8

The Trees

In this chapter we will discuss one of the important non-liner data structure in computer science, Trees. Many real life problems can be represented and solved using trees.

Trees are very flexible, versatile and powerful non-liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grand children as so on.

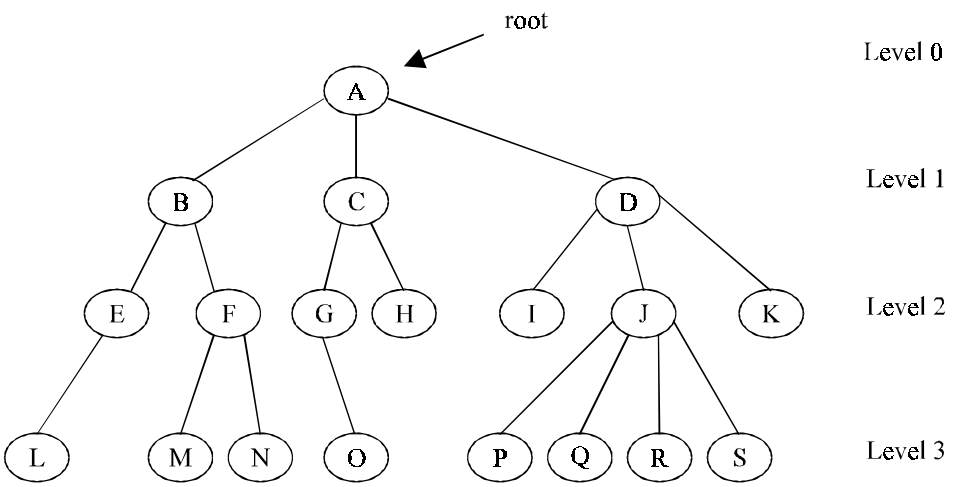


Fig. 8.1. A tree

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that :

1. There is a special node called the root of the tree.

2. Removing nodes (or data item) are partitioned into number of mutually exclusive (*i.e.*, disjoined) subsets each of which is itself a tree, are called sub tree.

8.1. BASIC TERMINOLOGIES

Root is a specially designed node (or data items) in a tree. It is the first node in the hierarchical arrangement of the data items. 'A' is a root node in the Fig. 8.1. Each data item in a tree is called a *node*. It specifies the data information and links (branches) to other data items.

Degree of a node is the number of subtrees of a node in a give tree. In Fig. 8.1

The degree of node A is 3

The degree of node B is 2

The degree of node C is 2

The degree of node D is 3

The *degree of a tree* is the maximum degree of node in a given tree. In the above tree, degree of a node J is 4. All the other nodes have less or equal degree. So the degree of the above tree is 4. A node with degree zero is called a *terminal node* or a *leaf*. For example in the above tree Fig. 8.1. M, N, I, O etc. are leaf node. Any node whose degree is not zero is called *non-terminal node*. They are intermediate nodes in traversing the given tree from its root node to the terminal nodes.

The tree is structured in different *levels*. The entire tree is leveled in such a way that the root node is always of level 0. Then, its immediate children are at level 1 and their immediate children are at level 2 and so on up to the terminal nodes. That is, if a node is at level n, then its children will be at level $n + 1$.

Depth of a tree is the maximum level of any node in a given tree. That is a number of level one can descend the tree from its root node to the terminal nodes (leaves). The term *height* is also used to denote the depth.

Trees can be divided in different classes as follows :

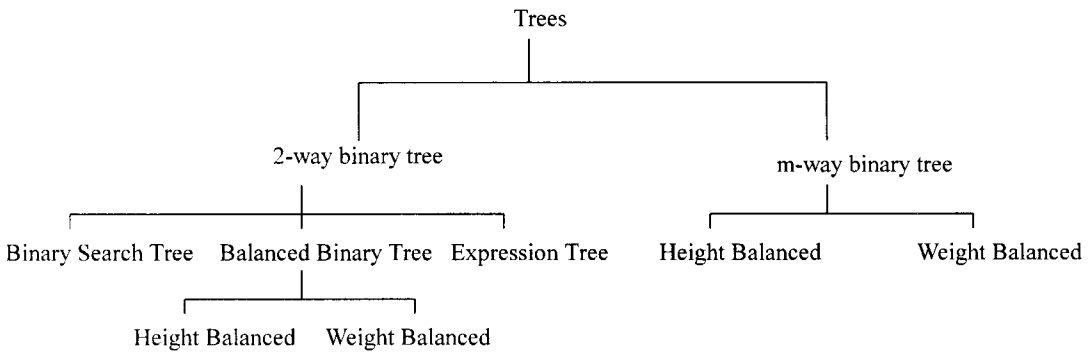


Fig. 8.2

8.2. BINARY TREES

A binary tree is a tree in which no node can have more than two children. Typically these children are described as "left child" and "right child" of the parent node.

A binary tree T is defined as a finite set of elements, called nodes, such that :

1. T is empty (*i.e.*, if T has no nodes called the *null tree* or *empty tree*).

2. T contains a special node R, called root node of T, and the remaining nodes of T form an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub tree of R. If T1 is non empty then its root is called the left successor of R, similarly if T2 is non empty then its root is called the right successor of R.

Consider a binary tree T in Fig. 8.3. Here 'A' is the root node of the binary tree T. Then 'B' is the left child of 'A' and 'C' is the right child of 'A' *i.e.*, 'A' is a father of 'B' and 'C'. The node 'B' and 'C' are called brothers, since they are left and right child of the same father. If a node has no child then it is called a leaf node. Nodes P,H,I,F,J are leaf node in Fig. 8.3.

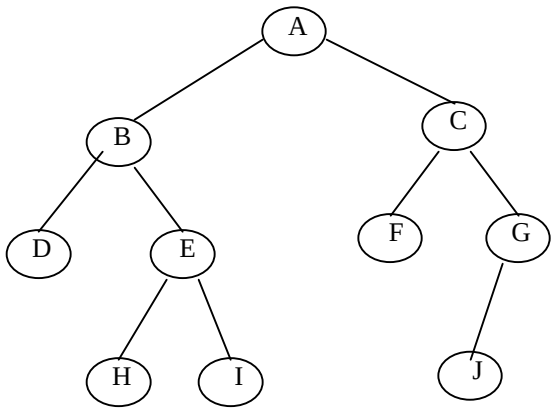


Fig. 8.3. Binary tree

The tree is said to be *strictly binary tree*, if every non-leaf made in a binary tree has non-empty left and right sub trees. A strictly binary tree with n leaves always contains $2n–1$ nodes. The tree in Fig. 8.4 is strictly binary tree, where as the tree in Fig. 8.3 is not. That is every node in the strictly binary tree can have either no children or two children. They are also called 2-*tree* or *extended binary tree.*

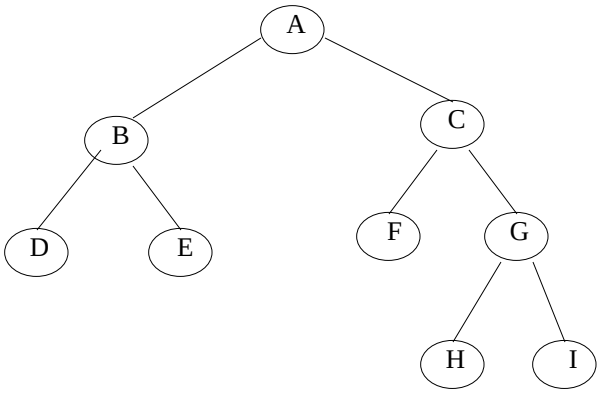


Fig. 8.4. Strictly binary tree

The main application of a 2-tree is to represent and compute any algebraic expression using binary operation.

For example, consider an algebraic expression E.

$$E = (a + b)/((c - d)*e)$$

E can be represented by means of the extended binary tree T as shown in Fig. 8.5. Each variable or constant in E appears as an internal node in T whose left and right sub tree corresponds to the operands of the operation.

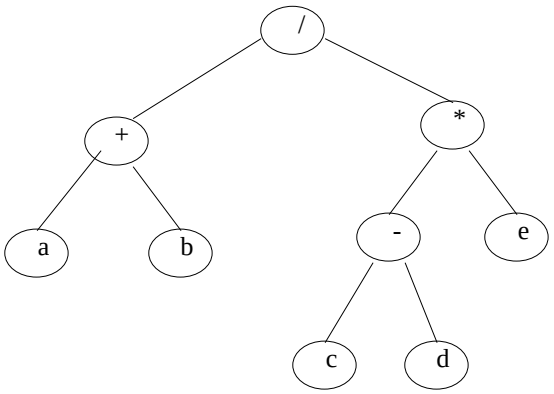


Fig. 8.5. Expression tree

A complete binary tree at depth '*d*' is the strictly binary tree, where all the leaves are at level *d*. Fig. 8.6 illustration the complete binary tree of depth 2.

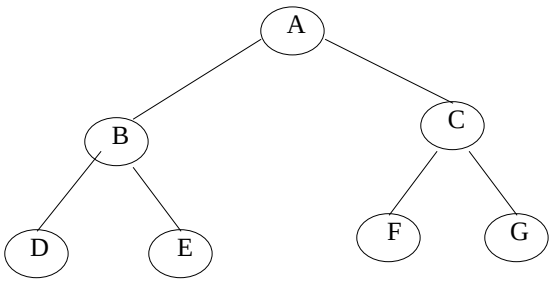


Fig. 8.6. Complete binary tree

A binary tree with n nodes, $n > 0$, has exactly $n - 1$ edges. A binary tree of depth d, $d > 0$, has at least d and at mast $2^d - 1$ nodes in it. If a binary tree contains n nodes at level l, then it contains at most $2n$ nodes at level $l + 1$. A complete binary tree of depth d is the binary tree of depth d contains exactly $2\ l$ nodes at each level l between 0 and d.

Finally, let us discuss in briefly the main difference between a binary tree and ordinary tree is:

1. A binary tree can be empty where as a tree cannot.
2. Each element in binary tree has exactly two sub trees (one or both of these sub trees may be empty). Each element in a tree can have any number of sub trees.
3. The sub tree of each element in a binary tree are ordered, left and right sub trees. The sub trees in a tree are unordered.

If a binary tree has only left sub trees, then it is called left skewed binary tree. Fig. 8.7(*a*) is a left skewed binary tree.

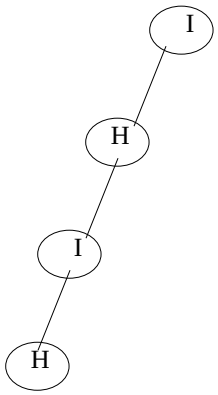


Fig. 8.7(*a*). Left skewed

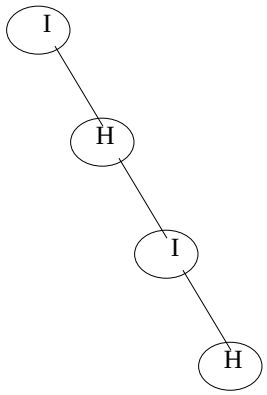


Fig. 8.7(*b*). Right skewed

If a binary tree has only right sub trees, then it is called right skewed binary tree. Fig. 8.7(*b*) is a right skewed binary tree.

8.3. BINARY TREE REPRESENTATION

This section discusses two ways of representing binary tree T in memory :

1. Sequential representation using arrays
2. Linked list representation

8.3.1. ARRAY REPRESENTATION

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially.

Suppose a binary tree T of depth *d*. Then at most 2^{d-1} nodes can be there in T.(*i.e.*, SIZE = 2^d–1) So the array of size "SIZE" to represent the binary tree. Consider a binary tree in Fig. 8.8 of depth 3. Then SIZE = $2^3 - 1 = 7$. Then the array A[7] is declared to hold the nodes.

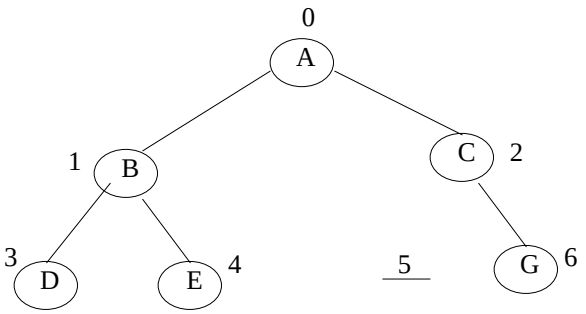


Fig. 8.8. Binary tree of depth 3

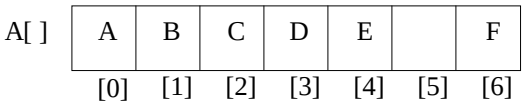


Fig. 8.9. Array representation of the binary tree

The array representation of the binary tree is shown in Fig. 8.9. To perform any operation often we have to identify the father, the left child and right child of an arbitrary node.

1. The father of a node having index n can be obtained by $(n - 1)/2$. For example to find the father of D, where array index $n = 3$. Then the father nodes index can be obtained

$$= (n - 1)/2$$
$$= 3 - 1/2$$
$$= 2/2$$
$$= 1$$

i.e., A[1] is the father D, which is B.

2. The left child of a node having index n can be obtained by (2n+1). For example to find the left child of C, where array index $n = 2$. Then it can be obtained by

$$= (2n + 1)$$
$$= 2*2 + 1$$
$$= 4 + 1$$
$$= 5$$

i.e., A[5] is the left child of C, which is NULL. So no left child for C.

3. The right child of a node having array index n can be obtained by the formula $(2n + 2)$. For example to find the right child of B, where the array index $n = 1$. Then

$$= (2n + 2)$$
$$= 2*1 + 2$$
$$= 4$$

i.e., A[4] is the right child of B, which is E.

4. If the left child is at array index n, then its right brother is at $(n + 1)$. Similarly, if the right child is at index n, then its left brother is at $(n - 1)$.

The array representation is more ideal for the complete binary tree. The Fig. 8.8 is not a complete binary tree. Since there is no left child for node C, *i.e.,* A[5] is vacant. Even though memory is allocated for A[5] it is not used, so wasted unnecessarily.

8.3.2. LINKED LIST REPRESENTATION

The most popular and practical way of representing a binary tree is using linked list (or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as :

(*a*) Left Child (LChild)

(*b*) Information of the Node (Info)

(*c*) Right Child (RChild)

The L Child links to the left child node of the parent node, Info holds the information of every node and R Child holds the address of right child node of the parent node. Fig. 8.10 shows the structure of a binary tree node.

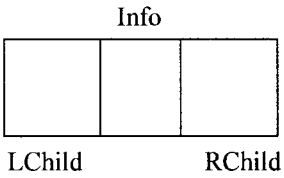


Fig. 8.10

Following figure (Fig. 8.11) shows the linked list representation of the binary tree in Fig. 8.8.

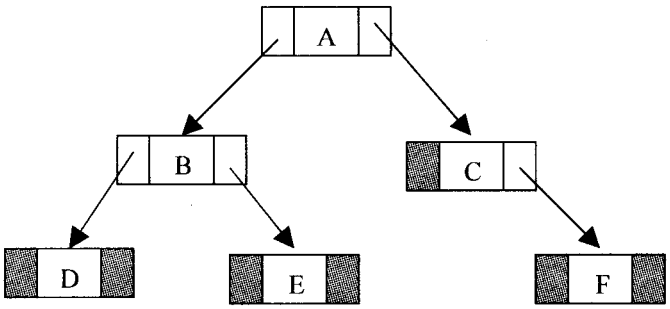


Fig. 8.11

If a node has left or/and right node, corresponding L Child or R Child is assigned to NULL. The node structure can be logically represented in C/C++ as:

```
struct Node
{
    int Info;
    struct Node *Lchild;
    struct Node *Rchild;
};

typedef struct Node *NODE;
```

8.4. OPERATIONS ON BINARY TREE

The basic operations that are commonly performed an binary tree is listed below :

1. Create an empty Binary Tree
2. Traversing a Binary Tree
3. Inserting a New Node

4. Deleting a Node
5. Searching for a Node
6. Copying the mirror image of a tree
7. Determine the total no: of Nodes
8. Determine the total no: leaf Nodes
9. Determine the total no: non-leaf Nodes
10. Find the smallest element in a Node
11. Finding the largest element
12. Find the Height of the tree
13. Finding the Father/Left Child/Right Child/Brother of an arbitrary node

Some primitive operations are discussed in the following sections. Implementation other operations are left to the reader.

8.5. TRAVERSING BINARY TREES RECURSIVELY

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are three standard ways of traversing a binary tree. They are:

1. Pre Order Traversal (Node-left-right)
2. In order Traversal (Left-node-right)
3. Post Order Traversal (Left-right-node)

8.5.1. PRE ORDERS TRAVERSAL RECURSIVELY

To traverse a non-empty binary tree in pre order following steps one to be processed

1. Visit the root node
2. Traverse the left sub tree in preorder
3. Traverse the right sub tree in preorder

That is, in preorder traversal, the root node is visited (or processed) first, before traveling through left and right sub trees recursively. It can be implement in C/C++ function as below :

```
void preorder (Node * Root)
{
        If (Root != NULL)
        {
                printf ("%d\n",Root → Info);
        preorder(Root → L child);
                preorder(Root → R child);
        }
}
```

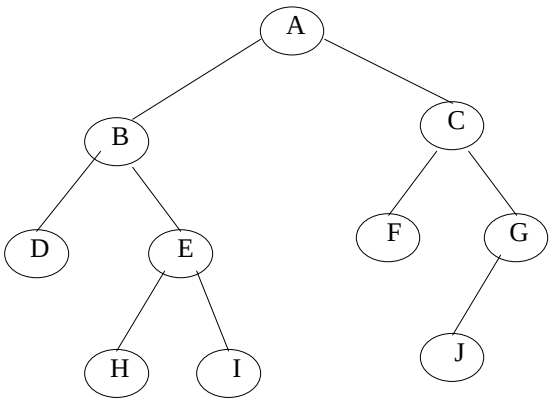


Fig. 8.12

The preorder traversal of a binary tree in Fig. 8.12 is A, B, D, E, H, I, C, F, G, J.

8.5.2. IN ORDER TRAVERSAL RECURSIVELY

The in order traversal of a non-empty binary tree is defined as follows :

1. Traverse the left sub tree in order
2. Visit the root node
3. Traverse the right sub tree in order

In order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root the right sub tree is traversed recursively, in order fashion. The procedure for an in order traversal is given below :

```
void inorder (NODE *Root)
{
        If (Root != NULL)
        {
        inorder(Root → L child);
        printf ("%d\n",Root → info);
        inorder(Root → R child);
        }
}
```

The in order traversal of a binary tree in Fig. 8.12 is D, B, H, E, I, A, F, C, J, G.

8.5.3. POST ORDER TRAVERSAL RECURSIVELY

The post order traversal of a non-empty binary tree can be defined as :

1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node

In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root.

```
void postorder (NODE *Root)
{
        If (Root != NULL)
        {
        postorder(Root → Lchild);
        postorder(Root → Rchild);
        printf ("%d\n",Root à info);
}
        }
```

The post order traversal of a binary tree in Fig. 8.12 is D, H, I, E, B, F, J, G, C, A

PROGRAM 8.1

```
//PROGRAM TO IMPLEMENT THE INSERTION AND DELETION IN B TREE
//CODED AND COMPILED USING TURBO C

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

#define M 5

//Structure is defined
struct node{
      int n; /* n < M No. of keys in node will always less than order of B tree */
      int keys[M–1]; /*array of keys*/
      struct node *p[M];   /* (n+1 pointers will be in use) */
}*root=NULL;

typedef struct node *NODE;
enum Key Status { Duplicate,Search Failure,Success,InsertIt,Less Keys };

//Function declrations
void insert(int key);
void display(NODE root,int);
```

```
void DelNode(int x);
void search(int x);
enum KeyStatus ins(NODE r, int x, int* y, NODE * u);
int searchPos(int x,int *key_arr, int n);
enum KeyStatus del(NODE r, int x);

void main()
{
	int key;
	int choice;
	while(1)
	{
		clrscr();//Clear the screen
		//Menu options
		printf ("\n1.Insert\n");
		printf ("2.Delete\n");
		printf ("3.Search\n");
		printf ("4.Display\n");
		printf ("5.Quit\n");
		printf ("\nEnter your choice : ");
		scanf ("%d",&choice);

		switch(choice)
		{
			case 1:
				printf ("\nEnter the key : ");
				scanf ("%d",&key);
				insert(key);
				break;
			case 2:
				printf ("\nEnter the key : ");
				scanf ("%d",&key);
				DelNode(key);
				break;
			case 3:
				printf ("\nEnter the key : ");
				scanf ("%d",&key);
				search(key);
				getch();
				break;
			case 4:
				printf ("\nBtree is :\n");
```

```
                        display(root,0);
                        getch();
                        break;
                case 5:
                        exit(1);
                default:
                        printf ("\nWrong choice\n");
                        getch();
                        break;
            }/*End of switch*/
        }/*End of while*/
}/*End of main()*/

void insert(int key)
{
        NODE newnode;
        int upKey;
        enum KeyStatus value;
        value = ins(root, key, &upKey, &newnode);
        //Cheking for duplicate keys
        if (value == Duplicate)
        {
                printf("\nKey already available\n");
                getch();
        }

        if (value == InsertIt)
        {
                NODE uproot = root;
                //Allocating memory
                root=(NODE)malloc(sizeof(struct node));
                root->n = 1;
                root->keys[0] = upKey;
                root->p[0] = uproot;
                root->p[1] = newnode;
        }/*End of if */
}/*End of insert()*/

enum KeyStatus ins(NODE ptr, int key, int *upKey,NODE *newnode)
{
        NODE newPtr,lastPtr;
        int pos, i, n,splitPos;
```

```
int newKey, lastKey;
enum KeyStatus value;
if (ptr == NULL)
{
        *newnode = NULL;
        *upKey = key;
        return InsertIt;
}
n = ptr->n;
pos = searchPos(key, ptr->keys, n);
if (pos < n && key == ptr->keys[pos])
        return Duplicate;
value = ins(ptr->p[pos], key, &newKey, &newPtr);
if (value != InsertIt)
        return value;
/*If keys in node is less than M–1 where M is order of B tree*/
if (n < M – 1)
{
        pos = searchPos(newKey, ptr->keys, n);
        /*Shifting the key and pointer right for inserting the new key*/
        for (i=n; i>pos; i--)
        {
                ptr->keys[i] = ptr->keys[i–1];
                ptr->p[i+1] = ptr->p[i];
        }
        /*Key is inserted at exact location*/
        ptr->keys[pos] = newKey;
        ptr->p[pos+1] = newPtr;
        ++ptr->n; /*incrementing the number of keys in node*/
        return Success;
}/*End of if */
/*If keys in nodes are maximum and position of node to be inserted is last*/
if (pos == M – 1)
{
        lastKey = newKey;
        lastPtr = newPtr;
}
else /*If keys in node are maximum and position of node to be inserted is not last*/
{
        lastKey = ptr->keys[M–2];
        lastPtr = ptr->p[M–1];
        for (i=M–2; i>pos; i--)
```

```
                {
                        ptr->keys[i] = ptr->keys[i–1];
                        ptr->p[i+1] = ptr->p[i];
                }
                ptr->keys[pos] = newKey;
                ptr->p[pos+1] = newPtr;
        }
        splitPos = (M – 1)/2;
       (*upKey) = ptr->keys[splitPos];

        (*newnode)=(NODE)malloc(sizeof(struct node));/*Right node after split*/
        ptr->n = splitPos; /*No. of keys for left splitted node*/
        (*newnode)->n = M–1-splitPos;/*No. of keys for right splitted node*/
        for (i=0; i < (*newnode)->n; i++)
        {
                (*newnode)->p[i] = ptr->p[i + splitPos + 1];
                if(i < (*newnode)->n – 1)
                        (*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
                else
                        (*newnode)->keys[i] = lastKey;
        }
        (*newnode)->p[(*newnode)->n] = lastPtr;
        return InsertIt;
}/*End of ins()*/

void display(NODE ptr, int blanks)
{
        if (ptr)
        {
                int i;
                for(i=1;i<=blanks;i++)
                        printf (“ ”);
                for (i=0; i < ptr->n; i++)
                        printf (“%d ”,ptr->keys[i]);
                printf (“\n”);
                for (i=0; i <= ptr->n; i++)
                        display(ptr->p[i], blanks+10);
        }/*End of if*/
}/*End of display()*/

void search(int key)
{
```

```
        int pos, i, n;
        NODE ptr = root;
        printf ("\nSearch path:\n");
        while (ptr)
        {
                n = ptr->n;
                for (i=0; i < ptr->n; i++)
                        printf (" %d",ptr->keys[i]);
                printf ("\n");
                pos = searchPos(key, ptr->keys, n);
                if (pos < n && key == ptr->keys[pos])
                {
                        printf ("\nKey %d found in position %d of last dispalyed node\n",key,i);
                        return;
                }
                ptr = ptr->p[pos];
        }
        printf ("\nKey %d is not available\n",key);
}/*End of search()*/

int searchPos(int key, int *key_arr, int n)
{
        int pos=0;
        while (pos < n && key > key_arr[pos])
                pos++;
        return pos;
}/*End of searchPos()*/

void DelNode(int key)
{
        NODE uproot;
        enum KeyStatus value;
        value = del(root,key);
        switch (value)
        {
        case SearchFailure:
                printf("\nKey %d is not available\n",key);
                break;
        case LessKeys:
                uproot = root;
                root = root->p[0];
                free(uproot);
```

```
            break;
        }/*End of switch*/
}/*End of delnode()*/

enum KeyStatus del(NODE ptr, int key)
{
        int pos, i, pivot, n ,min;
        int *key_arr;
        enum KeyStatus value;
        NODE *p,lptr,rptr;

        if (ptr == NULL)
                return SearchFailure;
        /*Assigns values of node*/
        n=ptr->n;
        key_arr = ptr->keys;
        p = ptr->p;
        min = (M – 1)/2;/*Minimum number of keys*/

        pos = searchPos(key, key_arr, n);
        if (p[0] == NULL)
        {
                if (pos == n || key < key_arr[pos])
                        return SearchFailure;
                /*Shift keys and pointers left*/
                for (i=pos+1; i < n; i++)
                {
                        key_arr[i–1] = key_arr[i];
                        p[i] = p[i+1];
                }
                return  --ptr->n >= (ptr==root ? 1 : min) ? Success : LessKeys;
        }/*End of if */

        if (pos < n && key == key_arr[pos])
        {
                struct node *qp = p[pos], *qp1;
                int nkey;
                while(1)
                {
                        nkey = qp->n;
                        qp1 = qp->p[nkey];
                        if (qp1 == NULL)
```

```
                    break;
                qp = qp1;
        }/*End of while*/
        key_arr[pos] = qp->keys[nkey–1];
        qp->keys[nkey – 1] = key;
}/*End of if */
value = del(p[pos], key);
if (value != LessKeys)
        return value;

if (pos > 0 && p[pos–1]->n > min)
{
        pivot = pos – 1; /*pivot for left and right node*/
        lptr = p[pivot];
        rptr = p[pos];
        /*Assigns values for right node*/
        rptr->p[rptr->n + 1] = rptr->p[rptr->n];
        for (i=rptr->n; i>0; i--)
        {
                rptr->keys[i] = rptr->keys[i–1];
                rptr->p[i] = rptr->p[i–1];
        }
        rptr->n++;
        rptr->keys[0] = key_arr[pivot];
        rptr->p[0] = lptr->p[lptr->n];
        key_arr[pivot] = lptr->keys[--lptr->n];
        return Success;
}/*End of if */
if (pos<n && p[pos+1]->n > min)
{
        pivot = pos; /*pivot for left and right node*/
        lptr = p[pivot];
        rptr = p[pivot+1];
        /*Assigns values for left node*/
        lptr->keys[lptr->n] = key_arr[pivot];
        lptr->p[lptr->n + 1] = rptr->p[0];
        key_arr[pivot] = rptr->keys[0];
        lptr->n++;
        rptr->n--;
        for (i=0; i < rptr->n; i++)
        {
                rptr->keys[i] = rptr->keys[i+1];
```

```
                rptr->p[i] = rptr->p[i+1];
        }/*End of for*/
        rptr->p[rptr->n] = rptr->p[rptr->n + 1];
        return Success;
    }/*End of if */

    if(pos == n)
        pivot = pos–1;
    else
        pivot = pos;

    lptr = p[pivot];
    rptr = p[pivot+1];
    /*merge right node with left node*/
    lptr->keys[lptr->n] = key_arr[pivot];
    lptr->p[lptr->n + 1] = rptr->p[0];
    for (i=0; i < rptr->n; i++)
    {
        lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
        lptr->p[lptr->n + 2 + i] = rptr->p[i+1];
    }
    lptr->n = lptr->n + rptr->n +1;
    free(rptr); /*Remove right node*/
    for (i=pos+1; i < n; i++)
    {
        key_arr[i–1] = key_arr[i];
        p[i] = p[i+1];
    }
    return  --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
}/*End of del()*/
```

8.6. TRAVERSING BINARY TREE NON-RECURSIVELY

In this section we will discuss the implementation of three standard traversals algorithms, which were defined recursively in the last section, non-recursively using stack.

8.6.1. PREORDER TRAVERSAL NON-RECURSIVELY

The preorder traversal non-recursively algorithms uses a variable PN (Present Node), which will contain the location of the node currently being scanned. Left(R) denotes the left child of the node R and Right(R) denoted the right child of R. A stack is used to hold the addresses of the nodes to be processed. Info(R) denotes the information of the node R.

Preorder traversal starts with root node of the tree *i.e.*, PN = ROOT. Then repeat the following steps until PN = NULL.

Step 1: Process the node PN. If any right child is there for PN, push the Right (PN) into the top of the stack and proceed down to left by PN = Left (PN), if any left child is there (*i.e.*, Left (PN) not equal to NULL).

Repeat the step 2 until there is no left child (*i.e.*, Left (PN) = NULL).

Step 2: Now we have to go back to the right node(s) by backtracking the tree. This can be achieved by popping the top most element of the stack. Pop the top element from the stack and assigns to PN.

Step 3: If (PN is not equal to NULL) go to the Step 1

Step 4: Exit

The implementation of the preorder non-recursively traversal algorithm can be illustrated with an example. Consider a binary tree in Fig. 8.13. Following steps are generated when the algorithm is applied to the following binary tree :

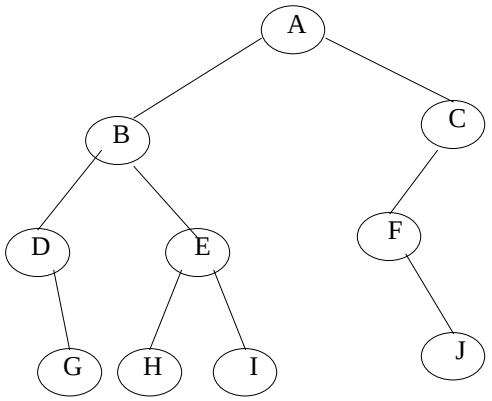


Fig. 8.13

1. Initialize the Root node to PN

SATCK :

PN = ROOT (*i.e.*, PN = A)

2. Process the node PN (*i.e.*, A)

If PN has the right child push it into stack (*i.e.*, C)

If PN has the left child proceed down to left by PN = Left(A) (*i.e.*, PN = B)

STACK: C

3. Process the node PN (*i.e.*, B)

If PN has the right child (*i.e.*, Right (PN) not equal to NULL) then push the right child of PN into the stack (*i.e.*, Right(B) is E)

If PN has the left child proceed down to left by PN = Left(B) (*i.e.*, Now PN = D)

STACK: C, E

4. Process or display the node PN (*i.e.*, D)

If PN has the right child, then push the right child of PN into the stack (*i.e.*, Right(D) is G)

If PN has the left child proceed down to left. Here Left(PN) is equal to NULL, so no left child.

STACK: C, E, G

5. Now the backtracking process will start (*i.e.*, when Left(PN) = NULL)

Pop the top element G from the stack and assign it to PN (*i.e.*, PN = G)

STACK: C, E

6. Process the node G

Check for right child of PN (*i.e.*, G) No right child (*i.e.*, Right(G) = NULL

Check for left child of PN (*i.e.*, G) No left child also (*i.e.*, Left(G) = NULL)

STACK: C, E

7. Again pop the top element E from the stack and assign it to PN (*i.e.*, PN = E)

STACK: C

8. Process the node E (PN)

Since (Right(E) is not equal to NULL)

Push(Right(E)) (*i.e.*, Right(E) is E)

Since (Left(E) is not equal to NULL)

PN = Left(PN) = Left(E) (*i.e.*, PN = H)

STACK: C, I

9. Process the node H

Since (Right(H) = NULL)

Do nothing

Since (Left(H) = NULL)

Do nothing

STACK: C, I

10. Backtracking to right sub tree elements

Pop the top element I from the stack and assign it to PN (*i.e.*, PN = I)

STACK: C

11. Process the node I

No left child for I

No right child for I

STACK: C

12. Again backtracking

Pop the top element C and assign it to PN (*i.e.*, PN=C)

STACK:

13. Display (or process) the node C

Since (Right(C) = NULL)

Do nothing

Since (Left(C) is not equal to NULL)

PN = Left(PN) = Left(C) (*i.e.*, PN = F)

STACK:

14. Display the node F

Since (Right(F) is not equal to NULL)

Push Right(F) to the stack (*i.e.*, J)

Since (Left(F) = NULL)

Do Nothing

STACK: J

15. Backtracking to right node(s)

Pop the top element J and assign it to PN (*i.e.*, PN = J)

STACK:

16. Display the node J

(Right(J) = NULL)

(Right(J) = NULL)

STACK:

17. Backtracking for right nodes. Now the top pointer is pointing to NULL. Assign the top value to PN. (*i.e.*, PN=NULL)

18. When (PN = NULL) STOP

The nodes are processed or displayed in the order A, B, D, G, E, H, I, C, F, J.

ALGORITHM

An array STACK is used to hold the addresses of nodes. TOP pointer points to the top most element of the STACK. ROOT is the root node of tree to be traversed. PN is the address of the present node under scanning. Info(PN) if the information contained in the node PN.

1. Initialize TOP = NULL, PN = ROOT
2. Repeat step 3 to 5 until (PN = NULL)
3. Display Info(PN)
4. If (Right(PN) not equal to NULL)
   (*a*) TOP = TOP+1
   (*b*) STACK(TOP) = Right(PN);
5. If(Left(PN) not equal to NULL)
   (*a*) PN = Left(PN)
6. Else
   (*a*) PN = STACK[TOP]
   (*b*) TOP = TOP–1
7. Exit

PROGRAM 8.2

```
//FUNCTION TO IMPLEMENT NON RECURSIVE PRE ORDER TRAVERSAL
//CODED AND COMPILED IN TURBO C++

void preorder(struct tnode *p)
{
        struct node *stack[100];
        int top;
        top = –1;
        if(p != NULL)
        {
                cout<<“ ”<<p->info;
                if(p->rchild != NULL)
{
                        top++;
                stack[top] = p->rchild;
                }
                p = p->lchild;
                while(top >= –1)
                {
while ( p!= NULL)/* push the left child onto stack*/
{
printf(“%d\t”,p->data);
                                if(p->rchild != NULL)
{
                        top++;
                stack[top] = p->rchild;
                        }
                        p = p->lchild;
                        }
                        p = stack[top];
                top--;
                }
        }
}
```

8.6.2. IN ORDER TRAVERSAL NON-RECURSIVELY

The in-order traversal algorithm uses a variable PN, which will contain the location of the node currently being scanned. Info (R) denotes the information of the node R, Left (R) denotes the left child of the node R and Right (R) denotes the right child of the node R.

In-order traversal starts from the ROOT node of the tree (*i.e.*, PN = ROOT). Then repeat the following steps until PN = NULL :

Step 1: Proceed down to left most node of the tree by pushing the root node onto the stack.

Step 2: Repeat the step 1 until there is no left child for a node.

Step 3: Pop the top element of the stack and process the node. PN = STACK[TOP]

Step 4: If the stack is empty then go to step 6.

Step 5: If the popped element has right child then PN = Right(PN). Then repeat the step from 1.

Step 6: Exit.

The in-order traversal algorithm can be illustrated with an example. Consider a binary tree in Fig. 8.13. Following steps may generate if we try to traverse the tree in in-order fashion :

1. Initialize root Node to PN (*i.e.*, PN = ROOT = A)

   STACK:

2. Since(Left(PN) is not equal to NULL

Push (PN) to the stack

PN = Left(PN) (*i.e.*, = B)

   STACK: A

3. Since(Left(PN) is not equal to NULL

Push (PN) to the stack (*i.e.*, = B)

PN = Left(PN) (*i.e.*, = D

   STACK: A, B

4. Since(Left(PN) = NULL

Display the node D

   STACK: A, B

5. Since(Right(PN) is not equal to NULL

PN = Right(PN) = G

   STACK: A, B

6. Since(Left(PN) = NULL

Display the node G

   STACK: A, B

7. Since(Right(PN) = NULL)

Pop the topmost element of the stack

PN = STACK[TOP] (*i.e.*, = B)

Display the node B

   STACK: A

8. Since(Right(PN) is not equal to NULL)

PN = Right(PN) = E

   STACK: A

9. Since(Left(PN) is not equal to NULL
Push(PN) to the stack (*i.e.*, E)
PN = Left(PN) = H
STACK: A, E
10 Since(Left(PN) = NULL)
Display the node H
STACK: A, E
11. Since(Right(PN) = NULL
Pop the topmost element of the stack
PN = STACK[TOP] = E
Display the node E
STACK: A
12. Since(Right(PN) is not equal to NULL)
PN = Right(PN) = I
STACK: A
13. Since(Left(PN) = NULL)
Display the node I
STACK: A
14. Since(Right(PN) = NULL)
Pop the topmost element of the stack
PN = STACK[TOP] = A
Display the node A
STACK:
15. Since(Right(PN) not equal to NULL)
PN = Right(PN) = C
STACK:
16. Since(Left(PN) is not equal to NULL)
Push(PN) to the stack (*i.e.*, = C)
PN = Left(PN) = F
STACK: C
17. Since(Left(PN) = NULL)
Display the node F
STACK: C
18. Since(Right(PN) is not equal to NULL)
PN = Right(PN) = J
STACK: C
19. Since(Left(PN) = NULL
Display the node J
STACK: C

20. Since(Right(PN) = NULL)

Pop the element from the stack

PN = STACK[TOP] = C

Display the node C

STACK:

21. Since(Right(PN) = NULL)

Try to pop an element from the stack. Since the stack is empty PN=NULL and Stop

The nodes are displayed in the order of D, G, B, H, E, I, A, F, J, C.

ALGORITHM

An array STACK is used to temporarily store the addresses of the nodes. TOP pointer always points to the topmost element of the STACK.

1. Initialize TOP = NULL and PN = ROOT
2. Repeat the Step 3, 4 and 5 until (PN = NULL)
3. TOP = TOP +1
4. STACK[TOP] = PN
5. PN = Left(PN)
6. PN = STACK[TOP]
7. TOP = TOP–1
8. Repeat steps 9, 10, 11 and 12 until (PN = NULL)
9. Display Info(PN)
10. If(Right(PN) is not equal to NULL
    (*a*) PN = Right(PN)
    (*b*) Go to Step 6
11. PN = STACK[TOP]
12. TOP = TOP –1
13. Exit

PROGRAM 8.3

```
//FUNCTION TO IMPLEMENT NON RECURSIVE IN ORDER TRAVERSAL
//CODED AND COMPILED IN TURBO C++
void inorder(struct tnode *p)
{
        struct tnode *stack[100];
        int top;
        top = –1;
        if(p != NULL)
        {
```

```
                top++;
                stack[top] = p;
                p = p->lchild;
                while(top >= 0)
                {
while ( p!= NULL)/* push the left child onto stack*/
{
                        top++;
                                stack[top] =p;
                        p = p->lchild;
                        }
                        p = stack[top];
                top--;
                        cout<<“ ”<<p->data;
                p = p->rchild;
                        if ( p != NULL) /* push right child*/
                        {
                                top++;
                        stack[top] = p;
                                p = p->lchild;
                        }
                }
        }
}
```

8.6.3. POSTORDER TRAVERSAL NON-RECURSIVELY

The post-order traversal algorithm uses a variable PN, which will contain the location of the node currently being scanned. Left (R) denotes the left child of the node R and Right (R) denotes the right child of the node R. Info (R) denotes the information of the node R.

The post-order traversal algorithm is more complicated than the proceeding two algorithms, because here we have to push the information of the node PN to stack in two different situations. These two situations are distinguished between by pushing Left(PN) and - Right(PN) on to stack. That is whenever a negative node sees in the stack; it means that it was a right child of a node. Post-order traversal starts from the ROOT node of the tree (*i.e.,* PN = ROOT).

Step 1: Proceed down to left most node of the tree by pushing the root node and - Right(PN) on the stack.

Step 2: Repeat the Step 1 until there is no left child for the node.

Step 3: Pop and display the positive nodes on the stack.

Step 4: If the stack is empty, go to Step 6

Step 5: If a negative node is popped, then PN = – PN (*i.e.,* to remove the negative sign in the node) and go to Step 1.

Step 6: Exit.

The post-order traversal algorithm can be illustrated with a binary tree in Fig. 8.13.

1. Initialize ROOT Node to PN (*i.e.*, PN = A)

STACK:

2. Push(PN) to the stack

Since(Right(A) is not equal to NULL)

Push(-Right(A)) to the stack

Then If(Left(A) is not equal to NULL)

PN = Left(PN) (*i.e.*, PN=B)

STACK: A, – C

3. Push (B) to the stack

Since (Right(B) is not equal to NULL)

Push(-Right(B)) to the stack

Then If(Left(B) is not equal to NULL)

PN=Left(PN) (*i.e.*, PN =D)

STACK : A, – C, B, –E

4. Push (D) to the stack

Since (Right(D) is not equal to NULL)

Push(-Right(D)) to the stack

Since(Left(D) = NULL)

STACK : A, – C, B, – E, D, –G

Next step is pop and display all the positive elements from the top until a negative element is reached. Here the top element is a negative one, so only the top of the stack is popped (*i.e.*, -G) and assigned to PN. Now PN= – G. Set PN = – PN (*i.e.*, PN = G)

STACK: A, – C, B, – E, D

5. Push(G) to the stack

If(Right(G) = NULL)

Do nothing

If(Left(G) = NULL)

STACK: A, – C, B, – E, D, G

Pop and display all the positive elements from the top of the stack until a negative element is reached. Here the G, D are popped and displayed. – E is popped and assigned it to PN then PN = – PN = E

STACK: A, – C, B,

6. Push(E) to the stack

Since(Right(E) is not equal to NULL)

Push(– Right(E)) to the stack

Since(Left(E) is not equal to NULL)

PN=Left(PN) (*i.e.*, PN=H)

STACK: A, – C, B, E, – I

7. Push(H) to the stack

If(Right(H) = NULL)

Do nothing

If(Left(H) = NULL)

STACK: A, – C, B, E, – I, H

Pop and display H then pop and assign I to PN (*i.e.*, PN = I)

STACK: A, – C, B, E,

8. Push(I) to the stack

If(Right(I) = NULL)

Do nothing

If(Left(I) = NULL)

STACK: A, – C, B, E, I

Pop and display all positive elements of stack from top, until a negative number is reached. Here I, E, B are popped and displayed. And – C is popped and assigned to PN (*i.e.*, PN = C)

STACK: A,

9. Push(C) to the stack

If(Right(C) = NULL)

Do nothing

If(Left(C) = NULL)

PN = Left(C) = F

STACK: A, C

10. Push(F) to the stack

Since(Right(F) is not equal to NULL)

Push(-Right(F))s

Since(Left(F) = NULL)

STACK: A, C, F, – J

Pop the top element and assign it to PN. (*i.e.*, PN = J)

11. Push(J) to the stack

Since(Right(J) = NULL)

Do nothing

Since(Left(J) = NULL)

Pop all positive elements from top, until a negative element is reached. Here all the elements J, F, C, A are popped and displayed. Now the stack is empty and PN = NULL and STOP.

The nodes are displayed in the following order G, D, H, I, E, B, J, F, C, A.

ALGORITHM

An array STACK is used to temporarily store the addresses of the nodes. TOP pointer always points to the top most element of the stack.

1. Initialize TOP = NULL and PN = ROOT
2. Repeat the steps 3 to 6 until (PN = NULL)
3. TOP = TOP+1
4. SATCK(TOP)=PN
5. If (Right(PN) is not equal to NULL)
   (*a*) TOP = TOP+1
   (*b*) STACK(TOP) = – Right(PN)
6. PN = Left(PN)
7. PN = STACK(TOP)
8. TOP = TOP–1
9. Repeat the step 9 until (PN less than or equal to 0)
   (*a*) Display Info(PN)
   (*b*) PN = STACK(TOP)
   (*c*) TOP = TOP=1
10. If(PN < 0)
   (*a*) PN = – PN
   (*b*) Go to step 2
11. Exit

PROGRAM 8.3

```
//FUNCTION TO IMPLEMENT NON RECURSIVE POST ORDER TRAVERSAL
//CODED AND COMPILED IN TURBO C++

void postorder(struct node *p)
{
      struct node *stack[100];
      int top,sig, sign[100];
      top = –1;
      if(p != NULL)
      {
         top++;
         stack[top] = p;
   sign[top]=1;
      if(p->rchild != NULL)
            {
                  top++;
                  stack[top] = p->rchild;
                  sign[top]=–1;
            }
```

```
        p = p->lchild;
        while(top >= 0)
        {
                while ( p!= NULL)/* push the left child onto stack*/
                {
                                top++;
                                stack[top] = p;
                        sign[top]=1;
                        if(p->rchild != NULL)
                  {
                                top++;
                                stack[top] = p->rchild;
                                sign[top]=–1;
                  }
                        p = p->lchild;
                  }

                  p = stack[top];
                  sig=sign[top];
                  top--;

                  while((sig > 0) && (top >= -1))
                  {
                                cout<<“ ”<<p->info;
                                p = stack[top];
                                sig=sign[top];
                                top--;
                 }


                 }
         }
  }
```

8.7. BINARY SEARCH TREES

A Binary Search Tree is a binary tree, which is either empty or satisfies the following properties :

1. Every node has a value and no two nodes have the same value (*i.e.*, all the values are unique).

2. If there exists a left child or left sub tree then its value is less than the value of the root.

3. The value(s) in the right child or right sub tree is larger than the value of the root node.

All the nodes or sub trees of the left and right children follows above rules. The Fig. 8.14 shows a typical binary search tree. Here the root node information is 50. Note that the right sub tree node's value is greater than 50, and the left sub tree nodes value is less than 50. Again right child node of 25 has large values than 25 and left child node has small values than 25. Similarly right child node of 75 has large values than 75 and left child node has small values that 75 and so on.

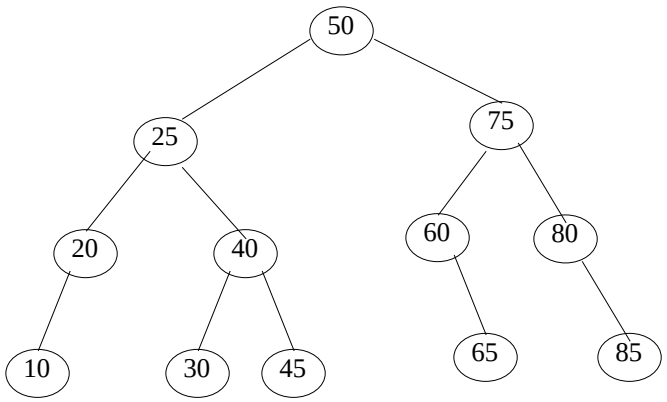


Fig. 8.14

The operations performed on binary tree can also be applied to Binary Search Tree (BST). But in this section we discuss few other primitive operators performed on BST :

1. Inserting a node
2. Searching a node
3. Deleting a node

Another most commonly performed operation on BST is, traversal. The tree traversal algorithm (pre-order, post-order and in-order) are the standard way of traversing a binary search tree.

8.7.1. INSERTING A NODE

A BST is constructed by the repeated insertion of new nodes to the tree structure. Inserting a node in to a tree is achieved by performing two separate operations.

1. The tree must be searched to determine where the node is to be inserted.
2. Then the node is inserted into the tree.

Suppose a "DATA" is the information to be inserted in a BST.

Step 1: Compare DATA with root node information of the tree

(*i*) If (DATA < ROOT → Info)

Proceed to the left child of ROOT

(*ii*) If (DATA > ROOT → Info)

Proceed to the right child of ROOT

Step 2: Repeat the Step 1 until we meet an empty sub tree, where we can insert the DATA in place of the empty sub tree by creating a new node.

Step 3: Exit

For example, consider a binary search tree in Fig. 8.14. Suppose we want to insert a DATA = 55 in to the tree, then following steps one obtained :

1. Compare 55 with root node info (*i.e.*, 50) since 55 > 50 proceed to the right sub tree of 50.

2. The root node of the right sub tree contains 75. Compare 55 with 75. Since 55 < 75 proceed to the left sub tree of 75.

3. The root node of the left sub tree contains 60. Compare 55 with 60. Since 55 < 60 proceed to the right sub tree of 60.

4. Since left sub tree is NULL place 55 as the left child of 60 as shown in Fig. 8.15.

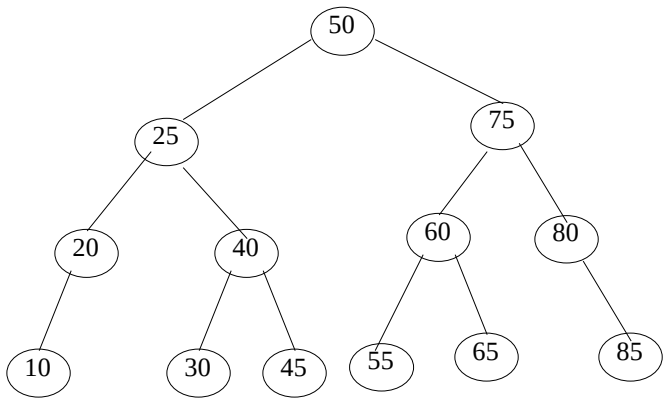


Fig. 8.15

ALGORITHM

NEWNODE is a pointer variable to hold the address of the newly created node. DATA is the information to be pushed.

1. Input the DATA to be pushed and ROOT node of the tree.
2. NEWNODE = Create a New Node.
3. If (ROOT == NULL)
   (*a*) ROOT=NEW NODE
4. Else If (DATA < ROOT → Info)
   (*a*) ROOT = ROOT → Lchild
   (*b*) GoTo Step 4
5. Else If (DATA > ROOT → Info)
   (*a*) ROOT = ROOT → Rchild
   (*b*) GoTo Step 4

6. If (DATA < ROOT → Info)
   (*a*) ROOT → LChild = NEWNODE
7. Else If (DATA > ROOT → Info)
   (*a*) ROOT → RChild = NEWNODE
8. Else
   (*a*) Display (“DUPLICATE NODE”)
   (*b*) EXIT
9. NEW NODE → Info = DATA
10. NEW NODE → LChild = NULL
11. NEW NODE → RChild = NULL
12. EXIT

8.7.2. SEARCHING A NODE

Searching a node was part of the operation performed during insertion. Algorithm to search as element from a binary search tree is given below.

ALGORITHM

1. Input the DATA to be searched and assign the address of the root node to ROOT.
2. If (DATA == ROOT → Info)
   (*a*) Display “The DATA exist in the tree”
   (*b*) GoTo Step 6
3. If (ROOT == NULL)
   (*a*) Display “The DATA does not exist”
   (*b*) GoTo Step 6
4. If(DATA > ROOT→Info)
   (*a*) ROOT = ROOT→RChild
   (*b*) GoTo Step 2
5. If(DATA < ROOT→Info)
   (*a*) ROOT = ROOT→Lchild
   (*b*) GoTo Step 2
6. Exit

Suppose a binary search tree contains *n* data items, A1, A2, A3.....A*n*. There are *n*! permutations of the *n* items. The average depth of the n! tree is approximately C log 2 *n*, where C=1.4. The average running time $f(n)$ to search for an item in a binary tree with *n* elements is proportional to $\text{Log}_2\ n$, that is

$$f(n) = O(\text{Log}_2\ n)$$

8.7.3. DELETING A NODE

This section gives an algorithm to delete a DATA of information from a binary search tree. First search and locate the node to be deleted. Then any one of the following conditions arises :

1. The node to be deleted has no children

2. The node has exactly one child (or sub tress, left or right sub tree)

3. The node has two children (or two sub tress, left and right sub tree)

Suppose the node to be deleted is N. If N has no children then simply delete the node and place its parent node by the NULL pointer.

If N has one child, check whether it is a right or left child. If it is a right child, then find the smallest element from the corresponding right sub tree. Then replace the smallest node information with the deleted node. If N has a left child, find the largest element from the corresponding left sub tree. Then replace the largest node information with the deleted node.

The same process is repeated if N has two children, *i.e.*, left and right child. Randomly select a child and find the small/large node and replace it with deleted node. NOTE that the tree that we get after deleting a node should also be a binary search tree.

Deleting a node can be illustrated with an example. Consider a binary search tree in Fig. 8.15. If we want to delete 75 from the tree, following steps are obtained :

Step 1: Assign the data to be deleted in DATA and NODE = ROOT.

Step 2: Compare the DATA with ROOT node, *i.e.*, NODE, information of the tree.

Since (50 < 75)

NODE = NODE → RChild

Step 3: Compare DATA with NODE. Since (75 = 75) searching successful. Now we have located the data to be deleted, and delete the DATA. (See Fig. 8.16)

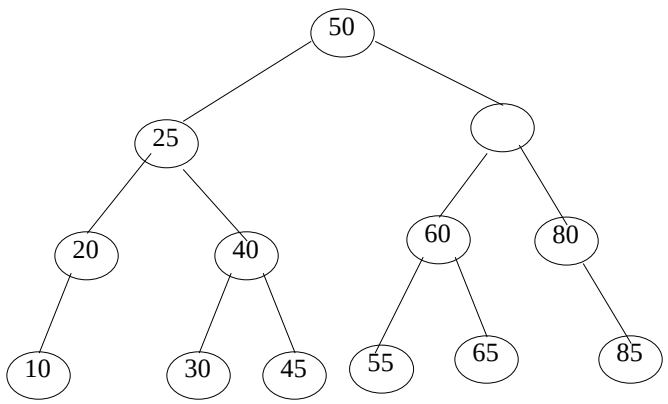


Fig. 8.16

Step 4: Since NODE (*i.e.*, node where value was 75) has both left and right child choose one. (Say Right Sub Tree) - If right sub tree is opted then we have to find the smallest node. But if left sub tree is opted then we have to find the largest node.

Step 5: Find the smallest element from the right sub tree (*i.e.*, 80) and replace the node with deleted node. (See Fig. 8.17)

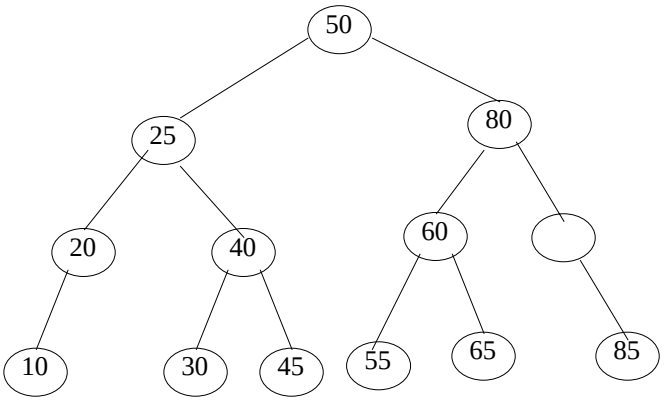


Fig. 8.17

Step 6: Again the (NODE → Rchild is not equal to NULL) find the smallest element from the right sub tree (Which is 85) and replace it with empty node. (See Fig. 8.18)

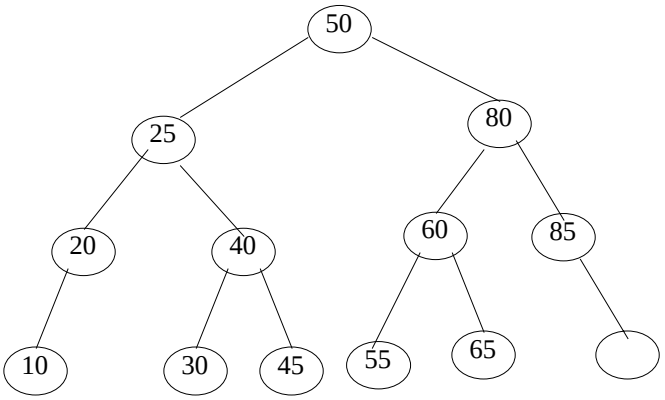


Fig. 8.18

Step 7: Since (NODE→Rchild = NODE→Lchild = NULL) delete the NODE and place NULL in the parent node. (See Fig. 8.19)

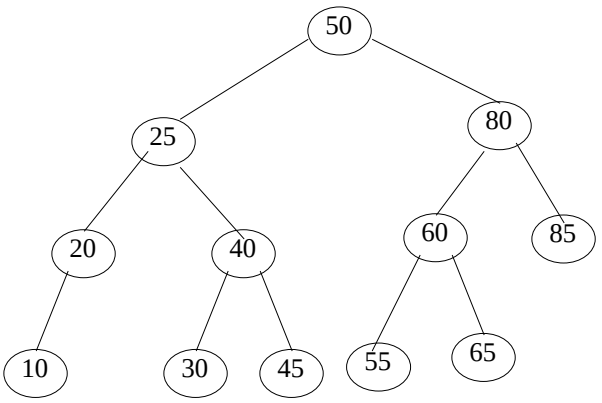


Fig. 8.19

Step 8: Exit.

ALGORITHM

NODE is the current position of the tree, which is in under consideration. LOC is the place where node is to be replaced. DATA is the information of node to be deleted.

1. Find the location NODE of the DATA to be deleted.
2. If (NODE = NULL)
   (*a*) Display "DATA is not in tree"
   (*b*) Exit
3. If(NODE → Lchild = NULL)
   (*a*) LOC = NODE
   (*b*) NODE = NODE → RChild
4. If(NODE → RChild= =NULL)
   (*a*) LOC = NODE
   (*b*) NODE = NODE → LChild
5. If((NODE → Lchild not equal to NULL) && (NODE → Rchild not equal to NULL))
   (*a*) LOC = NODE → RChild
6. While(LOC → Lchild not equal to NULL)
   (*a*) LOC = LOC → Lchild
7. LOC → Lchild = NODE → Lchild
8. LOC → RChild= NODE → RChild
9. Exit

PROGRAM 8.5

```
//PROGRAM TO IMPLEMENT THE OPERATION SUCH AS
//INSERTION, DELETION AND TRAVERSAL IN BINARY SEARCH TREE
//CODED AND COMPILED USING TURBO C++

#include<iostream.h>
#include<process.h>
#include<conio.h>

//Class is created for the implementation of BST
class BST
{
      struct node
      {
            int info;
            struct node *lchild;
            struct node *rchild;
```

```
        };
        typedef struct node *NODE;

        public:
                struct node *root;
                BST()
                {
                        root=NULL;
                }
                //public functions declarations
                void find(int,NODE *,NODE *);
                void case_a(NODE,NODE);
                void case_b(NODE,NODE);
                void case_c(NODE,NODE);
                void insert(int);
                void del(int);
                void preorder(NODE);
                void inorder(NODE);
                void postorder(NODE);
                void display(NODE,int);
};

//Function to find the item form the tree
void BST::find(int item,NODE *par,NODE *loc)
{
        NODE ptr,ptrsave;

        if(root==NULL)  /*tree empty*/
        {
                *loc=NULL;
                *par=NULL;
                return;
        }
        if(item==root->info) /*item is at root*/
        {
                *loc=root;
                *par=NULL;
                return;
        }
        /*Initialize ptr and ptrsave*/
        if(item<root->info)
                ptr=root->lchild;
```

```
        else
                ptr=root->rchild;
        ptrsave=root;

        while(ptr!=NULL)
        {
                if(item==ptr->info)
                {       *loc=ptr;
                         *par=ptrsave;
                         return;
                }
                ptrsave=ptr;
                if(item<ptr->info)
                        ptr=ptr->lchild;
                else
                        ptr=ptr->rchild;
         }/*End of while */
         *loc=NULL;   /*item not found*/
         *par=ptrsave;
}/*End of find()*/

void BST::case_a(NODE par,NODE loc )
{
        if(par==NULL) /*item to be deleted is root node*/
                root=NULL;
        else
                if(loc==par->lchild)
                        par->lchild=NULL;
                else
                        par->rchild=NULL;
}/*End of case_a()*/

void BST::case_b(NODE par,NODE loc)
{
        NODE child;

        /*Initialize child*/
        if(loc->lchild!=NULL) /*item to be deleted has lchild */
                child=loc->lchild;
        else            /*item to be deleted has rchild */
                child=loc->rchild;
```

```
        if(par==NULL )   /*Item to be deleted is root node*/
                root=child;
        else
                if( loc==par->lchild)   /*item is lchild of its parent*/
                        par->lchild=child;
                else                    /*item is rchild of its parent*/
                        par->rchild=child;
}/*End of case_b()*/

void BST::case_c(NODE par,NODE loc)
{
        NODE ptr,ptrsave,suc,parsuc;

        /*Find inorder successor and its parent*/
        ptrsave=loc;
        ptr=loc->rchild;
        while(ptr->lchild!=NULL)
        {
                ptrsave=ptr;
                ptr=ptr->lchild;
        }
        suc=ptr;
        parsuc=ptrsave;

        if(suc->lchild==NULL && suc->rchild==NULL)
                case_a(parsuc,suc);
        else
                case_b(parsuc,suc);

        if(par==NULL) /*if item to be deleted is root node */
                root=suc;
        else
                if(loc==par->lchild)
                        par->lchild=suc;
                else
                        par->rchild=suc;

        suc->lchild=loc->lchild;
        suc->rchild=loc->rchild;
}/*End of case_c()*/

//This function will insert an element to the tree
```

```
void BST::insert(int item)
{       NODE tmp,parent,location;
        find(item,&parent,&location);
        if(location!=NULL)
        {
                cout<<"\nItem already present";
                getch();
                return;
        }
        //creating new node to insert
        tmp=(NODE)new(struct node);
        tmp->info=item;
        tmp->lchild=NULL;
        tmp->rchild=NULL;

        if(parent==NULL)
                root=tmp;
        else
                if(item<parent->info)
                        parent->lchild=tmp;
                else
                        parent->rchild=tmp;
}/*End of insert()*/

//Function to delete a node
void BST::del(int item)
{
        NODE parent,location;
        if(root==NULL)
        {
                cout<<"\nTree is empty";
                getch();
                return;
        }

        find(item,&parent,&location);
        if(location==NULL)
        {
                cout<<"\nItem not present in tree";
                return;
        }
```

```
        if(location->lchild==NULL && location->rchild==NULL)
                case_a(parent,location);
        if(location->lchild!=NULL && location->rchild==NULL)
                case_b(parent,location);
        if(location->lchild==NULL && location->rchild!=NULL)
                case_b(parent,location);
        if(location->lchild!=NULL && location->rchild!=NULL)
                case_c(parent,location);
        delete(location);
}/*End of del()*/

//Function to traverse in a preorder fashion
void BST::preorder(NODE ptr)
{
        if(root==NULL)
        {
                cout<<"\nTree is empty";
                getch();
                return;
        }
        if(ptr!=NULL)
        {
                cout<<"  "<<ptr->info;
                preorder(ptr->lchild);
                preorder(ptr->rchild);
        }
}/*End of preorder()*/

//Function for Inorder traversal
void BST::inorder(NODE ptr)
{
        if(root==NULL)
        {
                cout<<"Tree is empty";
                getch();
                return;
        }
        if(ptr!=NULL)
        {
                inorder(ptr->lchild);
                cout<<"  "<<ptr->info;
                inorder(ptr->rchild);
```

```
        }
}

//This function will travel in a postorder fashion
void BST::postorder(NODE ptr)
{
        if(root==NULL)
        {
                cout<<"\nTree is empty";
                getch();
                return;
        }
        if(ptr!=NULL)
        {
                postorder(ptr->lchild);
                postorder(ptr->rchild);
                cout<<"  "<<ptr->info;
        }
}/*End of postorder()*/

//Function to display all the nodes of the tree
void BST::display(NODE ptr,int level)
{
        int i;
        if ( ptr!=NULL )
        {
                display(ptr->rchild, level+1);
                cout<<"\n";
                for (i = 0; i < level; i++)
                        cout<<"    ";
                cout<<ptr->info;
                display(ptr->lchild, level+1);
        }/*End of if*/
}/*End of display()*/

void main()
{
        int choice,num;
        BST bo;
        while(1)
        {
                clrscr();
```

```
//Menu options
cout<<"\n1.Insert\n";
cout<<"2.Delete\n";
cout<<"3.Inorder Traversal\n";
cout<<"4.Preorder Traversal\n";
cout<<"5.Postorder Traversal\n";
cout<<"6.Display\n";
cout<<"7.Quit\n";
cout<<"\nEnter your choice : ";
cin>>choice;

switch(choice)
{
 case 1:
       cout<<"\nEnter the number to be inserted : ";
       cin>>num;
       bo.insert(num);
       break;
 case 2:
       cout<<"\nEnter the number to be deleted : ";
       cin>>num;
       bo.del(num);
       break;
 case 3:
       bo.inorder(bo.root);
       getch();
       break;
 case 4:
       bo.preorder(bo.root);
       getch();
       break;
 case 5:
       bo.postorder(bo.root);
       getch();
       break;
 case 6:
       bo.display(bo.root,1);
       getch();
       break;
 case 7:
       exit(0);
 default:
```

```
                cout<<"\nWrong choice\n";
                getch();
            }/*End of switch */
        }/*End of while */
}/*End of main()*/
```

8.8. THREADED BINARY TREE

Traversing a binary tree is a common operation and it would be helpful to find more efficient method for implementing the traversal. Moreover, half of the entries in the Lchild and Rchild field will contain NULL pointer. These fields may be used more efficiently by replacing the NULL entries by special pointers which points to nodes higher in the tree. Such types of special pointers are called threads and binary tree with such pointers are called threaded binary tree.

Fig. 8.20 shows the threaded binary tree with threads replacing NULL pointer of binary tree in Fig. 8.13. The threads are drawn with dotted lines to differentiate then from tree pointers.

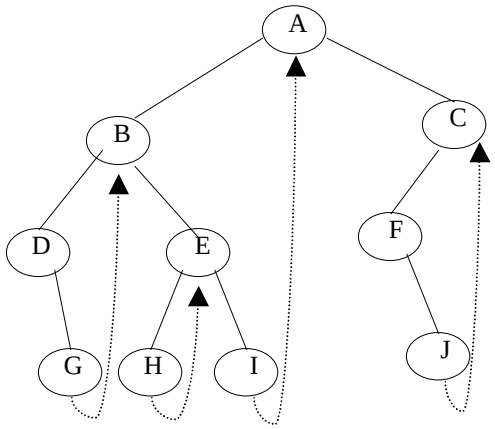


Fig. 8.20. Threaded binary tree

There are many ways to thread a binary tree. Right most nodes in the threaded binary tree have a NULL right pointer (*i.e.*, in-order successor). Such trees are called right in threaded binary trees. A left in threaded binary tree may be defined similarly as one in which each NULL left pointer is altered to contain a thread (*i.e.*, in-order predecessor). An in-threaded binary tree may be defined as a binary tree that is both left-in-threaded and right-in-threaded.

We can implement a right in threaded binary tree using arrays by distinguishing threads from ordinary pointers. Threads are denoted by negative numbers, when ordinary pointers are denoted by positive integers. The array representation of the right in thread binary tree in Fig.8.20 is shown below table (Fig. 8.21)

	Info	*Lchild*	*Rchild*
A[0]	A	1	2
A[1]	B	3	4
A[2]	C	5	
A[3]	D		8
A[4]	E	9	10
A[5]	F		12
A[6]			
A[7]			
A[8]	G		– 1
A[9]	H		– 4
A[10]	I		– 0
A[11]			
A[12]	J		– 2
A[13]			
A[14]			

Fig. 8.21

To implement a right-in-threaded binary tree using dynamic memory allocation, an extra 1 bit logical field, rthread, is used to distinguish threads from ordinary pointers. If a right pointer of a node is threaded, then the rthread = TRUE otherwise FALSE. Following program will construct a right-in-threaded binary tree and will traverse in-order fashion.

8.9. EXPRESSION TREE

An ordered tree may be used to represent a general expressions, is called expression tree. Nodes in one expression tree contain operators and operands.

For example: The expression, f(A,B,C) * (Sin(D) – Log(E * F!)), is represented in Fig. 8.22.

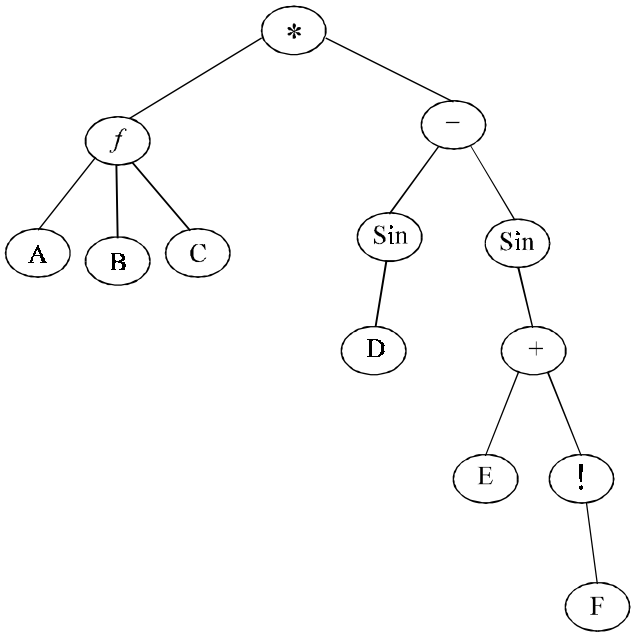


Fig. 8.22

A binary tree is used to represent a binary expression called binary expression tree. Root node of the binary expressions trees contains operator and two children node contain its operand.

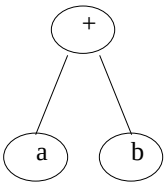


Fig. 8.23

For example, a+b can be represented in Fig. 8.23. And the expression E = (a + b) * (c / (d – e)))with more operators and operands) can be represented in binary expression tree as follows :

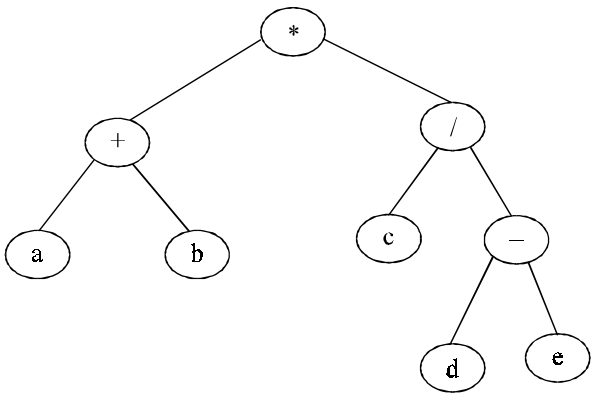


Fig. 8.24

Expression tree can be evaluated, if its operands are numerical constants. Following section will explain a C/C++ program that accepts a pointer of an expression tree and returns the value of the expression represented by the tree.

8.10. DECISION TREE

A decision tree is a binary tree, where sorting of elements is done by only comparisons. That is a decision tree can be applied to any sorting algorithms that sorts by using comparisons. (The sorting techniques were discussed in chapter 6). The external nodes (or leafs) correspond to the *n*! ways that *n* items can appear, because we are trying to sort *n* items *a*1, *a*2, ..., *an*. And internal nodes correspond to the different comparison that may take place during the execution. The decision tree is Fig. 8.25 represents an algorithm that sorts the tree elements *x*, *y* and *z*. The first comparison prefers at the root node between *x* and *y* and goes down.

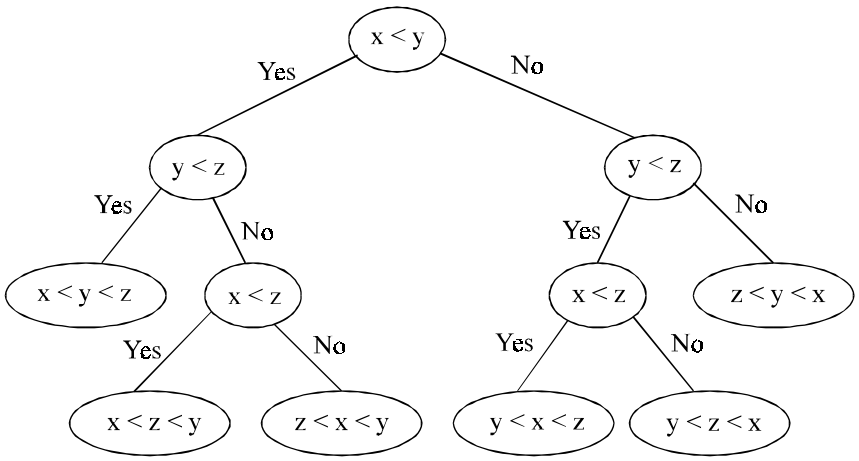


Fig. 8.25

The sorting techniques discussed in chapter 6 takes minimum of 0(*n* log *n*) to sort an array of *n* elements. The objective of decision tree is to develop an algorithm, which can sort *n* items of the order less than 0(*n* Log *n*).

The decision tree is more suitable when extremely small input size is to be sorted. The number at comparisons in the worst case is equal to the depth of the deepest leaf (*i.e.*, the largest path). In Fig. 8.25, maximum of three comparisons used, which is a worst case *i.e.*, O(*n*) and it is less than O(*n* log *n*). Moreover, in the average case, the average number(s) of comparisons are sufficient to sort the elements, which is equal to the average external path length of the tree (*i.e.*, average depth of the leaves).

8.11. FIBANOCCI TREE

Fibanocci tree of order *n* is a binary tree, which build by the following restriction :

1. If $n = 0$, then the empty tree is a fibanocci tree of order 0.
2. If $n = 1$, then the tree with a single node is a fibanocci tree of order 1.
3. If $n > 1$, the tree consists of a root with a left subtree of order $n - 1$ and right sub tree of order $n - 2$.

Here are some exmples of fibanocci tree

Fig. 8.26. Fibanocci tree of order 0

Fig. 8.27. Fibanocci tree of order 1

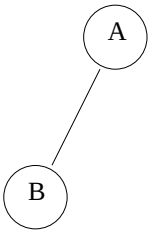


Fig. 8.28. Fibanocci tree of order 2

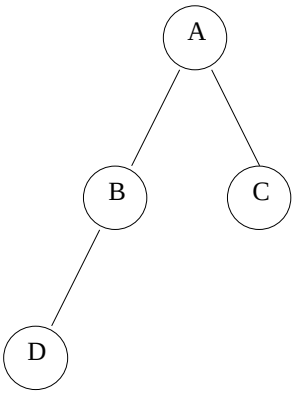


Fig. 8.29. Fibanocci tree of order 3

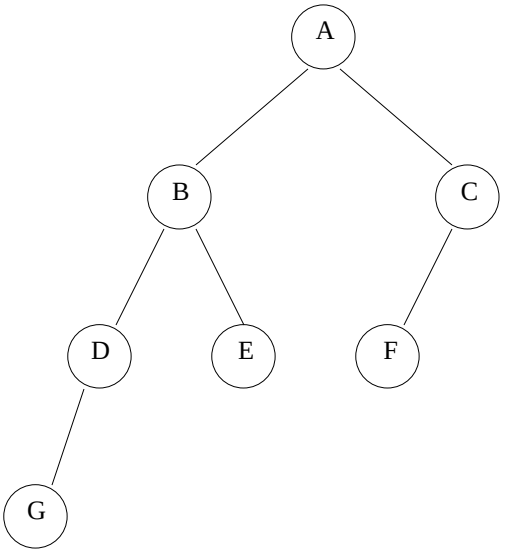


Fig. 8.30. Fibanocci tree of order 4

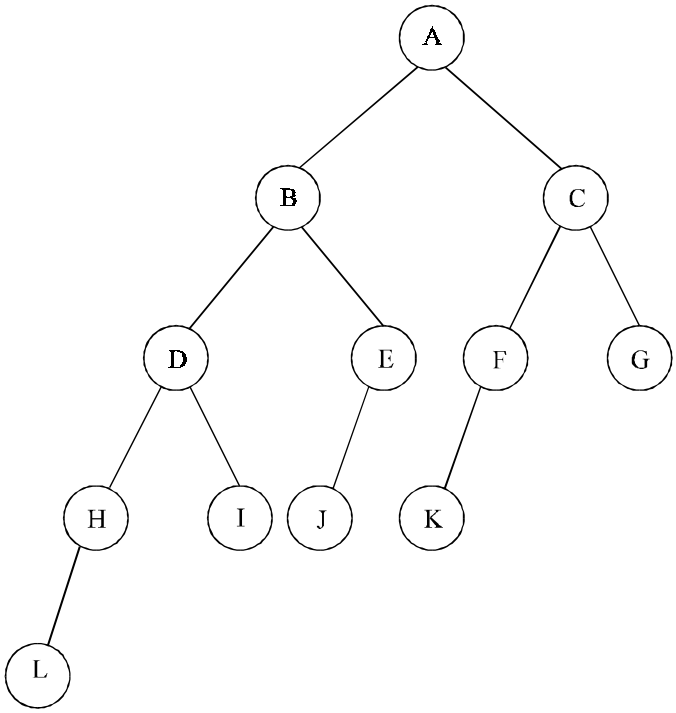


Fig 8.31. Fibanocci tree of order 5

8.12. SELECTION TREES

Suppose we have *k* ordered set of arrays, called runs, which are to be merged into a single ordered array. Each run consists of some elements and is in ascending order. The merging task can be accomplished by repeatedly outputting the smallest element from the *k* runs. The most general way to merge *k* runs is to make $(k - 1)$ comparisons to output the smallest element, (from the *k* runs) in every iteration. By using the data structure selection tree, we can reduce the number of comparisons needed to find the next smallest element. There are two kinds of selection trees :

(*a*) Winner trees

(*b*) Loser trees

8.12.1. WINNER TREES

A Winner Tree is a complete binary tree in which each node represents the smallest of its children. Thus the root node represents the smallest node in the tree, which is the next element in the merged single ordered array.

The construction of the winner tree may be compared to the playing of a tennis tournament. Then, each non-leaf node in the tree represents the winner of a tournament and root node represents the over all winner. Winner tree can be illustrated with the following example. Here we want to merge 8 runs into single sorted array.

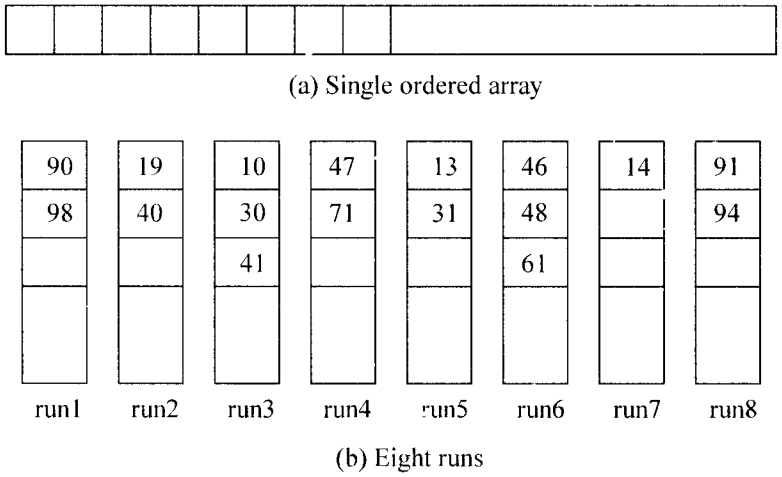


Fig. 8.32

Create a complete binary tree with 8 empty leaves (For Merging 8 run). Place the 1st element in the runs to leaves.

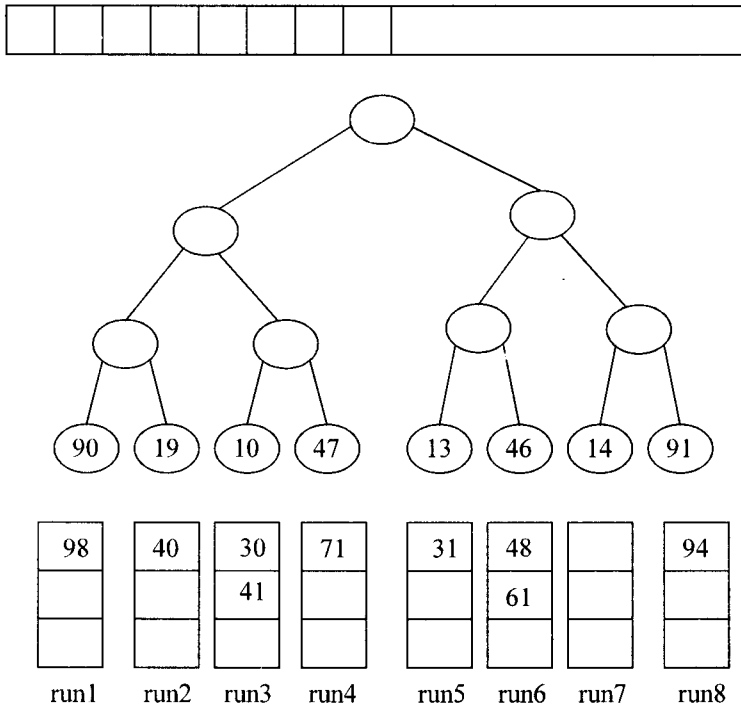


Fig. 8.33

Each of the empty nodes represents the smallest elements of its children. That is, since (19 < 90), place 19 at the father node etc.

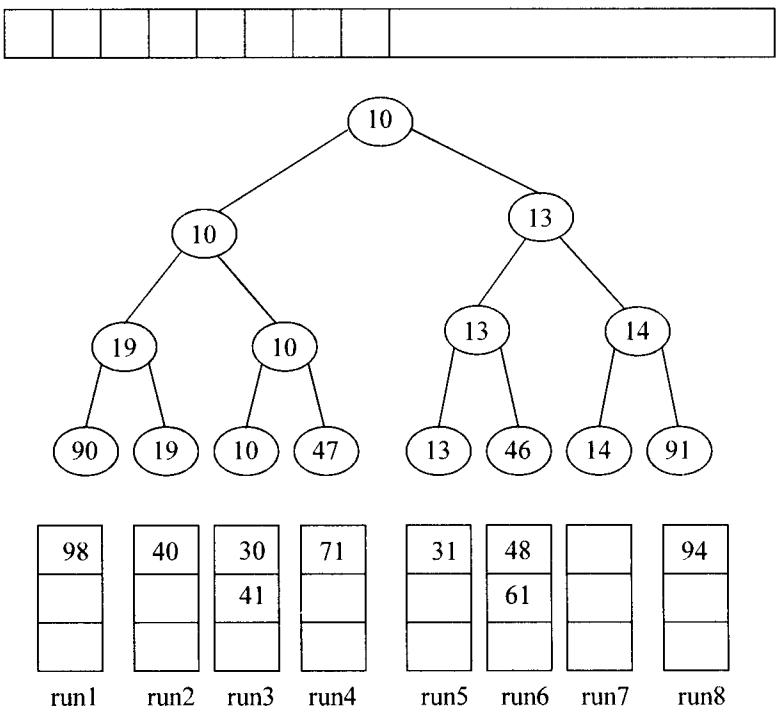


Fig. 8.34

Place 10, the root node, into the single ordered array. And replace the leaf node 10 with the next element in the corresponding run (*i.e.*, 30).

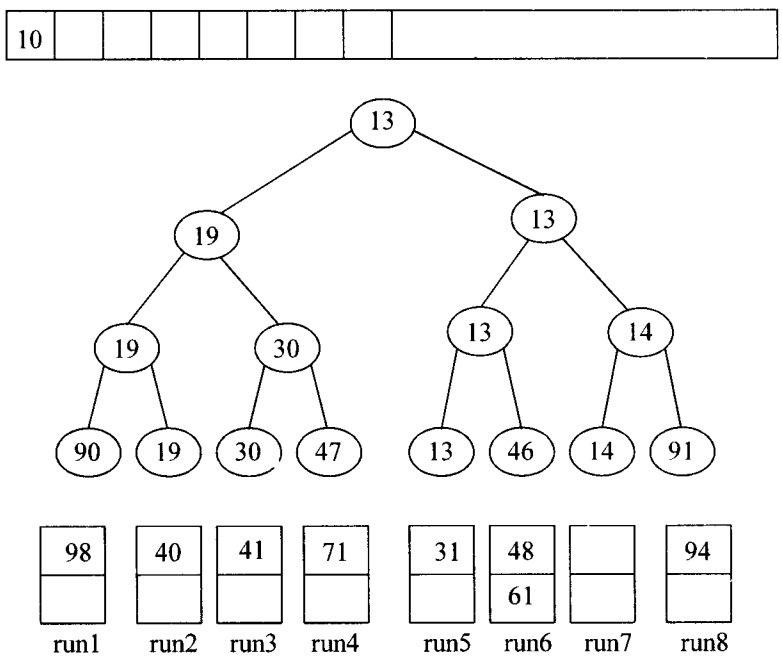


Fig. 8.35

Here we have restructured the tree by replacing along the path. That is we have located the smallest element (The winner) with just 3 comparisons. And repeat the process.

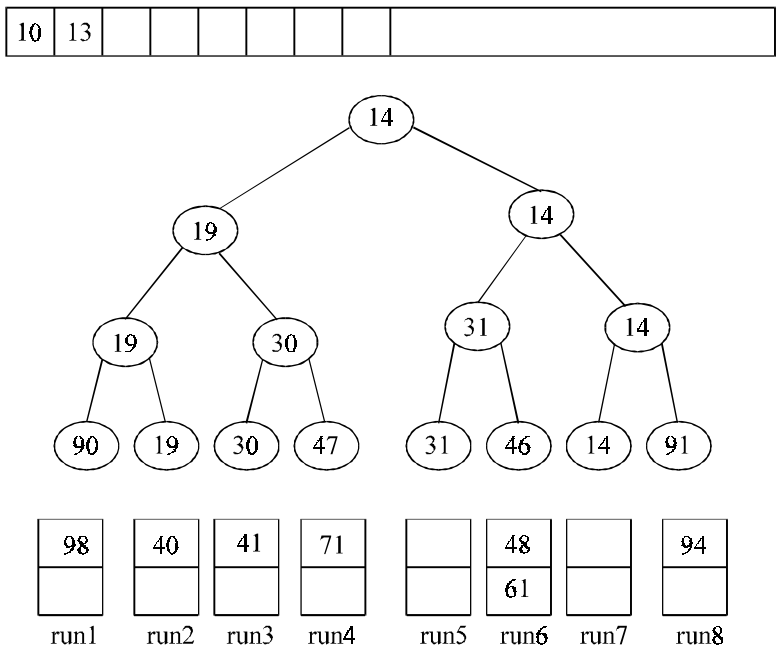


Fig. 8.36

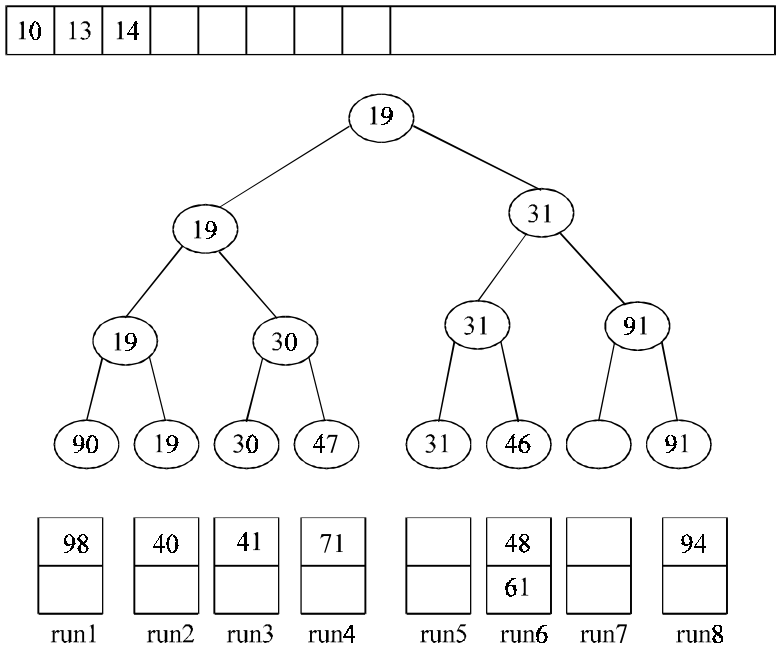


Fig. 8.37

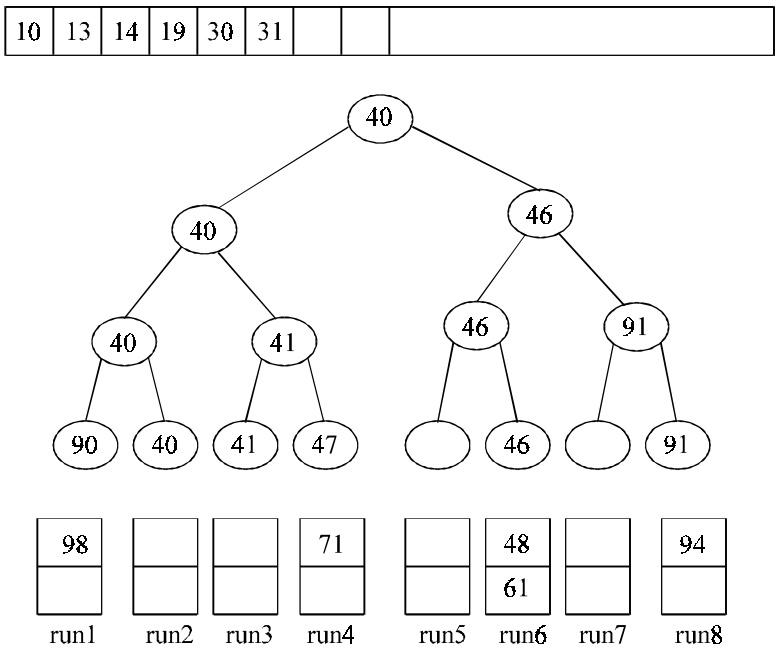


Fig. 8.38

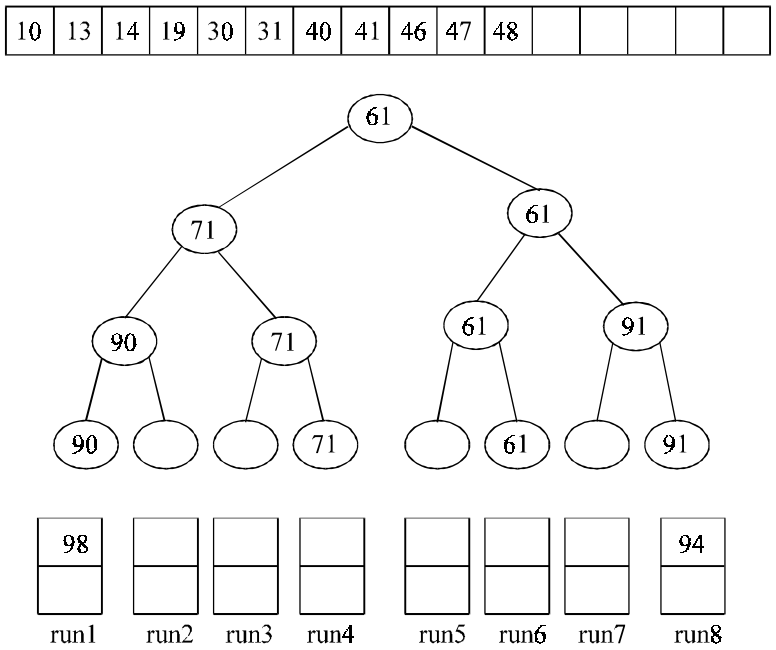


Fig. 8.39

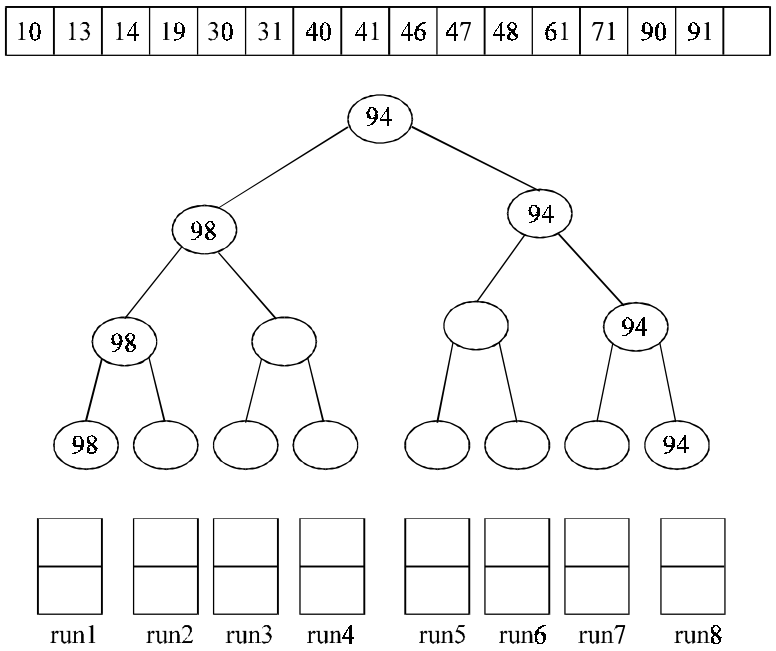


Fig. 8.40

The final merged sorted array of 8 runs is, 10, 13, 14, 19, 30, 31, 40, 41, 46, 47, 48, 61, 71, 90, 94, 98.

8.12.2. LOSER TREES

The restructuring process in winner tree can be further simplified by placing in each non-leaf node of the loser instead of winner. A selection tree in which each non-leaf node retains the information of the loser is called a loser tree. Fig. 8.41 shows the loser tree that corresponds to the winner tree of Fig. 8.34.

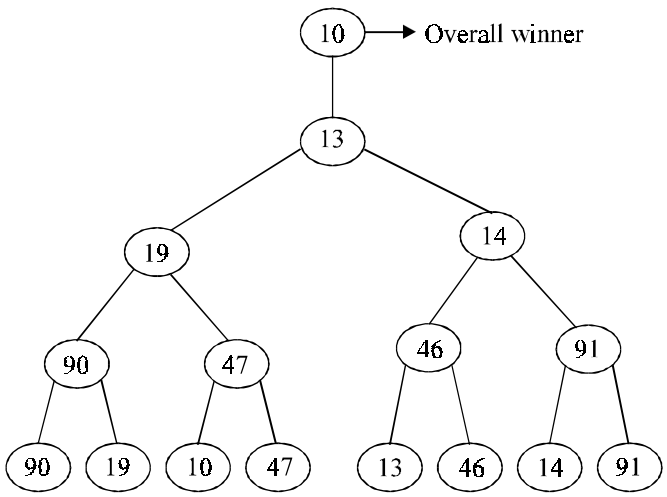


Fig. 8.41

8.13. BALANCED BINARY TREES

A balanced binary tree is one in which the largest path through the left sub tree is the same length as the largest path of the right sub tree, *i.e.*, from root to leaf. Searching time is very less in balanced binary trees compared to unbalanced binary tree. *i.e.*, balanced trees are used to maximize the efficiency of the operations on the tree. There are two types of balanced trees :

1. Height Balanced Trees
2. Weight Balanced Trees

8.13.1. HEIGHT BALANCED TREES

In height balanced trees balancing the height is the important factor. There are two main approaches, which may be used to balance (or decrease) the depth of a binary tree :

(*a*) Insert a number of elements into a binary tree in the usual way, using the algorithm given in the previous section (*i.e.*, Binary search Tree insertion). After inserting the elements, copy the tree into another binary tree in such a way that the tree is balanced. This method is efficient if the data(s) are continually added to the tree.

(*b*) Another popular algorithm for constructing a height balanced binary tree is the AVL tree, which is discussed in the next section.

8.13.2. WEIGHT BALANCED TREE

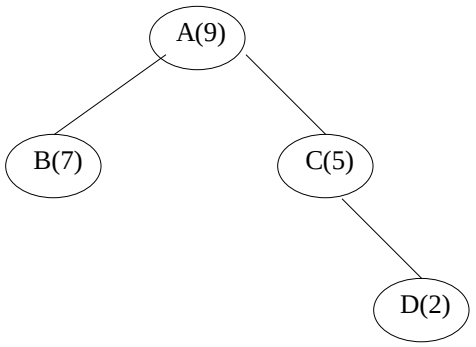


Fig. 8.42

A weight-balanced tree is a balanced binary tree in which additional weight field is also there. The nodes of a weight-balanced tree contain four fields :

(*i*) Data Element

(*ii*) Left Pointer

(*iii*) Right Pointer

(*iv*) A probability or weight field

The data element, left and right pointer fields are save as that in any other tree node. The probability field is a specially added field for a weight-balanced tree. This field holds the probability of the node being accessed again, that is the number of times the node has been previously searched for.

When the tree is created, the nodes with the highest probability of access are placed at the top. That is the nodes that are most likely to be accessed have the lowest search time. And the tree is balanced if the weights in the right and left sub trees are as equal as possible. The average length of search in a weighted tree is equal to the sum of the probability and the depth for every node in the tree.

The root node contain highest weighted node of the tree or sub tree. The left sub tree contains nodes where data values are less than the current root node, and the right sub tree contain the nodes that have data values greater than the current root node.

8.14. AVL TREES

This algorithm was developed in 1962 by two Russian Mathematicians, G.M. Adel'son Vel'sky and E.M. Landis; here the tree is called AVL Tree. An AVL tree is a binary tree in which the left and right sub tree of any node may differ in height by at most 1, and in which both the sub trees are themselves AVL Trees. Each node in the AVL Tree possesses any one of the following properties :

(*a*) A node is called left heavy, if the largest path in its left sub tree is one level larger than the largest path of its right sub tree.

(*b*) A node is called right heavy, if the largest path in its right sub tree is one level larger than the largest path of its left sub tree.

(*c*) The node is called balanced, if the largest paths in both the right and left sub trees are equal. Fig. 8.37 shows some example for AVL trees.

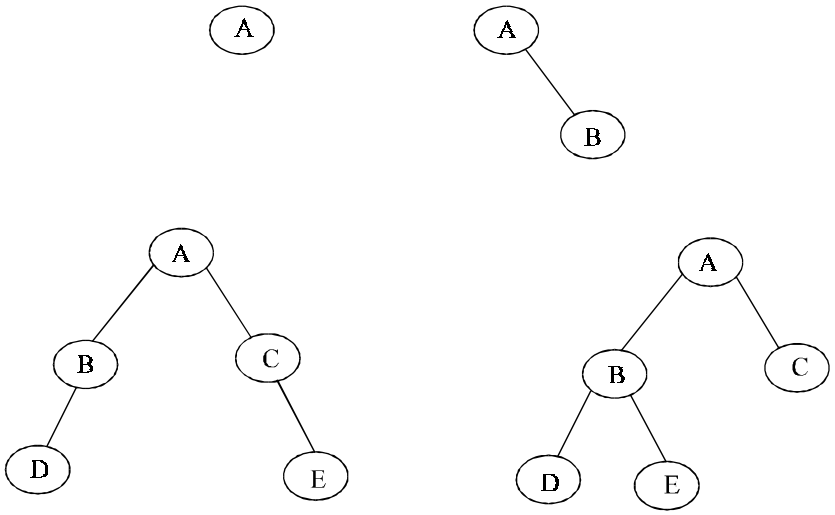


Fig. 8.43. AVL tree

The construction of an AVL Tree is same as that of an ordinary binary tree except that after the addition of each new node, a check must be made to ensure that the AVL balance conditions have not been violated. If the new node causes an imbalance in the tree, some rearrangement of the tree's nodes must be done. Following algorithm will insert a new node in an AVL Tree :

ALGORITHM

1. Insert the node in the same way as in an ordinary binary tree.
2. Trace a path from the new nodes, back towards the root for checking the height difference of the two sub trees of each node along the way.
3. Consider the node with the imbalance and the two nodes on the layers immediately below.
4. If these three nodes lie in a straight line, apply a *single rotation* to correct the imbalance.
5. If these three nodes lie in a dogleg pattern (*i.e.*, there is a bend in the path) apply a *double rotation* to correct the imbalance.
6. Exit.

The above algorithm will be illustrated with an example shown in Fig. 8.44, which is an unbalance tree. We have to apply the rotation to the nodes 40, 50 and 60 so that a balance tree is generated. Since the three nodes are lying in a straight line, single rotation is applied to restore the balance.

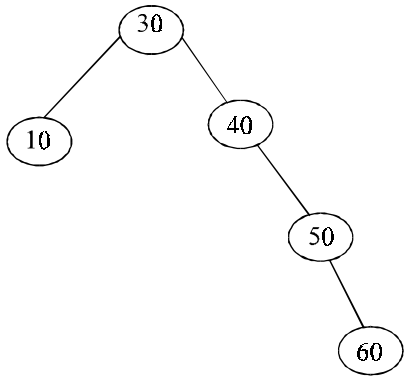


Fig. 8.44

Fig. 8.45 is a balance tree of the unbalanced tree in Fig. 8.44. Consider a tree in Fig. 8.46 to explain the double rotation.

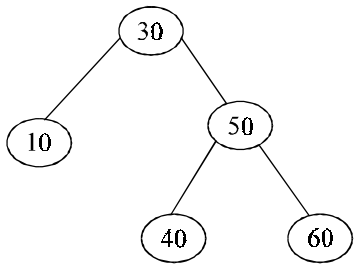


Fig. 8.45

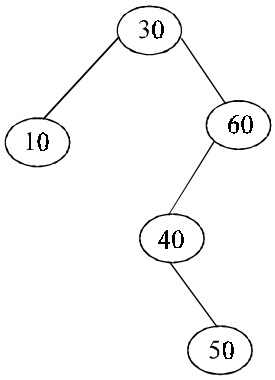


Fig. 8.46

While tracing the path, first imbalance is detected at node 60. We restrict our attention to this node and the two nodes immediately below it (40 and 50). These three nodes form a dogleg pattern. That is there is bend in the path. Therefore we apply double rotation to correct the balance. A double rotation, as its name implies, consists of two single rotations, which are in opposite direction. The first rotation occurs on the two layers (or levels) below the node where the imbalance is found (*i.e.,* 40 and 50). Rotate the node 50 up by replacing 40, and now 40 become the child of 50 as shown in Fig. 8.47.

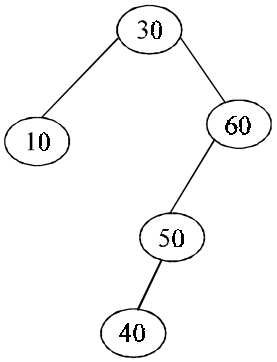


Fig. 8.47

Apply the second rotation, which involves the nodes 60, 50 and 40. Since these three nodes are lying in a straight line, apply the single rotation to restore the balance, by replacing 60 by 50 and placing 60 as the right child of 50 as shown in Fig. 8.48.

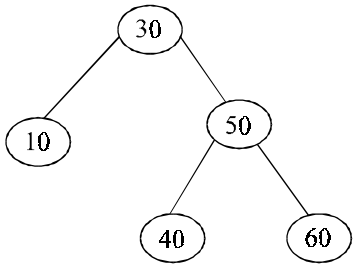


Fig. 8.48

Balanced binary tree is a very useful data structure for searching the element with less time. An unbalanced binary tree takes O(*n*) time to search an element from the tree, in the worst case. But the balanced binary tree takes only O(Log *n*) time complexity in the worst case.

8.15. M-WAY SEARCH TREES

Trees having (m–1) keys and *m* children are called *m*-way search trees. A binary tree is a 2-way tree. It means that it has $m - 1 = 2 - 1 = 1$ key (here $m = 2$) in every node and it can have maximum of two children. A binary tree is also called an *m*-way tree of order 2.

Similarly an *m*-way tree of order 3 is a tree in which key values could be either 1 or 2 (*i.e.*, inside every node and it can have maximum of two children). For example an *m*-way tree at order (degree) 4 is shown in Fig. 8.49.

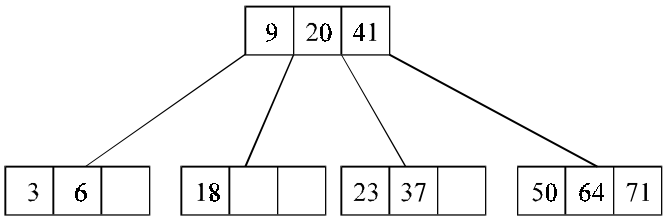


Fig. 8.49

8.16. 2-3 TREES

Every node in the 2-3 trees has two or three children. A 2-3 tree is a tree in which leaf nodes are the only nodes that contains data values. All non-leaf nodes contain two values of the sub trees. All the leaf nodes have the same path length from the root node. Fig. 8.50 show a typical 2-3 tree.

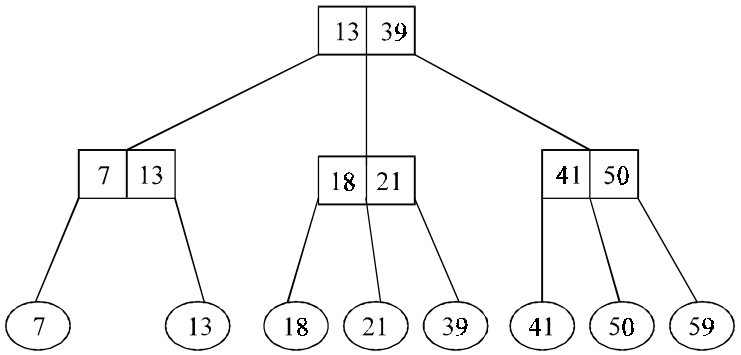


Fig. 8.50

The first value in the non-leaf root node is the maximum of all the leaf values in the left sub tree. The second value is the maximum value of the middle sub tree. With this root node information following conclusion can be obtained :

1. Every leaf node in the left sub tree of any non-leaf node is equal to or less than the first value.

2. Every leaf node in the middle sub tree is less than or equal to the second value and greater than the first value.

3. Every leaf node in the right sub tree is greater than the second value

Following figures will illustrate the construction (or insertion) of the 2-3 trees :

Fig. 8.51. Insert 13

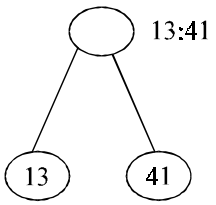


Fig. 8.52. Insert 41

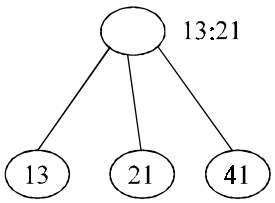


Fig. 8.53. Insert 21

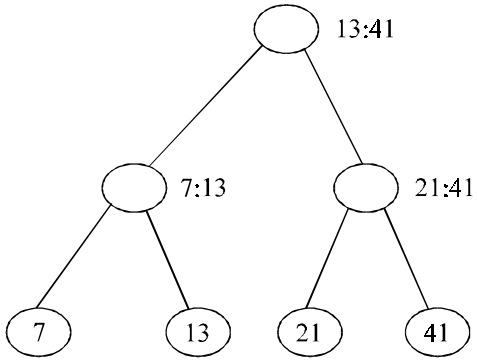


Fig. 8.54. Insert 7

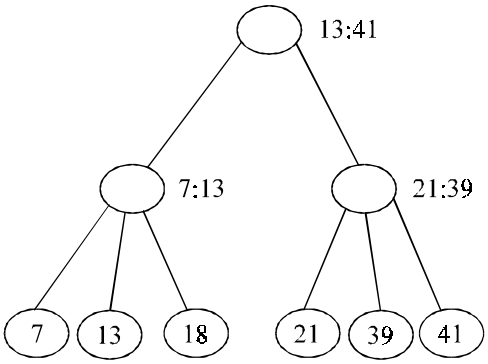


Fig. 8.55. Insert 18, 39

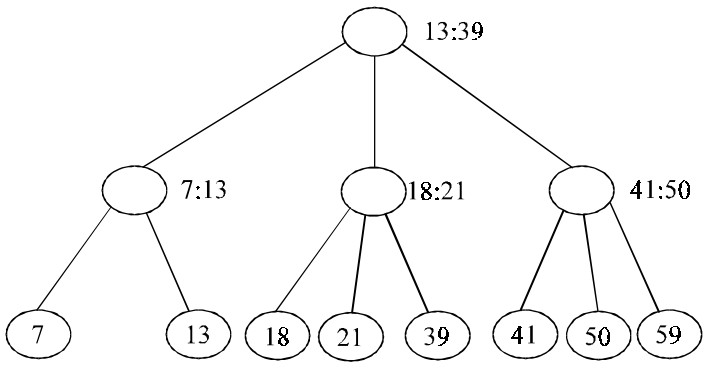


Fig. 8.56. Insert 50, 59

8.17. 2-3-4 TREES

A 2-3-4 tree is an extension of a 2-3 tree. Every node in the 2-3-4 trees can have maximum of 4 children. A typical 2-3-4 tree is shown in Fig. 8.57.

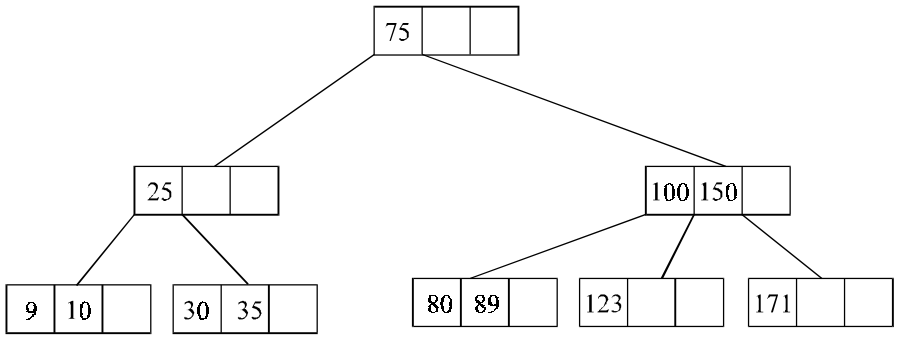


Fig. 8.57

A 2-3-4 tree is a search tree that is either empty or satisfies the following properties.

1. Every internal node is a 2, 3, or 4 node. A 2 node has one element, a 3 node has two elements and a 4 node has three elements.
2. Let LeftChild and RightChild denote the children of a 2 node and Data be the element in this node. All the elements in LeftChild have the elements less than the data, and all elements in the RightChild have the elements greater than the Data.
3. Let LeftChild, MidChild and RightChild denote the children of a 3 node and LeftData and RightData be the element in this node. All the elements in LeftChild have the elements less than the LeftData, all the elements in the MidChild have the element greater than LeftData but less than RightData, and all the elements in the RightChild have the elements greater than Right Data.
4. Let LeftChild, LeftMidChild, RightMidChild, and RightChild denote the children of a 4 node. Let LeftData, MidData and RightData be the three elements in this node. Then LeftData is less than MidData and it is less than RightData. All the elements in the LeftChild is less than LeftData, all the elements in the Left MidChild is less than MidData but greater than LeftData, all the elements in RightMidChild is less than RightData but greater than MidData, and all the elements in RightChild is greater than RightData.
5. All external nodes are at the same level.

Consider a tree in Fig. 8.58. If we want to insert an element, 77, top-down insertion method is used by splitting the root, Fig. 8.59.

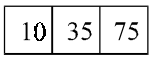


Fig. 8.58

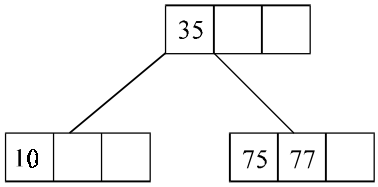


Fig. 8.59

If a 2-3-4 tree of highest h has only 2 nodes, then it contains 2_h–1 elements. If it contains only 4 nodes, then the number of elements is 4_h-1. A 2-3-4 tree is height h with a mixture of 2, 3 and 4 nodes, has between 2_h–1 and 4_h–1 elements. In other words, the height of a 2-3-4 with n elements is between log4(n + 1) and log2(n + 1). A 2-3-4 tree can be represented efficiently as a binary tree called a red-black tree, which will be discussed in the next section.

8.18. RED-BLACK TREE

A red-black tree is a balanced binary search tree with the following properties :

1. Every node is colored red or black.

2. Every leaf is a NULL node, and is colored black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

The red-black tree algorithm is a method for balancing trees. The name derives from the fact that each node is colored red or black, and the color of the node is instrumental in determining the balance of the tree. During insert and delete operations, nodes may be rotated to maintain tree balance. Both average and worst-case search time is O(log *n*).

We classify red-black trees according to the order *n*, the number of internal nodes. In the Fig. 8.60, *n* = 6. Among all red-black trees of height 4, this is one with the minimum number of nodes. A red-black tree with *n* nodes has height at least O(log *n*) and at most 2O(log *n* + 1).

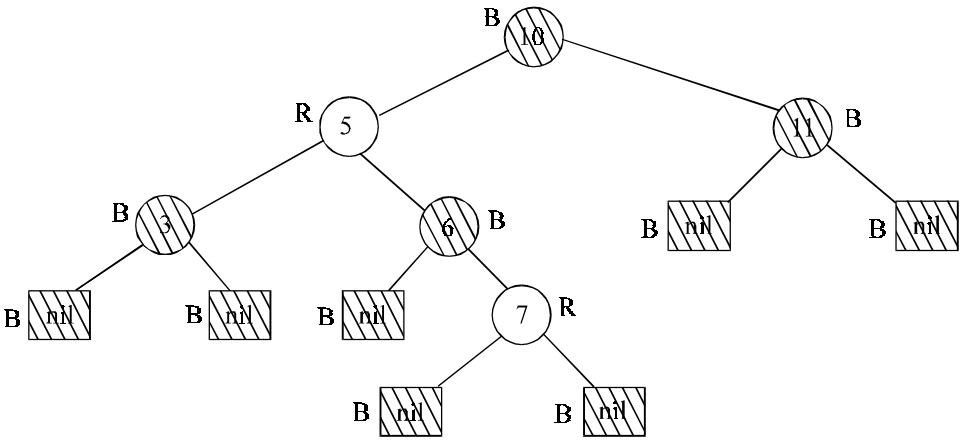


Fig. 8.60

To insert a node, we search the tree for an insertion point, and add the node to the tree. A new node replaces an existing NULL node at the bottom of the tree, and has two NULL nodes as children. In the implementation, a NULL node is simply a pointer to a common *sentine*l node that is colored black. After insertion, the new node is colored red. Then the parent of the node is examined to determine if the red-black tree properties have been violated. If necessary, we recolor the node and do rotations to balance the tree.

By inserting a red node with two NULL children, we have preserved black-height property (property 4). However, property 3 may be violated. This property states that both children of a red node must be black. Although both children of the new node are black (they're NULL), consider the case where the parent of the new node is red. Inserting a red node under a red parent would violate this property. There are two cases to consider:

- *Red parent, red uncle*: Fig. 8.61 illustrates a red-red violation. Node X is the newly inserted node, with both parent and uncle colored red. A simple recoloring removes the red-red violation. After recoloring, the grandparent (node B) must be checked for validity, as its parent may be red. Note that this has the effect of propagating a red node up the tree. On completion, the root of the tree is marked black. If it was originally red, then this has the effect of increasing the black-height of the tree.

- *Red parent, black uncle:* Fig. 8.62 illustrates a red-red violation, where the uncle is colored black. Here the nodes may be rotated, with the subtrees adjusted as shown. At this point the algorithm may terminate as there are no red-red conflicts and the top of the subtree (node A) is colored black. Note that if node X was originally a right child, a left rotation would be done first, making the node a left child.

Each adjustment made while inserting a node causes us to travel up the tree one step. At most 1 rotation (2 if the node is a right child) will be done, as the algorithm terminates in this case. The technique for deletion is similar.

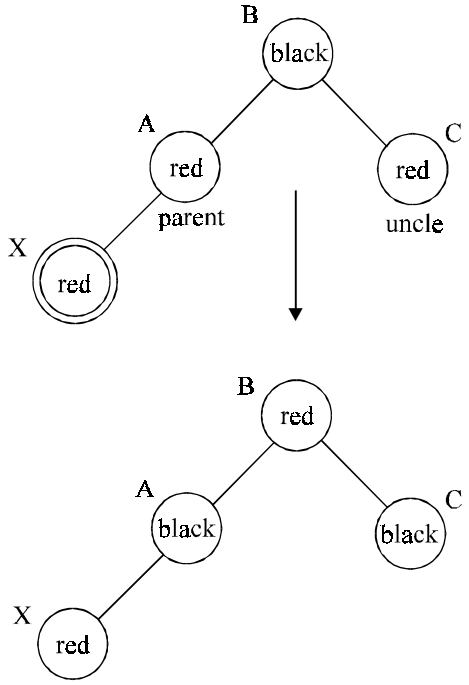
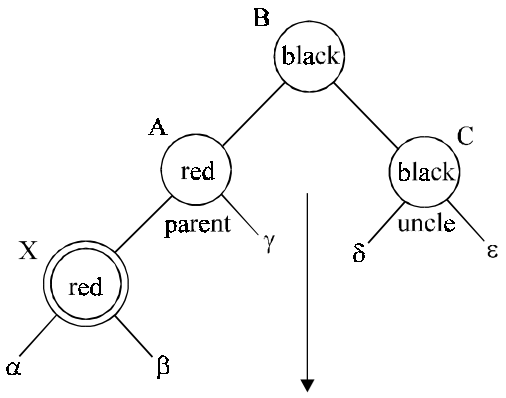


Fig. 8.61



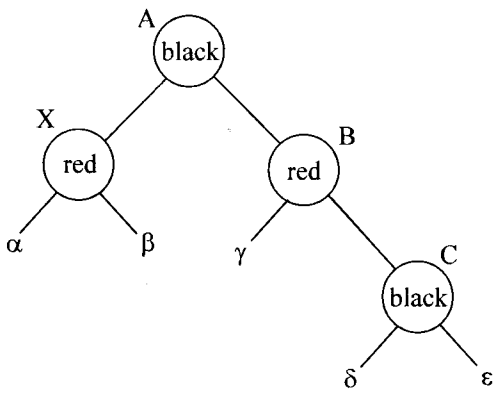


Fig. 8.62

8.19. B-TREE

B-trees are tree data struture that are most commonly found in databases and file systems. B-trees keep data sorted and allow amortized logarithmic time insertions and deletions. B-trees generally grow from the bottom up as elements are inserted, whereas most binary tree grow down.

B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes. This usually occurs when most nodes are in secondary storage such as hard drive. By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often, and efficiency increases. Usually this value is set such that each node takes up a full disk block or an analogous size in secondary storage.

The idea behind B-trees is that inner nodes can have a variable number of child nodes within some pre-defined range. Hence, B-trees do not need re-balancing as frequently as other self-balancing binary search trees. The lower and upper bounds on the number of child nodes are fixed for a particular implementation. For example, in a 2-3 B-tree (often simply 2-3 tree), each internal node may have only 2 or 3 child nodes. A node is considered to be in an illegal state if it has an invalid number of child nodes.

The B-tree's creator, Rudolf Bayer, has not explained what the B stands for. The most common belief is that B stands for *balanced*, as all the leaf nodes are at the same level in the tree. B may also stand for *Bayer*, or for *Boeing*, because he was working for *Boeing Scientific Research Labs*.

Fig. 8.63 illustrates a B-tree with 3 keys/node. Keys in internal nodes are surrounded by pointers, or record offsets, to keys that are less than or greater than, the key value. For example, all keys less than 22 are to the left and all keys greater than 22 are to the right.

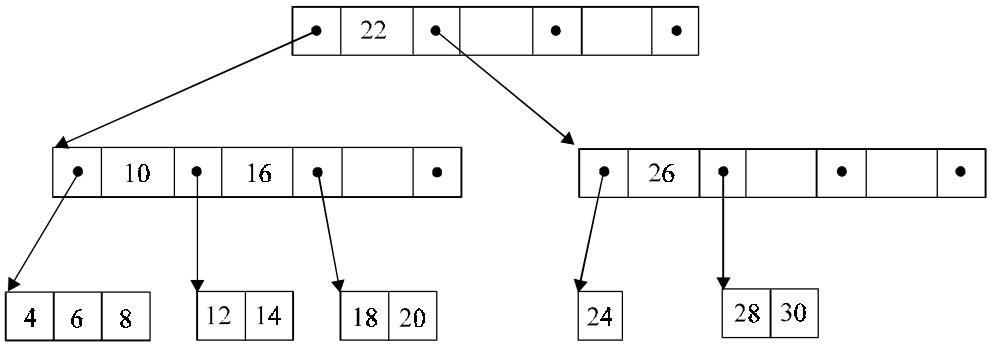


Fig. 8.63. B-tree

Nodes in a B-Tree are usually represented as an ordered set of elements and child pointers. Every node but the root contains a minimum of *m* elements, a maximum of *n* elements, and a maximum of *n* + 1 child pointers, for some arbitrary *m* and *n*. For all internal nodes, the number of child pointers is always one more than the number of elements. Since all leaf nodes are at the same height, nodes do not generally contain a way of determining whether they are leaf or internal.

Each inner node's elements act as separation values which divide its subtrees. For example, if an inner node has three child nodes (or subtrees) then it must have two separation values or elements a_1 and a_2. All values in the leftmost subtree will be less than a_1 , all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

ALGORITHMS

SEARCH

Search is performed in the typical manner, analogous to that in a binary search tree. Starting at the root, the tree is traversed top to bottom, choosing the child pointer whose separation values are on either side of the value that is being searched. Binary search is typically used within nodes to determine this location.

INSERTION

For a node to be in an illegal state, it must contain a number of elements which is outside of the acceptable range.

1. First, search for the position into which the node should be inserted. Then, insert the value into that node.
2. If no node is in an illegal state then the process is finished.
3. If some node has too many elements, it is split into two nodes, each with the minimum amount of elements. This process continues recursively up the tree until the root is reached. If the root is split, a new root is created. Typically the minimum and maximum number of elements must be selected such that the minimum is no more than one half the maximum in order for this to work.

DELETION

1. First, search for the value which will be deleted. Then, remove the value from the node which contains it.
2. If no node is in an illegal state then the process is finished.
3. If some node is in an illegal state then there are two possible cases:
   (*a*) Its sibling node, a child of the same parent node, can transfer one or more of its child nodes to the current node and return it to a legal state. If so, after updating the separation values of the parent and the two siblings the process is finished.
   (*b*) Its sibling does not have an extra child because it is on the lower bound. In that case both siblings are merged into a single node and we recurse onto the parent node, since it has had a child node removed. This continues until the current node is in a legal state or the root node is reached, upon which the root's children are merged and the merged node becomes the new root.

Several variants on the B-tree are listed in following table :

	B-Tree	*B*-Tree*	*B′-Tree*	*B″-Tree*
Data stored in	Any node	Any node	Leaf only	Leaf only
On insert, split	1 × 1 → 2 × 1/2	2 × 1 → 3 × 2/3	1 × 1 → 2 × 1/2	3 × 1 → 4 × 3/4
On delete, join	2 × 1/2 → 1 × 1	3 × 2/3 → 2 × 1	2 × 1/2 → 1 × 1	3 × 1/2 → 2 × 3/4

As we have discussed earlier the *standard* B-tree stores keys and data in both internal and leaf nodes. When descending the tree during insertion, a full child node is first redistributed to adjacent nodes. If the adjacent nodes are also full, then a new node is created, and half the keys in the child are moved to the newly created node. During deletion, children that are 1/2 full first attempt to obtain keys from adjacent nodes. If the adjacent nodes are also 1/2 full, then two nodes are joined to form one full node. B*-trees are similar, only the nodes are kept 2/3 full. This results in better utilization of space in the tree, and slightly better performance.

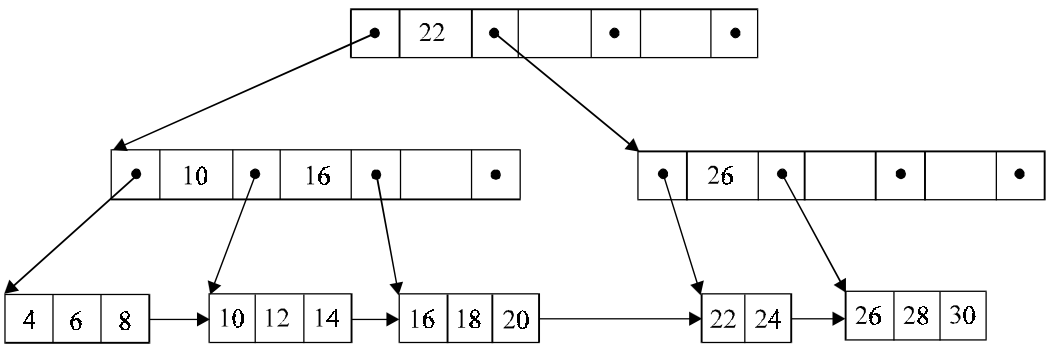


Fig. 8.64. B′-Tree

Fig. 8.64 illustrates a B^+*-tree*. All keys are stored at the leaf level, with their associated data values. Duplicates of the keys appear in internal parent nodes to guide the search. Pointers have a slightly different meaning than in conventional B-trees. The left pointer designates all keys less than the value, while the right pointer designates all keys greater than or equal to the value. For example, all keys less than 22 are on the left pointer, and all keys greater than or equal to 22 are on the right. Notice that key 22 is duplicated in the leaf, where the associated data may be found. During insertion and deletion, care must be taken to properly update parent nodes. When modifying the first key in a leaf, the tree is walked from leaf to root. The last greater than or equal to pointer found while descending the tree will require modification to reflect the new key value. Since all keys are in the leaf nodes, we may link them for sequential access.

The organization of B^{++}*-trees i*s similar to B^+-trees, except for the split/join strategy. Assume each node can hold k keys, and the root node holds $3k$ keys. Before we descend to a child node during insertion, we check to see if it is full. If it is, the keys in the child node and two nodes adjacent to the child are all merged and redistributed. If the two adjacent nodes are also full, then another node is added, resulting in four nodes, each 3/4 full. Before we descend to a child node during deletion, we check to see if it is 1/2 full. If it is, the keys in the child node and two nodes adjacent to the child are all merged and redistributed. If the two adjacent nodes are also 1/2 full, then they are merged into two nodes, each 3/4 full. This is halfway between 1/2 full and completely full, allowing for an equal number of insertions or deletions in the future.

Recall that the root node holds $3k$ keys. If the root is full during insertion, we distribute the keys to four new nodes, each 3/4 full. This increases the height of the tree. During deletion, we inspect the child nodes. If there are only three child nodes, and they are all 1/2 full, they are gathered into the root, and the height of the tree decreases.

8.20. SPLAY TREES

A splay tree is a self-balancing binary search tree with the additional unusual property that recently accessed elements are quick to access again. It performs basic operations such as insertion, search and removal in O(log(n)) amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Sleator and Robert Tarjan.

All normal operations on a splay tree are combined with one basic operation, called splaying, also called rotations. That is the efficiency of splay trees comes not from an explicit structural constraint, as with balanced trees, but from applying a simple restructuring heuristic, called splaying, whenever the tree is accessed. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a bottom-up algorithm can combine the search and the tree reorganization.

There are several ways in which splaying can be done. It always involves interchanging the root with the node in operation. One or more other nodes might change

position as well. The purpose of splaying is to minimize the number of (access) operations required to recover desired data records over a period of time.

SPLAY OPERATION

The most important tree operation is splay, also called rotation. If we apply splay rotation to splay(N), which moves an element N to the root of the tree. In case N is not present in the tree, the last element on the search path for N is moved instead.

To do a splay, we carry out a sequence of rotations, each of which moves the target node N closer to the root. Each particular step depends on only two factors:

Whether N is the left or right child of its parent node, P,

Whether P is the left or right child of its parent, G (for grandparent node).

Thus, there are four cases:

Case 1: N is the left child of P and P is the left child of G. In this case we perform a double right rotation, so that P becomes N's right child, and G becomes P's right child.

Case 2: N is the right child of P and P is the right child of G. In this case we perform a double left rotation, so that P becomes N's left child, and G becomes P's left child.

Case 3: N is the left child of P and P is the right child of G. In this case we perform a rotation so that G becomes N's left child, and P becomes N's right child.

Case 4: N is the right child of P and P is the left child of G. In this case we perform a rotation so that P becomes N's left child, and G becomes N's right child.

Finally, if N doesn't have a grandparent node, we simply perform a left or right rotation to move it to the root. By performing a splay on the node of interest after every operation, we keep recently accessed nodes near the root and keep the tree roughly balanced, so that we achieve the desired amortized time bounds.

The run time for a splay(N) operation is proportional to the length of the search path for N: While searching for N we traverse the search path top-down. Let Z be the last node on that path. In a second step, we move Z along that path by applying rotations. There are six different rotations :

1. Zig Rotation (Right Rotation)
2. Zag Rotation (Left Rotation)
3. Zig-Zag (Zig followed by Zag)
4. Zag-Zig (Zag followed by Zig)
5. Zig-Zig
6. Zag-Zag

Consider the path going from the root down to the accessed node. Each time we move left going down this path, we say we "zig" and each time we move right, we say we "zag."

Zig Rotation and Zag Rotation: Note that a zig rotation is the same as a right rotation whereas the zag step is the left rotation. See Fig. 8.65.

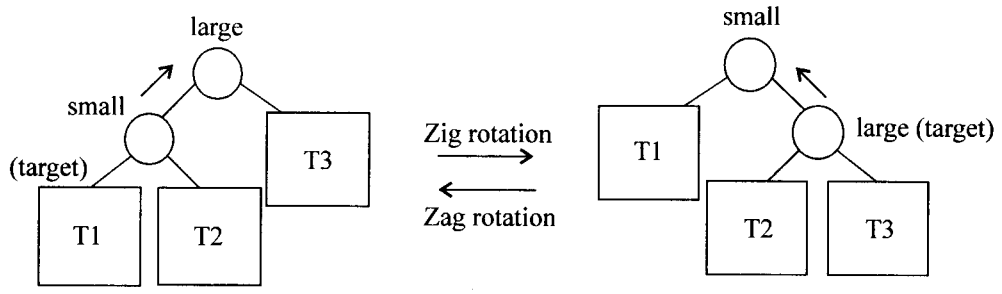


Fig. 8.65. Zig rotation and zag rotation

Zig-Zag: This is the same as a double rotation in an AVL tree. Note that the target element is lifted up by two levels. See Fig. 8.66

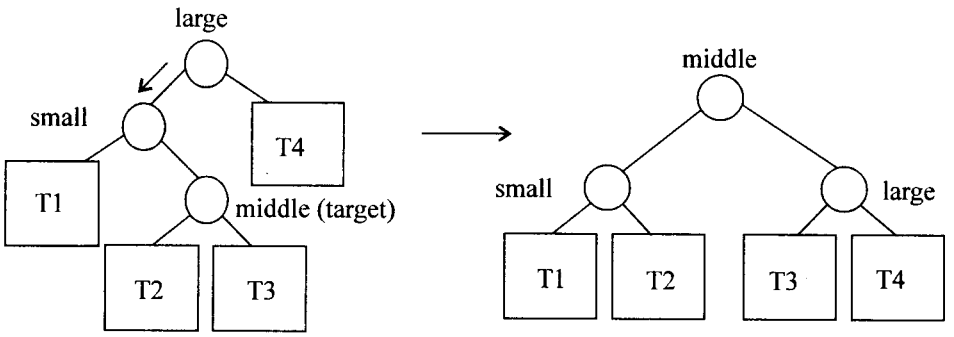


Fig. 8.66. Zig-zag rotation

Zag-Zig: This is also the same as a double rotation in an AVL tree. Here again, the target element is lifted up by two levels. See Fig. 8.67

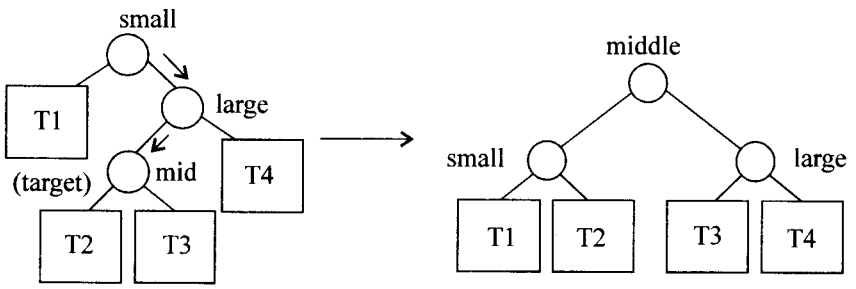


Fig. 8.67. Zag-zig rotation

Zig-Zig and Zag-Zag: The target element is lifted up by two levels in each case. Zig-Zig is different from two successive right rotations; zag-zag is different from two successive left rotations. For example, see Fig. 8.68 and Fig. 8.69.

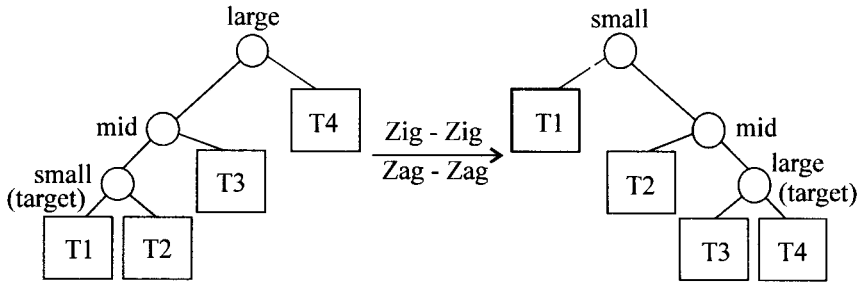


Fig. 8.68. Zig-zig and zag-zag rotations

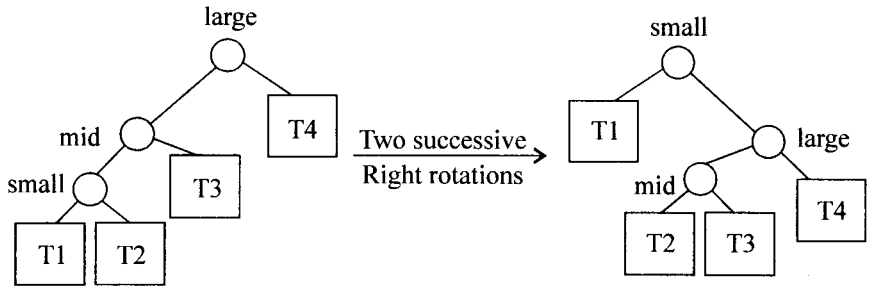


Fig. 8.69. Two successive right rotations

You may see an example in Fig. 8.70

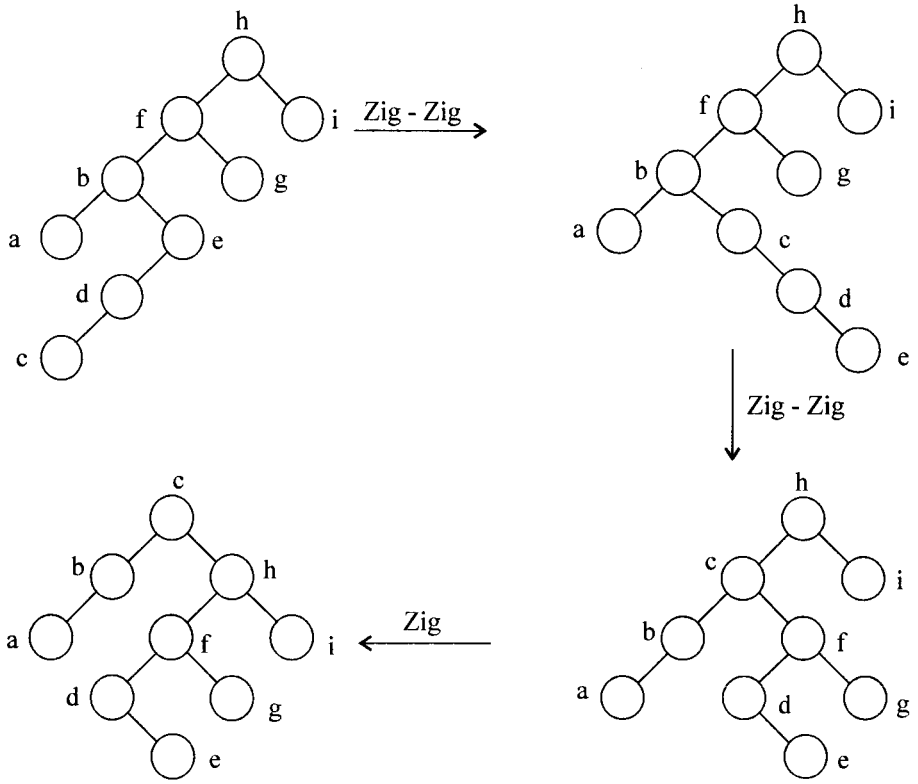


Fig. 8.70. An example of splaying

The above scheme of splaying is called *bottom-up splaying*. In *top-down splaying*, we start from the root and as we locate the target element and move down, we splay as we go. This is more efficient.

ADVANTAGES AND DISADVANTAGES

Good performance for a splay tree depends on the fact that it is self-balancing, and indeed self-optimising, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. This is an advantage for nearly all practical applications, and is particularly useful for implementing caches; however it is important to note that for uniform access, a splay tree's performance will be considerably (although not asymptotically) worse than a somewhat balanced simple binary search tree.

Splay trees also have the advantage of being considerably simpler to implement than other self-balancing binary search trees, such as red-black trees or AVL trees, while their average-case performance is just as efficient. Also, splay trees don't need to store any bookkeeping data, thus minimizing memory requirements. However, these other data structures provide worst-case time guarantees, and can be more efficient in practice for uniform access.

One worst-case issue with the basic splay tree algorithm is that of sequentially accessing all the elements of the tree in the sort order. This leaves the tree completely unbalanced (this takes n accesses- each an O(1) operation). Re-accessing the first item triggers an operation that takes O(n) operations to rebalance the tree before returning the first item. This is a significant delay for that final operation, although the amortized performance over the entire sequence is actually O(1). However, recent research shows that randomly rebalancing the tree can avoid this unbalancing effect and give similar performance to the other self-balancing algorithms.

It is possible to create a persistent version of splay trees which allows access to both the previous and new versions after an update. This requires amortized O(log n) space per update.

8.21. DIGITAL SEARCH TREES

A digital search tree is a binary tree in which each node contains one element. The element-to-node assignment is determined by the binary representation of the element keys. Suppose that we number the bits in the binary representation of a key from left to right beginning at one. Then bit one of 1000 is 1, and bits two, three, and four are 0. All keys in the left subtrees of a node at level I have bit I equal to zero whereas those in the right subtrees of nodes at this level have bit I = 1. Fig. 8.71 shows a digital search tree. This tree contains the keys 1000, 0010, 1001, 0001, 1100, 0000.

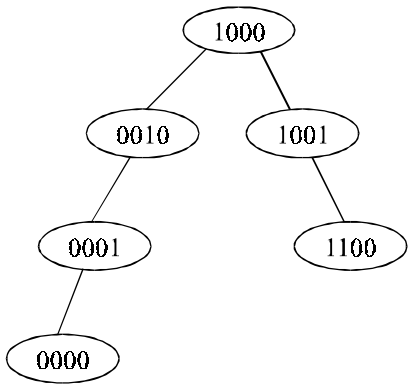


Fig. 8.71

Suppose we are searching for a key k = 0011 in the tree (Fig. 8.71). k is first compared with the key in the root. Since k is different from the key in the root, and since bit one of k is 0, we move to the left child (*i.e.*, 0010) of the root. Now, since k is different from the key in node and bit two of k is 0, we move to the left chills (*i.e.*, 0001). Since k is different from the key in node and bit three of k is one, we move to the right child of node 0001, which is NULL. From this we conclude that k = 0011 is not in the search tree. If we wish to insert k into the tree, then it is added as the right child of node 0001 as shown in the Fig. 8.72.

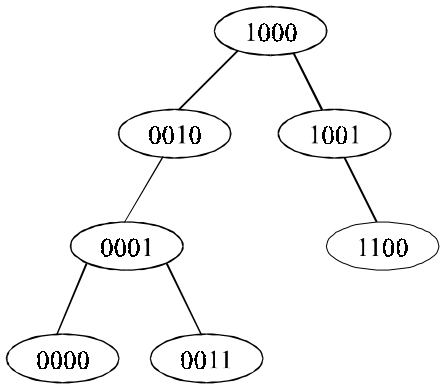


Fig. 8.72

The digital search tree functions to search and insert are quite similar to the corresponding functions for binary search trees. The essential difference is that the subtree to move to is determined by a bit in the search key rather than by the result of the comparison of the search key in the current node. The deletion of an item in a leaf is done by removing the leaf node. To delete from any other node, the deleted item must be replaced by a value from any leaf in its subtree and that leaf removed.

Each of these operations can be performed in O(h) time, where h is the height of the digital search tree. If each key in a search tree has SIZE bits, then the height of the digital search tree is at most SIZE + 1.

8.21. TRIES

The trie is a data structure that can be used to do a fast search in a large text. And a *trie* (from retrieval), is a multi-way tree structure useful for storing strings over an alphabet and it was introduced in the 1960's by Fredkin. It has been used to store large dictionaries of English (say) words in spelling-checking programs and in natural-language "understanding" programs.

A trie is an ordered tree data structure that is used to store an associative array where the datas (or key or information) are strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest.

That is no node in the trie contains full characters (or information) of the word (which is a key). But each node will contain associate characters of the word, which may ultimately lead to the full word at the end of a path. For example consider the data: an, ant, cow, the corresponding trie would look like be in Fig. 8.73, which is a non-compact trie.

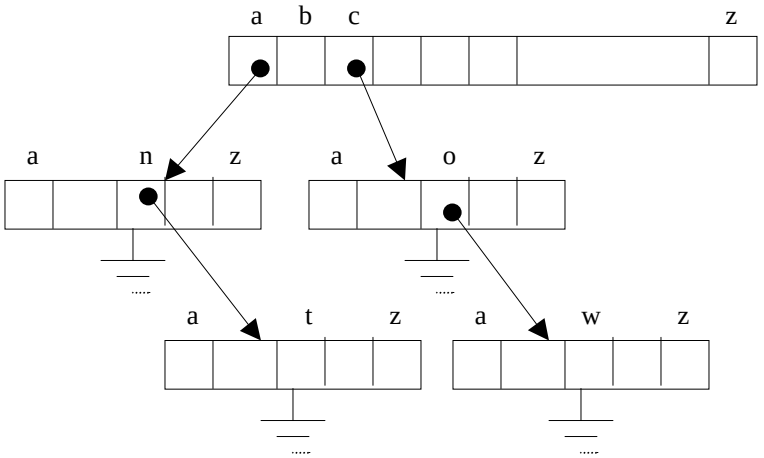


Fig. 8.73

The idea is that all strings sharing a common stem (or character or *prefix*) hang off a common node. When the strings are words over {*a*, *z*}, a node has at most 27 children — one for each letter plus a terminator.

The elements in a string can be recovered in a scan from the root to the leaf that ends a string. All strings in the trie can be recovered by a depth-first scan of the tree. The height of a trie is the length of the longest key in the trie.

ADVANTAGES AND DISADVANTAGES

There are three main advantages of tries over binary search trees (BSTs):

Searching a data is faster in tries. Searching a key (or data) of length *m* takes worst case O(*m*) = O(1) time; where BST takes O(log *n*) time, because initial characters are exam-

ined repeatedly during multiple comparisons. Also, the simple operations tries use during search, such as array indexing using a character, are fast on real machines.

Tries require less space. Because the keys are not stored explicitly, only an amortized constant amount of space is needed to store each key.

Tries make efficient longest-prefix matching, where we wish to find the key sharing the longest possible prefix with a given key. They also allow one to associate a value with an entire group of keys that have a common prefix.

Although it seems restrictive to say a trie's key type must be a string, many common data types can be seen as strings; for example, an integer can be seen as a string of bits. Integers with common bit prefixes occur as map keys in many applications such as routing tables and address translation tables.

Tries are most useful when the keys are of varying lengths and we expect some key search to fail, because the key is not present. If we have fixed-length keys, and expect all search to succeed, then we can improve key search by combining every node with a single child (such as "*i*" and "in" above) with its child, producing a patricia trie. That is when we are dealing with very long keys, the cost of a key comparison is high. We can reduce the number of key compariosns to one by using a related structure called patricia (Practical Algorithm To Retrieve Information Coded In Alphanumeric). We shall develop this structure in three steps. First, we introduce a structure called binary tries. Then we transform binary tries into compressed binary tries. Finally from compressed binary tries we obtain patricia. Binary tries and compressed binary tries are introduced only as a means of arriving at patricia.

SELF REVIEW QUESTIONS

1. Define and explain trees and binary trees.
   [*KERALA - DEC 2004* (*BTech*), *MG - MAY 2004* (*BTech*)]
2. What is binary search tree? Write an algorithm to insert and delete an item from a binary search tree. [*CUSAT - NOV 2002* (*BTech*), *MG - MAY 2004* (*BTech*) *ANNA - MAY 2004* (*BE*)]
3. What are different methods of binary tree traversal with examples?
   [*MG - NOV 2004* (*BTech*), *MG - NOV 2003* (*BTech*) *KERALA - DEC 2003* (*BTech*)]
4. Write an algorithm for the in-order traversal of a binary tree. [*MG - NOV 2004* (*BTech*)]
5. Explain the structure of a threaded tree. What are the conventions of representing threads?
   [*MG - MAY 2003* (*BTech*), *MG - MAY 2000* (*BTech*)]
6. Explain a height balanced binary tree.
   [*CUSAT - MAY 2000* (*BTech*), *MG - MAY 2003* (*BTech*) *ANNA - MAY 2004* (*MCA*)]
7. Discuss the improvement in performance of binary trees brought by using threads.
   [*MG - NOV 2003* (*BTech*)]
8. Discuss the difference between a general tree and a binary tree. What is a complete binary tree? Give an algorithm for deleting an information value X from a given lexically ordered binary tree. [*MG - NOV 2003* (*BTech*)]

9. What is a threaded binary tree? Explain in-order threading. [*MG - NOV 2002* (*BTech*)]
10. Given an account of the different classification of trees. [*MG - NOV 2002* (*BTech*)]
11. What are common operations performed on binary trees? [*MG - MAY 2002* (*BTech*)]
12. Write notes on height balanced and weight balanced trees. [*MG - MAY 2002* (*BTech*)]
14. Discuss the linked storage representation for binary trees. [*MG - MAY 2002* (*BTech*)]
15. What is a height of a binary tree? [*MG - MAY 2000* (*BTech*)]
16. Draw a binary tree for the expression A*B–(C+D)*(P / Q). [*MG - MAY 2000* (*BTech*)]
17. Discuss the internal memory representation of a binary tree using sequential and linked representation? [*MG - MAY 2000* (*BTech*)]
18. How to insert a node to the right of a given node in a threaded binary tree? [*Calicut - APR 1995* (*BTech*)]
19. Give a non-recursive algorithm for post order traversal of a binary tree. [*Calicut - APR 1995* (*BTech*)]
20. What are the applications of tree data structure? [*Calicut - APR 1997* (*BTech*), *Calicut - APR 1995* (*BTech*)]
21. Explain preorder, Inorder and Post order traversals. [*CUSAT - MAY 2000* (*BTech*)]
22. Explain binary tree traversals. Write an iterative function for inorder traversal. Explain the advantages of threaded binary tree over ordinary binary tree. [*CUSAT - JUL 2002* (*MCA*), *CUSAT - MAY 2000* (*BTech*) *ANNA - MAY 2004* (*MCA*)]
23. Explain the techniques for balancing heights of a binary tree. [*CUSAT - NOV 2002* (*BTech*)]
24. What is a threaded binary tree? Write algorithm for adding an element to a threaded binary tree. [*CUSAT - DEC 2003* (*MCA*)]
25. Define Binary search tree and its operations. [*ANNA - DEC 2003* (*BE*)]
26. Define various tree traversal methods. Write non-recursive algorithm for in-order tree traversal. [*ANNA - DEC 2004* (*BE*), *ANNA - DEC 2003* (*BE*) *ANNA - MAY 2003* (*BE*)]
27. What is Weight balanced tree? [*ANNA - MAY 2004* (*MCA*)]
28. Write an algorithm to convert a general tree to binary tree. [*ANNA - MAY 2004* (*BE*), *ANNA - MAY 2003* (*BE*)]
29. Define single rotation on AVL tree. [*ANNA - MAY 2003* (*BE*)]
30. What is the difference between binary tree and binary search tree? [*ANNA - MAY 2003* (*BE*)]
31. Write iterative procedure for preorder tree traversal. [*KERALA - DEC 2004* (*BTech*)]
32. Write an algorithm to traverse a binary tree in port order. [*KERALA - MAY 2003* (*BTech*)]
33. What are the uses of tree traversals? [*KERALA - DEC 2002* (*BTech*)]
34. Give the binary tree representation. [*KERALA - MAY 2001* (*BTech*)]
35. Write an iterative procedure for post-order tree traversal. [*KERALA - MAY 2001* (*BTech*)]
36. Draw a binary tree for the following expression A * B - (C - D) * (P / Q). [*KERALA - MAY 2002* (*BTech*)]
37. Write an algorithm to count the leaf nodes in a binary tree. [*KERALA - MAY 2002* (*BTech*)]