   root = temp;
}
else                          //else look for a proper//
                              //place//
{
p = root;
while( p!=NULL)
{
   if ( temp->info < p->info) //if new value is//
                              //smaller than//
   {                          //the node value//
      pre = p;                //proceed with left half//
                              //of the tree//
      p = p->left_child;
   }
   else if (temp->info > p->info )  //if new value is//
                                   //bigger than//
   {                                //the node value//
      pre = p;                     //proceed with right//
                                  //half of the tree//
      p = p->right_child;
   }
   else                        //if new value is equal//
                              //to the //
   {                         //node value//
      printf("\n element already present.");
      break;

   }
}


if ( temp->info < pre->info )  //if new value is//
                              //smaller than//
                              //the node value//
{
   pre->left_child = temp;    //make new node the//
                                    //left child//
}
else if ( temp->info > pre->info )  //if new value is//
                              //bigger than//
                              //the node value//
{
   pre->right_child = temp;   //make new node the//
                              //right child//


}
}
printf("\n");
```

```
            print(root);                            //calling print//
                                                    //function//

            printf("\ndo you want to continue(yes=1/no=0):");
            scanf("%d",&flag);
            fflush(stdin);

            }

}                             /* main ends here */

print( node * q)              /* printing the tree in inorder */
{

      if( q!=NULL)
      {

            print(q->left_child);         /*print the left child*/
            printf("%d ",q->info);        /*print the node*/
            print(q->right_child);        /*print the right child*/

      }

}
```

In the above program, we accept items and depending on whether they are smaller or greater than the root, we keep inserting them as its left or right child. While printing the tree, we follow inorder traversal. Reader can make the tree on paper and compare the two. The same tree could have been printed in preorder or postorder format.

Printing the tree in a particular order, namely, inorder, preorder and postorder, is just a question of executing which statement in what sequence.

The print routine in the above program can be rewritten as follows:

For preorder traversal :

```
print( node * q)              /* printing the tree in preorder */
{
      if( q!=NULL)
      {
            printf("%d ",q->info);
            print(q->left_child);
            print(q->right_child);
      }
}
```

For postorder traversal:

```
print( node * q)              /* printing the tree in postorder */
{

      if( q!=NULL)
      {

            print(q->left_child);
            print(q->right_child);
printf("%d ",q->info);

      }

}
```

Searching for an element in the tree

Searching in a tree follows the same methodology of comparing the key with the root everytime, until the element is found. If the key is smaller, search proceeds with the left half of the tree. If the key is greater, search proceeds with the right half of the tree. Following program illustrates searching in a tree.

Program: To search for an element in the tree.

```
/*This program creates a tree and if it exists, searches for an
element*/

#include<stdio.h>

#include<stdlib.h>

struct link_list              /* structure of a node */
{
      int info;

      struct link_list *left_child;
      struct link_list *right_child;
};
```

```
typedef struct link_list node;

main()                        /*main starts here*/
{

      node *root;             /* pointer to the root node */
      node *temp;             /* temporary node */
      node *p;                      /* traversing pointer */
      node *pre;              //pointer to keep track of previous //
                              // location of p //

      int flag=1;

      root = NULL;

      while(flag!=4)
      {
            printf("\n1. Insert”);
            printf(“\n2. Search”);
            printf(“\n3. Print”);
            printf(“\n4. Exit");

            scanf("%d",&flag);
            switch(flag)
            {
              case 1:                 /*if the choice is insert*/
                  system("clear");
                  temp = (node *)malloc(sizeof(node));

            printf("\nenter the element:");  //accept the new//
                                             //element//
            scanf("%d",&temp->info);
            fflush(stdin);


            temp->left_child = NULL;
            temp->right_child = NULL;

            if( root==NULL)         /*if it is the first node*/
            {
               root = temp;
            }
            else              //look for a proper place to insert //
            {                       //in the usual manner//
            p = root;

            while( p!=NULL)
            {
               if ( temp->info < p->info)
               {

                  pre = p;
                  p = p->left_child;
               }

               else if (temp->info > p->info )
               {
                  pre = p;
                  p = p->right_child;
               }
               else
               {
                  printf("\n element already present.");
```

```
                    break;

                 }
              }

              if ( temp->info < pre->info )
              {
                 pre->left_child = temp;
              }
              else if ( temp->info > pre->info )
              {
                 pre->right_child = temp;
              }
               }
               break;

               case 2:                /*if the choice is search*/
                     search(root);    /*calling the search function*/
                           break;
               case 3:
                    print(root);      /* calling print function */
                          break;
                 case 4:
                    break;
               }                            /*end of switch */


              }

        }                                 /*end of main*/

        print( node * q)                  /*printing the tree in inorder*/
        {

              if( q!=NULL)
              {

                    print(q->left_child);
                    printf("%d ",q->info);

                    print(q->right_child);
              }

        }


        search( node * r )                /*search function starts here*/
        {
              node *pre;
              node *temp;
              int item;

              printf("\nenter the element to be searched:");
              scanf("%d",&item);      /*accept the element to be searched*/
              fflush(stdin);



              while( r!=NULL && r->info!=item)
              {                       //traverse through the tree in the//
                                            //usual manner//
                 if ( item < r->info)
                 {
                    pre = r;
```

```
            r = r->left_child;
         }
         else if (item > r->info )
         {
            pre = r;
            r = r->right_child;
         }
      }

      if ( r==NULL )          //if the element is not present//
                              //in the tree//
      {
         printf("\n element not present.");
      }

      if (r->info==item)            /*if the element is present*/
      {
         printf("\nelement present .");
      }
}                                   /*search function ends here*/
```

Deleting an element from a tree:

Deleting an element from a tree is not as simple as inserting or searching for an element in the tree. We will see why it is so. Deletion requires two operations to be performed:

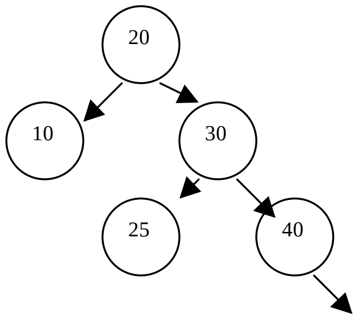
1. Find the node in the tree that contains the element.
2. Delete that node.

First part is easy, as we have implemented it earlier. The second part of the operation – deletion of the node from the tree is more complicated. This task varies according to the position of the node in the tree. It is easy to delete a leaf node than to delete any other node. We can break down this problem into 3 cases, depending on the number of children linked to the node we want to delete.

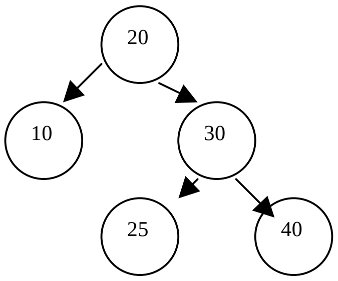
1. Deleting a leaf node (no children)

As shown in the following figure, deleting a leaf node is simply a matter of setting the appropriate link of its parent to NULL and then disposing off the unnecessary node.

Before deletion



After deletion



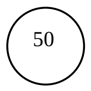


Fig. 8.11
Deleting the node containing 50

2. Deleting a node with only one child

Here, we don't want to loose the descendents of the node. So, we make the pointer from the parent skip over the deleted node and point instead to the child of the node we want to delete. We can then dispose the unwanted node.

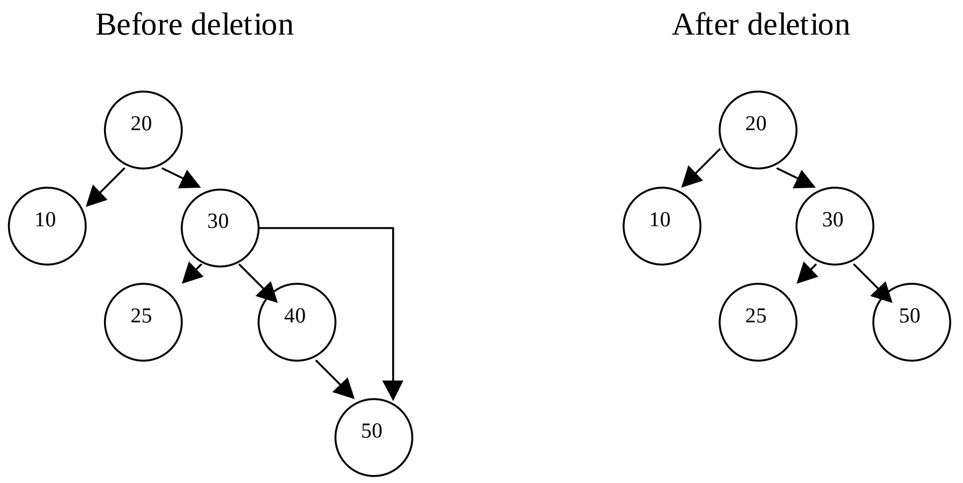
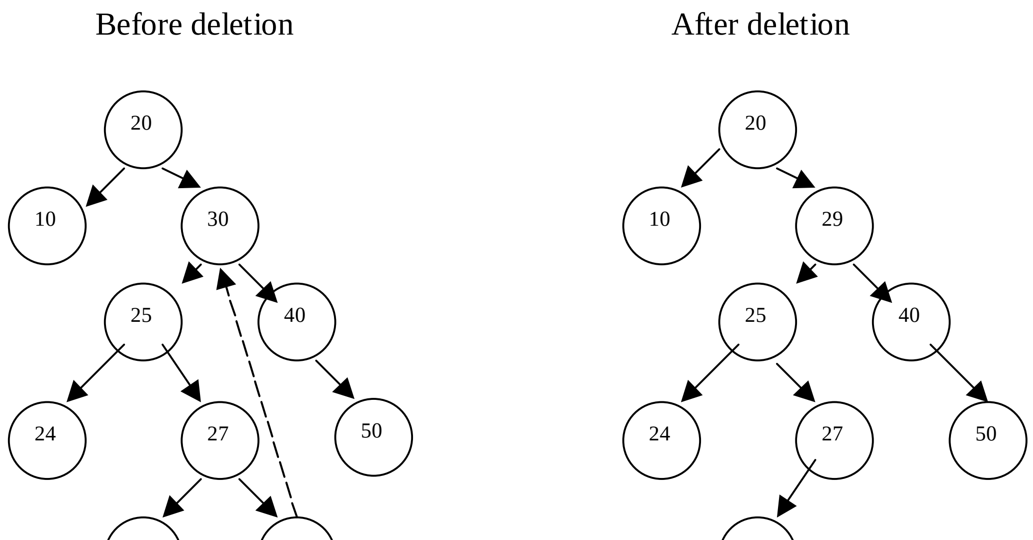


Fig. 8.12
Deleting the node containing 40

3. Deleting a node with two children

This case is the most complicated because we cannot make the parent of the deleted node point to both of the deleted node's children. We will replace the info part of the node we want to delete with the info part of its closest lesser predecessor. We can then delete the replacement node by changing one of its parent's pointers.



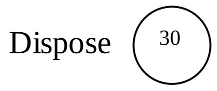


Fig. 8.13
Deleting the node containing 30.

Before writing the actual code, let us go through the algorithm for the above activity.

Algorithm

Step 1. Start.

Step 2. Declare the node structure.

Step 3. Ask for user's choice, i.e. whether he wants to insert, delete or print the tree.

Step 4. If the choice is insert, look for a proper place to insert in the usual manner.

Step 5. If the choice is delete,
1. Accept the element to be deleted.
2. Search for the element in the tree in the usual manner.
3. If the element is not found, give an appropriate message.
4. If the element is found, proceed as follows:

   (a) If the element is the last node, make the right and left children of the previous node as NULL.

   (b) If the element has only a left child,
      1. If the node is the right child of the previous node, make the left child, right child of the previous node.
      2. If the node is the left child of the previous node, make the left child, left child of the previous node.

   (c) If the element has only a right child,
      1. If the node is the right child of the previous node, make the right child, right child of the previous node.
      2. If the node is the left child of the previous node, make the right child, left child of the previous node.

   (d) If the element has both the left and right children,
      1. Proceed with the left child.
      2. Find the right most child of the left node.

3. Place it in place of element to be deleted.

(e) Release the space that is not required now.

Step 6. If the choice is print, print the tree in inorder.

Step 7. Stop.

Program: To delete an element from a tree.

```
/*This program creates a tree and if it exists, deletes an element*/

#include<stdio.h>
#include<stdlib.h>

struct link_list              /* structure of a node */
{
      int info;
      struct link_list *left_child;
      struct link_list *right_child;
};

typedef struct link_list node;

main()                              /* main starts here */
{

      node *root;               /* pointer to the root node */
      node *temp;               /* temporary node */
      node *p;                        /* traversing pointer */
      node *pre;               /* pointer to keep track of previous */
                              /* location of p */

      int flag=1;

      root = NULL;


      while(flag!=4)
      {
            printf("\n1. Insert”);
            printf(“\n2. Delete”);
            printf(“\n3. Print”);
            printf(“\n4. Exit");

            scanf("%d",&flag);
            switch(flag)
            {
              case 1:               /*if the choice is insert*/
                  system("clear");
                  temp = (node *)malloc(sizeof(node));

            printf("\nenter the element:");     //accept the new//
                                                //element//
            scanf("%d",&temp->info);
            fflush(stdin);

            temp->left_child = NULL;
            temp->right_child = NULL;
            if( root==NULL)         /*if it is the first node*/
            {

               root = temp;
            }
            else              //look for a proper place to insert //
            {                       //in the usual manner//
            p = root;
            while( p!=NULL)
            {
               if ( temp->info < p->info)
               {
                  pre = p;
                  p = p->left_child;
               }

               else if (temp->info > p->info )
               {
                  pre = p;
                  p = p->right_child;
               }
               else
               {
                  printf("\n element already present.");
                  break;

               }
            }


            if ( temp->info < pre->info )
            {
               pre->left_child = temp;
            }
            else if ( temp->info > pre->info )
            {
               pre->right_child = temp;
            }
             }
             break;

             case 2: delete(root);  //calling delete//
                                    //function//
                         break;

             case 3:
                  print(root);      /* calling print function */
                        break;

                  case 4:    break;
             }                            /*end of switch */




            }

}                             /* main ends here */


print( node * q)              /* printing the tree in inorder */
{

      if( q!=NULL)
      {

            print(q->left_child);
            printf("%d ",q->info);
            print(q->right_child);
      }

}


delete( node * r )            /* module for deleting the node */
{
      node *pre;
      node *temp;
      int item;

      printf("\nenter the element to be deleted:");
      scanf("%d",&item);      /*accept the element to be deleted*/
      fflush(stdin);


      while( r!=NULL && r->info!=item)
      {                            //search for the element in the//
                                  //existing //
                                 //tree in the usual manner//
         if ( item < r->info)
         {
            pre = r;
            r = r->left_child;
         }
         else if (item > r->info )
         {
            pre = r;
            r = r->right_child;
         }
      }

      if ( r==NULL )          /*if the element is not present*/
      {
         printf("\n element not present.");
      }

      if (r->info==item)            /*if the element is present*/
      {
         printf("\nelement present and now getting deleted.");


      /* If node is the last node in any branch */

            if ( r->right_child==NULL && r->left_child==NULL )

              {

               if ( pre->right_child==r )
               {

                  pre->right_child = NULL;
                  free(r);

               }
               else if ( pre->left_child==r )
               {

                  pre->left_child = NULL;
                  free(r);

               }

            }

      /* If node has only left child */

      if( r->left_child!=NULL && r->right_child==NULL )
      {

         if ( pre->right_child==r )
         {
            pre->right_child = r->left_child;
            free(r);
         }
         else if ( pre->left_child==r )
         {
            pre->left_child = r->left_child;
            free(r);
         }

      }

      /* If node has only right child */

      if( r->left_child==NULL && r->right_child!=NULL )
      {

         if ( pre->right_child==r )
         {
            pre->right_child = r->right_child;
            free(r);
         }
         else if ( pre->left_child==r )
         {
            pre->left_child = r->right_child;
            free(r);
         }

      }


      /* If node has both left and right children */


      if( r->left_child!=NULL && r->right_child!=NULL )
      {
         temp = r->left_child;
         pre = r;
         while ( temp->right_child!=NULL )
         {
            pre = temp;
            temp = temp->right_child;
         }

         r->info = temp->info;
         if (pre == r)
            pre ->left_child = temp->left_child;
           else
            pre -> right_child = temp->left_child;

         free(temp);
    }
```

```
}
}
```

8.3.1.2 Array Implementation

In the linked implementation, the pointers from parent to children are explicit in the data structure. A field is declared in each node for the pointer to the left child and a pointer to the right child.

A binary tree can be stored in an array in such a way that the parent-child relationship is maintained. What we will do, is to store the tree in the array, level by level, left to right as shown in the following figure. The number of nodes in the tree is MAX (size of array). The first node, i.e. the root is stored in T[0] and the last node in T[MAX-1].

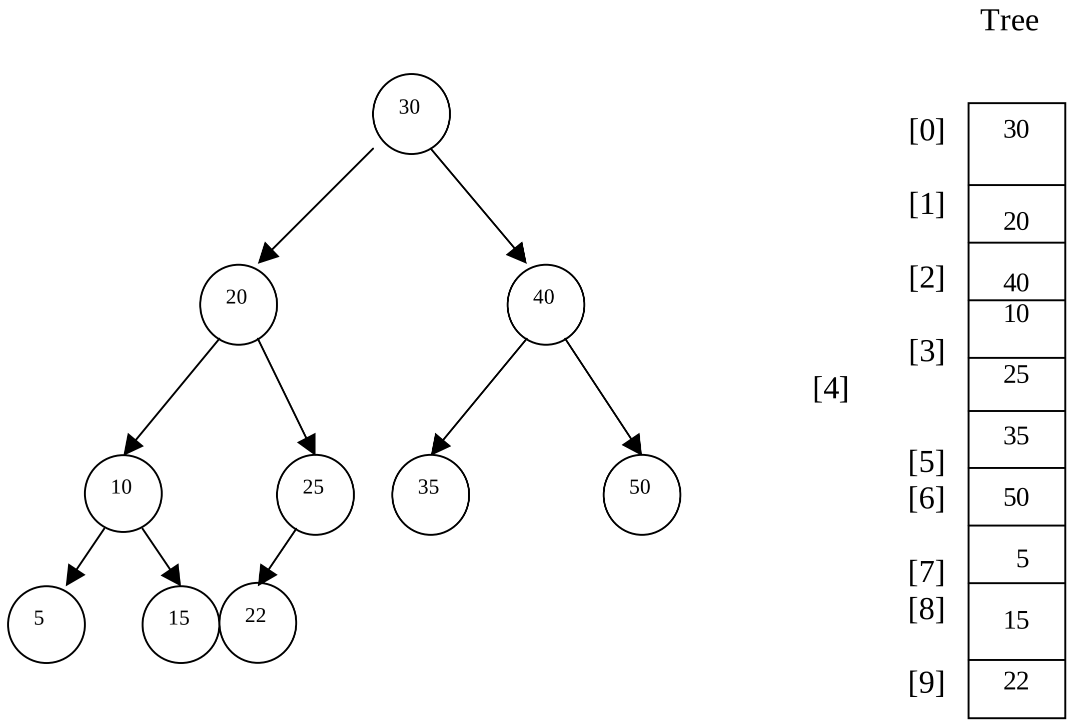


Fig. 8.14
Array representation of a binary tree

To implement the algorithm to manipulate the tree, we must be able to find the left and right child of a node in the tree. Comparing the tree and the array in the figure, we can observe following facts:

T[0]'s children are in T[1] and T[2],
T[1]'s children are in T[3] and T[4],
T[2]'s children are in T[5] and T[6],
And so on.

We can see that there is a pattern:

For any node `T[index]`, its left child is in `T[2*index + 1]` and right child is in `T[2*index + 2]`.

Also the nodes in the array from `T[MAX / 2]` to `T [MAX - 1]` are leaf nodes.

Not only can we easily calculate the location of a child, we can also find the parent of a particular node. We know that this is not so simple in the linked implementation but is very easy in an array structure.

You can observe from the figure that:

- If index is odd, then the parent of `T[index]` is `T[index / 2]`.
- If index is even, then the parent of `T[index]` is `T[index / 2 – 1]`.

This shows that the array implementation of a binary tree is linked in both directions: from parent to child and from child to parent. We will see that this is of tremendous use in later sections.

This tree implementation works fine for a full or complete binary tree. Let's review our concepts of full and complete binary trees.

A full binary tree is a binary tree in which all the leaves are on the same level and every non-leaf node has 2 children. See the figure below:

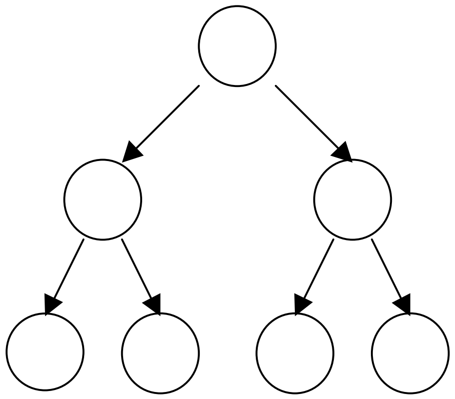
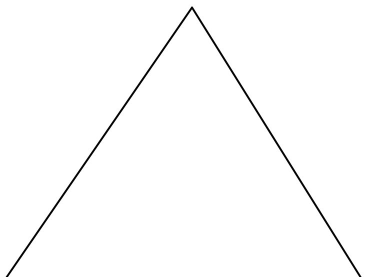


Fig. 8.15
A full binary tree.

If you try to enclose this figure by drawing straight lines around it, you can see that you get a figure of triangle.



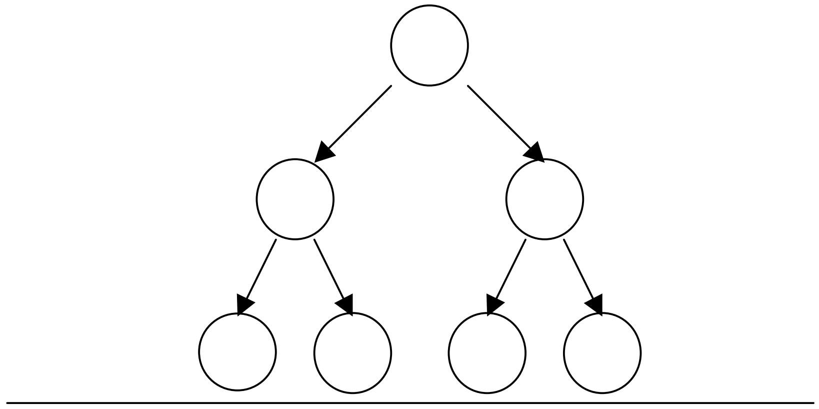


Fig. 8.16
A full binary tree as a triangle

A complete binary tree is a binary tree that is either full or the leaves on the last level. Therefore, the shape of a complete binary tree is either triangular (if the tree is full) as shown above or like the following.

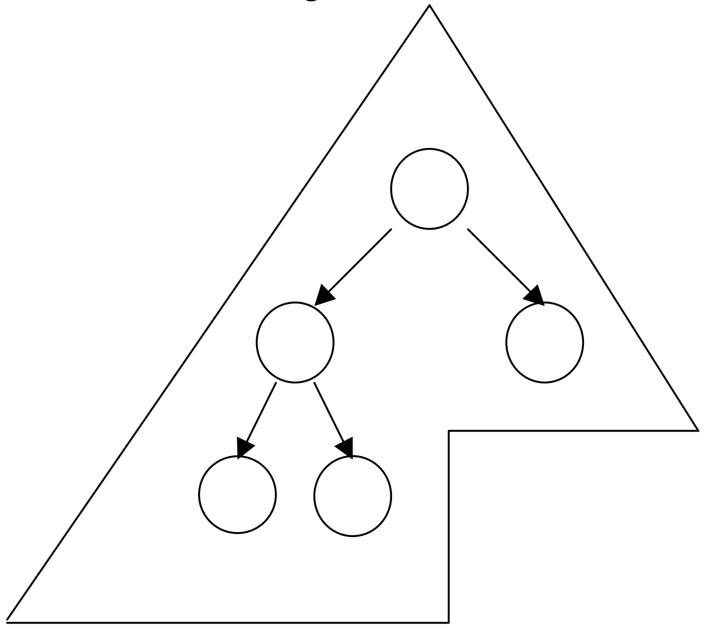
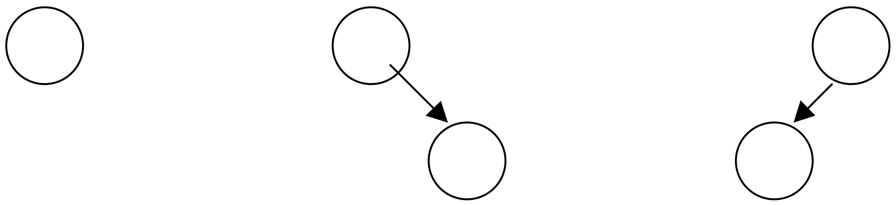
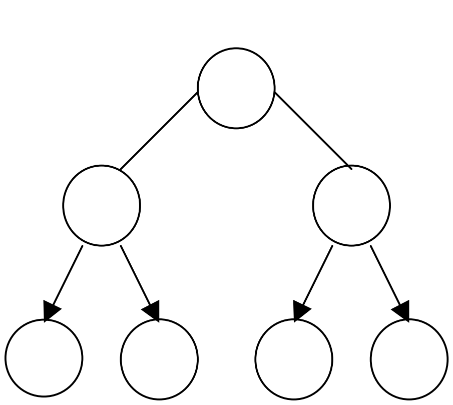
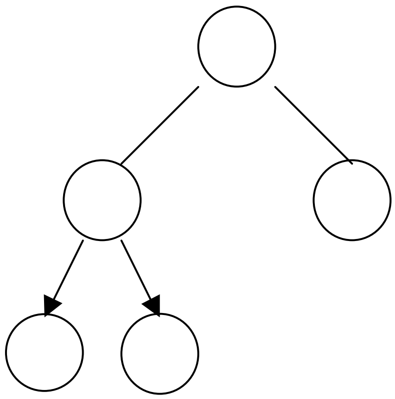


Fig. 8.17
A complete binary tree.

The Fig. 8.18 provides more illustration about complete and full binary trees.



(a) full and complete (b) neither full nor complete (c) complete

(d) full and complete (e) complete

Fig 8.18
Illustration of full and complete binary trees

The array-based implementation is simple for trees that are full or complete because the elements occupy contiguous array slots.

8.4 Heap

A heap is a special case of a binary tree that satisfies 2 properties,

1. shape property,
2. order property.

The shape property states that the heap must be a complete binary tree.

The order property states that, for every node in the heap, the value stored in that node is greater than or at least equal to that stored in its children. See the following diagram of a heap:

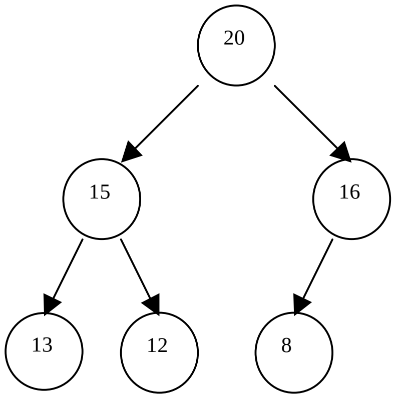


Fig. 8.19
A heap

Because of the order property, the root node will always contain the largest value in the heap. This is a feature of heap. We always know where the maximum value is. It is always in the root.

Now, let us try to define some operations on the heap. Suppose, we want to remove the largest element of the heap. The largest element is in the root, so we can easily remove it. But, this leaves a hole in the root position. To keep the tree complete, we fill the hole with the bottom rightmost element from the heap, so that the structure satisfies the shape property. But, since the new value has come from the bottom of the heap, the heap no longer satisfies the order property.

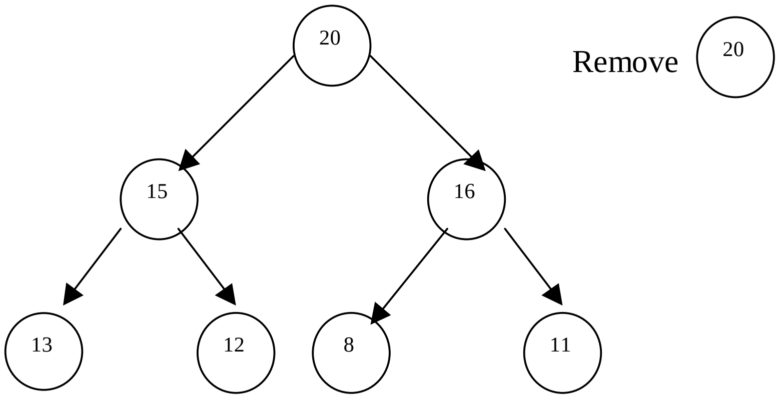


Fig. 8.20
Original heap

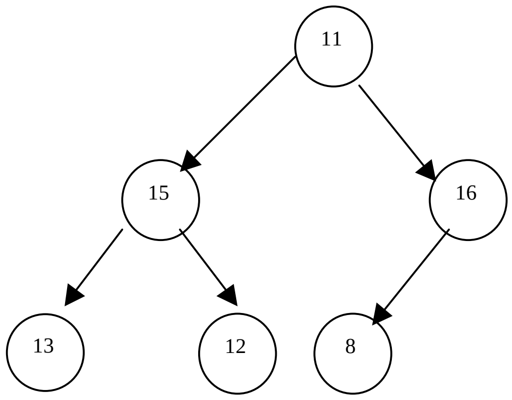
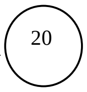
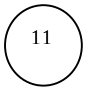


Fig. 8.21

Removed  **and replaced by** 

We need to restructure the tree to make it a heap. This involves moving an element down until the heap satisfies the order property also. This is called a **heapdown operation.**

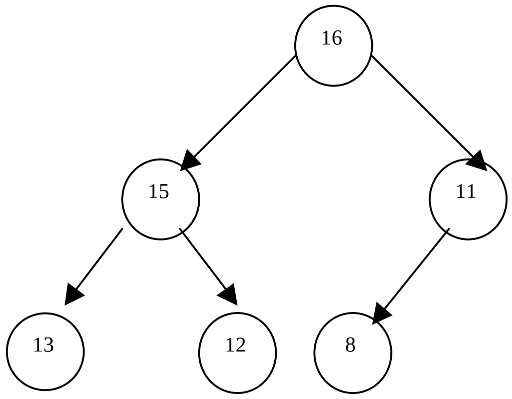
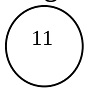


Fig.8.22

Pushing  **down to satisfy order property.**

If we want to add a new element in the heap, we can do so only at the rightmost bottom node, to maintain the shape property. See the adjoining figure.

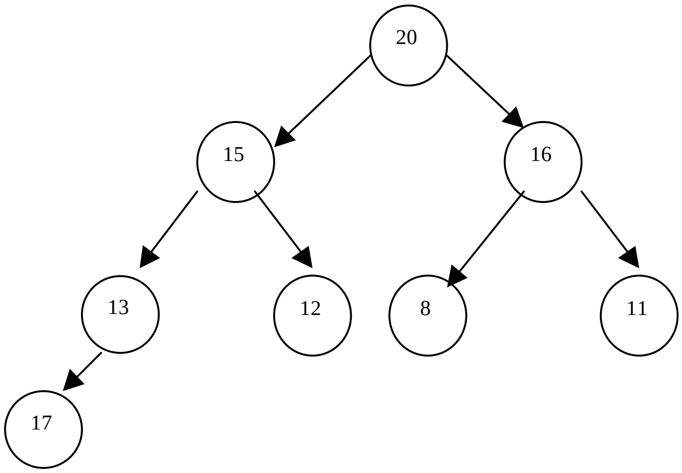


Fig 8.23
Original heap $\textcircled{17}$ **added**

But, this may violate the order property. To repair this situation, we need to float the element up the tree, until it is in its correct place. This is called a **heapup operation**.

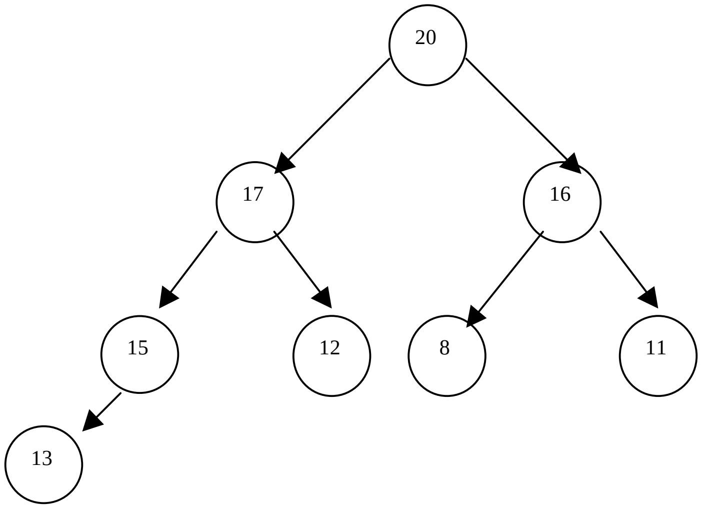


Fig.8.24
Floating $\textcircled{17}$ **up to satisfy order property.**

We can try a program to implement all the above operations on trees. But before that we will write an algorithm.

Algorithm

Step 1. Start.

Step 2. Declare an array for the heap.

Step 3. Ask for user's choice, i.e. whether he wants to find the maximum element, delete the maximum element, add a new element, print the heap or exit the program.

Step 4. If the choice is to find the maximum element,
1. Check if the heap is empty.
2. If not, then print the first element of the heap.

Step 5. If the choice is to delete the maximum element,
1. Check if the heap is empty.
2. If not,
   (a) Delete the top element.
   (b) Place the lower rightmost element of the heap at the top to satisfy shape property.
   (c) Perform heapdown operation to satisfy order property.

Step 6. If the choice is to add an element,
1. Check if there is space in the heap.
2. If there is, accept the new element.
3. Place it at the lower rightmost location of the heap to satisfy shape property.
4. Perform heapup operation to satisfy the order property.

Step 7. If the choice is print, print the heap.

Step 8. Stop.

Program: To implement heap operations.

```
/*This program implements a heap*/

#include<stdio.h>
#include<stdlib.h>

#define MAX 10                /* defining array size */

int array[MAX];

main()                              /* main starts here */
{
      int i;
      int flag;

      int input = -1;
      for(i=0;i<MAX;i++)   /* initializing array elements as zero */
      {
         array[i]=0;
      }
      while( flag!=5 )
      {
         system("clear");
         printf(" Chart.");
         printf("\n1. Find MAX.");
         printf("\n2. Delete MAX.");
         printf("\n3. Add.");
         printf("\n4. Print.");
         printf("\n5. Exit.");

         printf("\n Choice:");

         scanf("%d",&flag);
         fflush(stdin);

         switch(flag)
         {

            case 1: findmax();      /*calling findmax function*/
                  break;

            case 2: delete();       /*calling delete function*/
                  break;

            case 3: add();                /*calling add function*/
                  break;

            case 4: print();              /*calling print function*/
                  break;

            case 5: break;


         }                    /* switch ends here */

      }                       /* while ends here */


}                             /* main ends here */


print()                             /*print function starts here*/
{
      int i;
      if ( array[0]==0 )                        /*checking for empty heap*/
      {
         printf("\n heap is empty. choose add option to add.");
      }

      else                    /*if the heap is not empty*/

      {
      printf("\nprinting the heap....");
      for(i=0;i<MAX;i++)
      {
         printf("\n%d",array[i]);
      }
      }
      getchar();

}                             /*print function ends here*/
```

```
findmax()               /* module for finding the maximum element */
{
      if ( array[0]==0 )                   /*if the heap is empty*/
      {
         printf("\n heap is empty. choose add option to add.");
      }
      else                                            /*if the heap is
not empty*/
      {
         printf("\n the maximum element is:");
         printf("%d",array[0]);    //printing the maximum element,//
                                  //i.e. the root//
      }
      getchar();
}

delete()               /* module for deleting the maximum element */
{
      int i;

      if ( array[0]==0 )            /*if the heap is empty*/
      {
         printf("\n heap is empty. choose add option to add.");
      }
      else                          /*if the heap is not empty*/
      {
      printf("\nthe element deleted is:");
      printf("\n%d",array[0]);      //printing the maximum element//
                                    //to be deleted//

      i=1;
      while( array[i]!=0 )
      {
         i++;
      }
      array[0] = array[i-1];        //place the lower rightmost//
                                    //element at the top//
      array[i-1] = 0;

      print();                      /*printing the disordered heap*/

      printf("\nperforming heapdowning operation.\n");

      heapdown();             /*calling heapdown operation*/
      print();                      /*printing the ordered heap*/
      }
      getchar();
}                             /*delete function ends here*/

add()                         /* module for adding a new element */
{

      int i=0;
      int new;

      while(array[i]!=0)      //looking for the first empty //
                              //position//
      {
         i++;
      }

      if(i==MAX)              /*if the heap is full*/
      {
         printf("\nsorry. no space for more fruits in the tree.");
         getchar();
      }

      else                    /*if empty location is found*/
      {
            printf("\n enter the element to be added:");
            scanf("%d",&new);  /*accept the element to be added*/
            fflush(stdin);

              array[i] = new;   /*place it in the empty location*/

            print();                /*printing the disordered tree*/

            printf("\nperforming heapup operation.");

            heapup(i);        /*calling heapup function*/

            print();                /*printing the ordered tree*/

      }


}                             /*add function ends here*/


heapdown()                    /* module for heapdown operation */
{

      int i=0;
      int left_diff;          //difference between a node and its//
                              //left child //
      int right_diff;         // difference between a node and its//
                              //right child //
      int temp;


      left_diff = ( array[i] - array[2*i + 1]);
      right_diff = ( array[i] - array[2*i + 2]);

      if(left_diff > 0 && right_diff > 0) //looking for a proper//
                                         //place for the new//
      {                             //top element of the heap//
            /*do nothing*/
      }

      if(left_diff > 0 && right_diff < 0)
      {

         temp = array[2*i + 2];
         array[2*i + 2] = array[i];
         array[i] = temp;
      }


      if(left_diff < 0 && right_diff > 0)
      {

         temp = array[2*i + 1];
         array[2*i + 1] = array[i];
         array[i] = temp;
      }

      if(left_diff < 0 && right_diff < 0)
      {
        if(( left_diff - right_diff) > 0)
        {

         temp = array[2*i + 2];
         array[2*i + 2] = array[i];
         array[i] = temp;
        }
        else if((left_diff - right_diff) < 0)
        {

         temp = array[2*i + 1];
         array[2*i + 1] = array[i];
         array[i] = temp;
        }

      }


}


heapup(int i)                 /* module for heapup operation */
{

      int parent;
      int temp;

   while(i!=0)
     {
      if((i % 2) == 1)
      {
         parent = (i/2);
      }



     else if((i % 2) == 0)
      {
         parent = ((i/2)-1);
      }

      if (array[parent] < array[i])
      {
         temp = array[parent];
         array[parent] = array[i];
         array[i] = temp;
      }
      else if (array[parent] > array[i])
      {
         /*do nothing*/
      }
      i--;

    }

}
```

Heap applications:

1. Operating System:

Operating system of a multiuser computer system may use job queues to save user's request in the order in which they are made. Another way such requests may be handled is according to how important the job request is. For example, an interactive program might get higher priority than a batch job. These requests may be stored in a structure called a priority queue. Since heap gives us fast access to the largest element (highest priority) in the structure, it is an excellent way to implement a priority queue.

2. Sorting:

   We always know where the largest element in the heap lies. Therefore, we can sort the list step by step.

8.5 Other trees

Different situations demand different implementations of trees. To help manage some peculiar situations, many variations of trees have been developed. We will briefly look at some of them.

1. Threaded Binary Trees

Traversing a binary tree is a common operation and threading provides a more efficient method for implementing the traversal. Let us consider the case of inorder traversal. We start traversing from leftmost lower node and whenever the current subtree is traversed, we go back to its parent, traverse it until we get back to root and start with right subtree. In this whole process, the location of the parent of the current subtree is to be stored.

Instead of this, what we can do is to have a pointer in the rightmost node of current subtree which points to the subtree's inorder successor. Then there would not be any need for storing parent's location, since the last node visited during traversal of left subtree points directly to its inorder successor. Such a pointer is called a thread. Have a look at the following diagram:

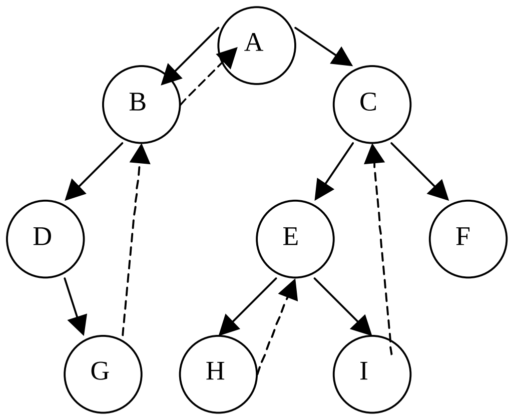


Fig. 8.25
A Right Threaded Binary Tree

Fig. 8.25 shows binary trees in which threads replace NULL pointers in nodes with empty right subtrees. In the figure, threads are drawn with dotted lines to differentiate them from tree branches. Also, the rightmost node in each tree still has a NULL right pointer, since it has no inorder successor. Such trees are called right threaded binary trees.

To implement threaded binary trees in C under the dynamic implementation, an extra logical field, rthread (right thread) is included within each node to indicate whether or not its right pointer is a thread.

```
struct node
{
      int info;
      struct node *left_child;
      struct node *right_child;
      int rthread;
}
```

A **left threaded binary tree** can be defined similarly, as one in which each NULL left pointer is altered to contain a thread to that node's inorder predecessor.

A **simple threaded binary tree** may be defined as a binary tree that is both left and right threaded.

2. Heterogeneous Binary Trees (Expression Tree)

Often the information contained in different nodes of a binary tree is not all of the same type. For example, to represent a mathematical expression we might like to use a binary tree whose leaves contain numbers but whose nonleaf nodes contain characters representing operators. Look at the following figure:

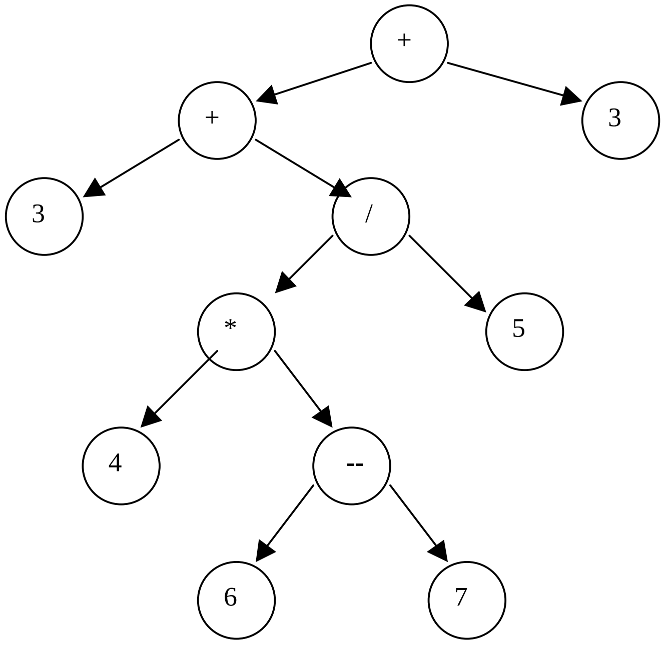


Fig. 8.26
Expression tree representing 3 + 4 * (6 – 7) / 5 + 3

3. AVL Trees

In many applications, insertions and deletions occur continually, with no predictable order. In some situations it may be essential to optimize search times by keeping the tree very nearly balanced at all times. Two Russian mathematicians achieved this and the resulting binary tree was named AVL tree in their honor.

In AVL trees, search, insertion and deletion in a tree with n nodes can all be achieved in time that is O (log $_2$ n), even in the worst case. Behavior of AVL trees closely approximates that of the ideal, completely balanced binary search trees.

In a completely balanced tree, the left and right subtrees of any node would have same height. Although, we cannot always achieve this goal, by building a search tree carefully, we can always ensure that the height of every left and right subtree never differs by more than 1.

Thus, an AVL tree is a binary search tree in which the heights of left and right subtrees of the root differ by atmost 1 and in which the left and right subtrees are again AVL trees.

With each node of an AVL tree, there is a balance factor associated with it that left high, equal or right high, accordingly as the left subtree has height greater than, equal to or less than that of right subtree.

Following are some of the examples of AVL trees:

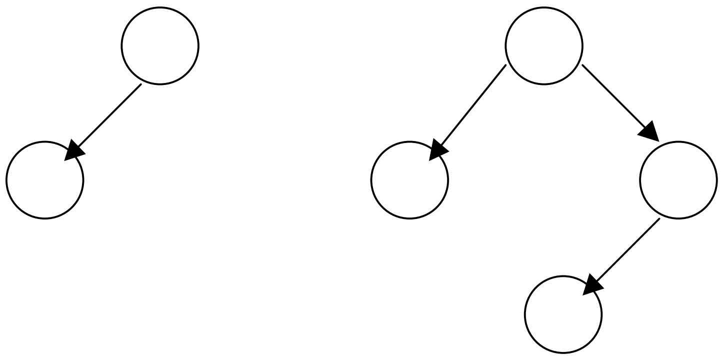


Fig 8.27
Examples of AVL trees

Further discussion of these different types of trees would be out of scope.

8.6 Analysis Of Binary Search Trees

A binary tree is appropriate for many applications of other list structures. A tree facilitates searching and at the same time provides the benefits of a linked structure. Like a sequential list, it can be searched quickly, using a binary search. Like a linked list, it allows insertions and deletions without having to move data. This is especially suitable for applications in which search time must be minimized or in which the nodes will not be necessarily be processed in sequential order.

Again, there is a trade-off. With an extra pointer in each node, a tree will take up more memory space than a linked list. Also the algorithms which invariably requires sequential processing rather than random processing of elements, the tree implementation can be avoided.

Suppose, from a customer database, if the main activity is to send monthly reports to each customer, then a linked list representation is more justifiable. But if, the activity involves information on a particular customer whenever required, then a tree structure would be a good choice.

Let's try to find out Big Oh measures of various tree operations.

1. **Searching for a node:**
   In a binary tree, any node can be located in $\log_2 n+1$ comparisons, where n is the number of elements in the tree. Therefore, the Big Oh measure of this operation is $O(\log_2 n)$.

2. **Inserting an element:**
   This operation is basically a search task plus one more simple operation of creating a node. Therefore, its Big Oh measure is $O(\log_2 n)$.

3. **Modifying an element:**
   This operation is basically a search task plus one more simple operation of changing the value. Therefore, its Big Oh measure is $O(\log_2 n)$.

4. **Retrieving an element:**
   This operation is basically a search task plus one more simple operation of copying the value. Therefore, its Big Oh measure is $O(\log_2 n)$.

5. **Deleting a node:**
   This operation is basically a search task plus one more operation of changing some pointers, which is $O(1)$. Therefore, its Big Oh measure is $O(\log_2 n)$.

6. **Printing a tree:**
   This operation requires the tree to be traversed, processing each element once. Thus, this is an $O(n)$ operation.

7. **Destroy tree:**
   This operation also requires the tree to be traversed, processing each element once. Thus, this is an `O(n)` operation.

The following table summarizes Big Oh measures of various operations on trees and linked lists.

Operation	**Binary Search Tree**	**Linked List**
`create`	`O(1)`	`O(1)`
`destroy`	`O(n)`	`O(n)`
`print`	`O(n)`	`O(n)`
`search`	$O(\log_2 n)$	`O(n)`
`insert`	$O(\log_2 n)$	`O(n)`
`eetrieve`	$O(\log_2 n)$	`O(n)`
`modify`	$O(\log_2 n)$	`O(n)`
`delete`	$O(\log_2 n)$	`O(n)`

Table 8.1
Big Oh Measures

8.7 Application Of Binary Trees

A binary tree is useful for maintaining a frequency count for the occurrence of each word in a text. Each structure stores a word and a frequency of occurrence of the word. As each new word is read, we search for it in the tree. If it is not in the tree, we add it and set the counter to 1. If it is already in the tree we increment the counter. This concept is used in various system programs like compilers. Also, in text editors we sometimes need to find out the number of occurrences of a particular word or replace every instance of it with some other word.

We develop our program using simple numbers instead of words to keep it as simplistic as possible.

Program: This program measures the frequency of occurrence of a particular number.

```
/*This program creates a tree and if it exists, inserts a new element at
a new place and looks for duplicates*/

#include<stdio.h>
#include<stdlib.h>

struct link_list                    /* structure of a node */

{
      int info;                     /* information field */
      int count ;             /* counter field */
      struct link_list *left_child; /* pointer to left child */
      struct link_list *right_child; /* pointer to right child */
};

typedef struct link_list node;

main()                              /* main starts here */
{

      node *root;             /* pointer to the root node */
      node *temp;             /* temporary node */
      node *p;                      /* traversing pointer */
      node *pre;              // pointer to keep track of //
                              // previous location //
      int flag=1;

      root = NULL;


      while(flag==1)
      {
            system("clear");
            temp = (node *)malloc(sizeof(node));

            printf("\nenter the element:");     // accepting the//
                                               //element //
            scanf("%d",&temp->info);
            fflush(stdin);

            temp->count = 1;
            temp->left_child = NULL;
            temp->right_child = NULL;

            if( root==NULL )        /* if it is the first node */
            {
               root = temp;
            }
            else                    //looking for a proper place //
            {                       //in the usual manner//
            p = root;
            while( p!=NULL)

            {
               if ( temp->info < p->info)
               {
                  pre = p;
                  p = p->left_child;
                  printf("leftchild\n");
               }
               else if (temp->info > p->info )
               {
                  pre = p;
                  p = p->right_child;
                  printf("right child\n");
               }
               else           /*if the number is already present*/
               {
                  printf("\n element already present.");

                  p->count = ( p->count + 1 );    //incrementing//
                                              //frequency counter//

                  printf("\nnumber of appearances: %d",p->count);

                  temp = NULL;
                  break;

               }
            }


            if ( temp->info < pre->info )

            {
               pre->left_child = temp;
            }
            else if ( temp->info > pre->info )
            {
               pre->right_child = temp;
            }
            }
            printf("\n");
            print(root);


            printf("\ndo you want to continue(yes=1/no=0):");
            scanf("%d",&flag);
            fflush(stdin);

            }

}                             /* main ends here */

print( node * q)              /* printing in inorder */
{
      if( q!=NULL)
      {

            print(q->left_child);
            printf("%d ",q->info);
            print(q->right_child);
      }

}
```

SELF ASSESSMENT

1. The three basic ways to traverse a list are _________, _________and ________.
2. A simple ____________ may be defined as a binary tree that is both left and right threaded.
3. The number of nodes joined to a node by edges is called the __________ that node.
4. The ___________ of a tree is the maximum number of nodes in a branch of a tree.
5. What are the applications of heaps ?
6. How is Binary tree different from a tree ?
7. What are AVL Trees ? Explain Heterogeneous trees.
8. Explain the term thread ?
9. Explain briefly applications of Binary trees.

SUMMARY

Key points covered in this chapter are:

- ♦ A tree is a non-linear data structure. This structure is used to represent data containing a hierarchical relationship between elements of a record.

- ♦ Trees, implemented by using pointers and linked lists can reduce the search to (log n) instead of O (n).

- ♦ A tree consists of nodes. The tree originates from a unique node. Such a node of a tree is called its root. Every node has a number of children nodes (may be zero). The nodes with no children are called leaves of the tree.

- ♦ Remember that a tree can be drawn in any direction, not necessarily with the root at the top and growing downwards. It is just a matter of convention to draw the root at the top.

- ♦ One node is joined to another node by means of an edge or a branch. The number of nodes joined to a node by edges is called the degree of that node.

- ♦ The level of a node in a tree T is defined by saying that the root is at level zero, and other nodes have a level that is one higher than the level its parent has.

- ♦ 2 trees are said to be similar if they have the same structure i.e. same shape. The trees are said to be copies if they have the same contents at corresponding nodes.

- ♦ The depth (or height) of a tree is the maximum number of nodes in a branch of a tree.

- ♦ The difference between a binary tree and a tree are,
    - A binary tree can be empty, whereas a tree cannot.

    - Each element in a binary tree has exactly 2 subtrees (one or both of them may be empty). Each element in a tree can have any number of subtrees.

- ♦ There are various operations that can be performed on a binary tree. Some basic sets of operations are as follows:
    - create
    - destroy
    - retrieve
    - insert
    - modify
    - delete
    - traversal

- ♦ Traversal means, passing through the tree, visiting each of its nodes once. There are 3 basic ways of traversal, as follows

  1. Preorder.
  2. Inorder.
  3. Postorder.

- ♦ A binary search tree is a binary tree that is either empty or in which each node contains a key that satisfies the conditions:

  1. All keys (if any) in the left subtree of the root precede the key in the root.
  2. The key in the root precedes all keys (if any) in its right subtree.
  3. The left and right subtrees of the root are again search trees.

- ♦ Searching in a tree follows the methodology of comparing the key with the root everytime, until the element is found. If the key is smaller, search proceeds with the left half of the tree. If the key is greater, search proceeds with the right half of the tree.

- ♦ Deletion requires two operations to be performed:

  1. Find the node in the tree that contains the element.
  2. Delete that node.

- ♦ A full binary tree is a binary tree in which all the leaves are on the same level and every non-leaf node has 2 children.

- ♦ A complete binary tree is a binary tree that is either full or the leaves on the last level.

- ♦ The array-based implementation of trees is simple for trees that are full or complete because the elements occupy contiguous array slots.

- ♦ A heap is a special case of a binary tree that satisfies2 properties,
  1. shape property,
  2. order property.

  - The shape property states that the heap must be a complete binary tree.
  - The order property states that, for every node in the heap, the value stored in that node is greater than or at least equal to that stored in its children.

- ♦ Different situations demand different implementations of trees. To help manage some peculiar situations, many variations of trees have been developed. Some of these are:
  1. Threaded trees.
  2. Heterogeneous trees.
  3. AVL trees.

- A binary tree is appropriate for many applications of other list structures. A tree facilitates searching and at the same time provides the benefits of a linked structure. This is especially suitable for applications in which search time must be minimized or in which the nodes will not be necessarily be processed in sequential order. Again, there is a trade-off. With an extra pointer in each node, a tree will take up more memory space than a linked list.

LAB EXERCISE

1. Write a program that prints only the terminal nodes of a tree in:

    1. Inorder
    2. Preorder
    3. Postorder

2. a) Write a function that will return the width of a linked binary tree, that is, the maximum number of nodes at the same level.

    b) Write a function that converts a binary tree into a doubly linked list, in which the nodes have the same order of inorder traversal of the tree. The function returns a pointer to the leftmost node of the doubly linked list, and the links `right` and `left` should be used to move through the list and be `NULL` at the two end of the list.

3. Write a function `GetNote()` that will traverse a linked list and get each node from the list in turn. Assume that list is simply linked with the links in the right field of each node.

NOTE: Q4, Q5 and Q6 have to be carried out in the next lab session.

4. Design a function that will delete the item with largest key (the root) from the top of the heap and restore the heap properties of the resulting, smaller list. Analyze the space and time requirements of your function.

5. Reader should try to implement binary search trees and heaps with all the possible operations that can be performed on them.

6. Create and maintain an employee record of a company. Perform the following activities on it:

    i. Print the list of employee records in alphabetical order.
    ii. Accept a name and print the particular record.
    iii. Accept an identification code and print the particular record.
    iv. Accept an integer S and print the names of all male employees when K=1 and the names of all female employees when K=2.
    v. Accept a name and delete the particular record from the database.
    vi. Accept a new record for a new employee and insert it in the proper place.