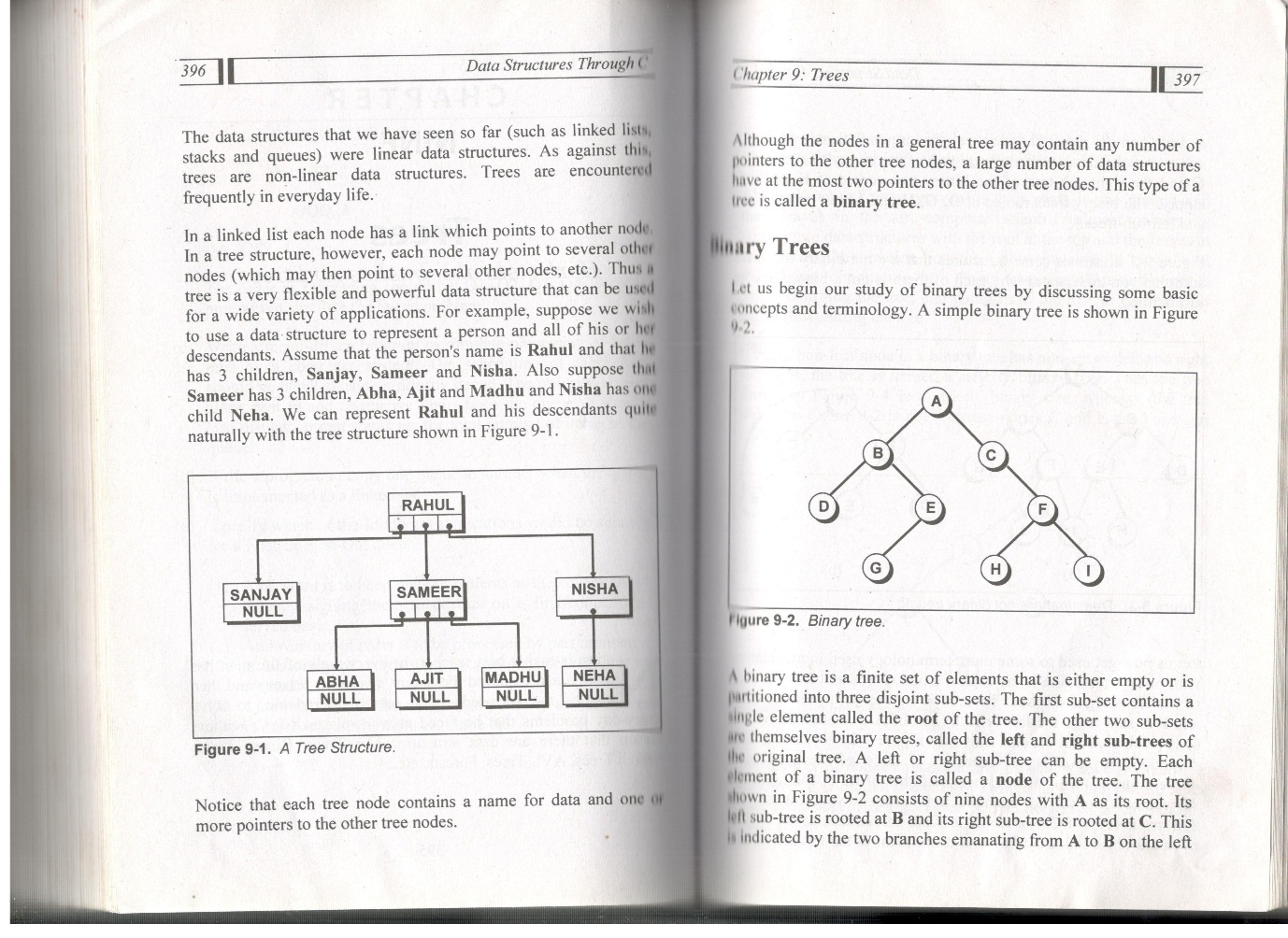
Trees-Herbs, Shrubs, Bushes and trees

Nature is man’s best teacher. In every walk of life ma has looked and explored the nature, learnt his lessons and then every-day problems that he faced at work-place. It isn’t without reason that there are data structures like Trees, Binary Trees, Search Trees, AVL Trees, Forests, etc.

The data structure that we have seen so far (such as linked lists, stacks and queues) were liner data structures. As against this, trees are non-liner data structures. Trees are encountered frequently in everyday life.

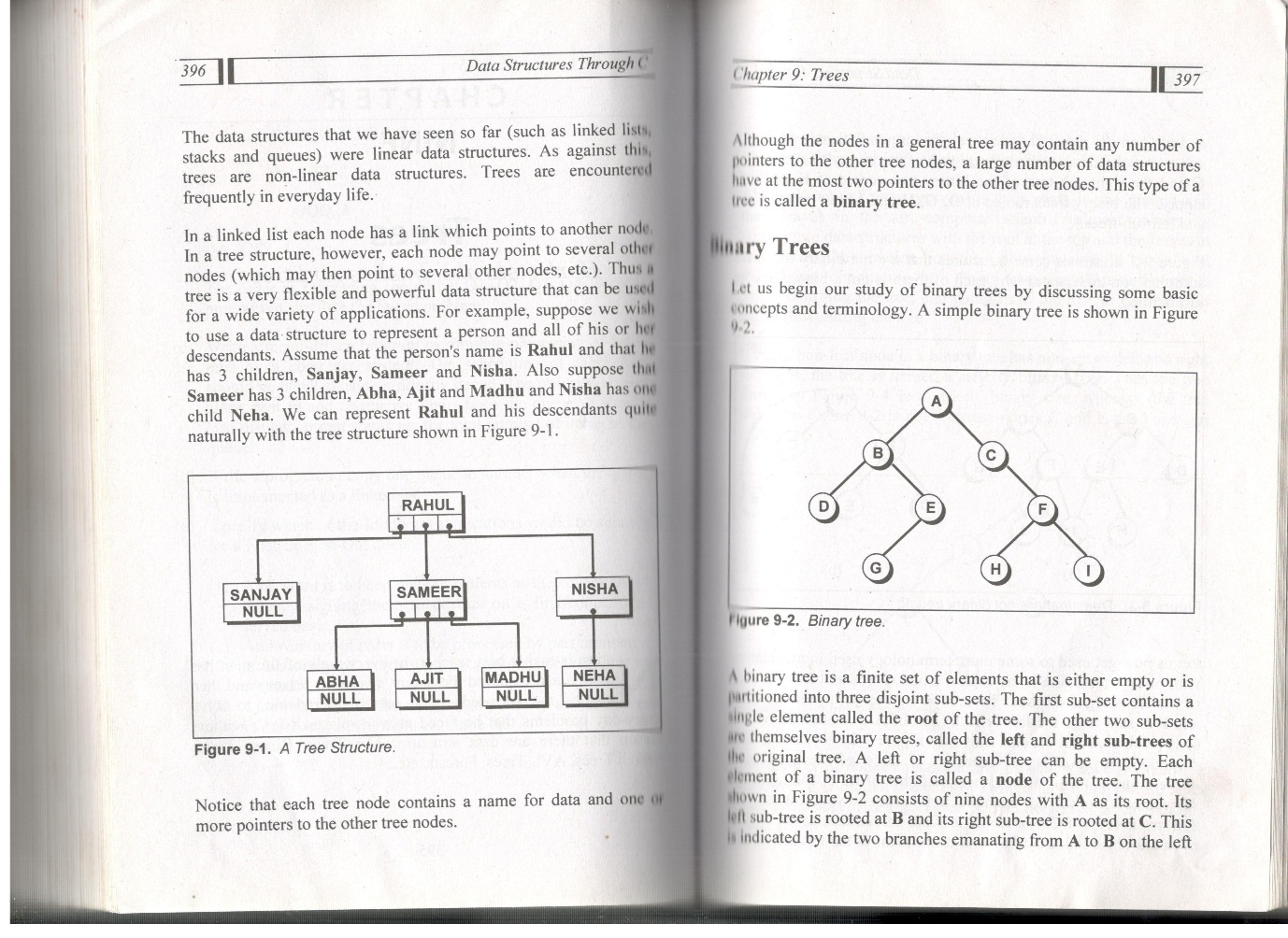
In a linked list each node has a link whlch point another node. In a tree structure, however, each node may point to several other nodes (whlch may then point to several other nodes, etc.)Thus a tree is a very flexible and powerful data structure that can be used for a wide variety of appllcations. For example, suppose we wish to use a data structure to represent a person and all of his or her descendants. Assume that the person’s name is Rahul and that he has 3 children,Sanjay,Sameer and Nisha.Also suppose that Sameer has 3 children,Abha,Ajit and Madhu and Nisha has one child of Neha .We can represent Rahul and his descendants quite naturally with the tree structure shown in 9-1. (9-1 A Tree Structure )

Notlce that each tree node contain a name for data and one or more pointers to the other tree nodes.

Although the nodes is a general tree may contain any number of pointers to the 0ther tree nodes, a large number of data structures have at the most two pointer to the other tree nodes, This type of tree is called a binary tree.

Binary Trees

Let us begin our study of binary trees by discussing some basic concepts and terminology. A simple binary tree is shown in figure 9-2.

(9-2Binery Tree)

A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint sub-sets. The first sub-set contains a single elements called the root of the tree the other two sub-sets are themselves binary trees, called the left and right sub-trees of the original tree. A left or right sub-tree can be empty. Each element of a binary tree is called a node of the tree the tree shown I figure 9-2 consists of nine nodes with A as its root. Its left sub-tree is rooted at B and its right sub-tree is rooted at C. This is indlcated by the two branched emanating from A to B on the left and to C on the right .The absence of a branch indicates an empty sub-tree. For example, the left sub-tree of the binary tree rooted at C and the right sub-tree of the binary rooted at E are both empty.This binary trees rooted at D, G, H and I have empty right and left sub-trees.

Figure 9-3 illustrates some structures that are not trees. Be sure that you understand why each of them is not a binary tree as just defined.

(Figure 9-3 Trees that are not binary trees.)

Let us now get used to some more terminology used in association with binary trees. If A is the root of a binary tree and B is the root of its left or right sub-tree. Then, A is said to be the father of B and B is said to be the left or right son of .A node that has no sons (such as D, G, H, or I shown in figure 9-2) is called a **leaf.** Any node say n1,is an ancestor of node n2 or the father of some ancestor

Of n2.For example ,in the tree shown in figure 9-2, A is an ancestor of C . A node n2 is a leaf descendant of node n1 if n2 is either the left son of n1 or a descendant of the left son of n1.

A right descendant may be similarly defined.Two nodes are brothers if thay are left and right sons of the same father.

Althogh natural trees grow with their roots in the ground and their leaves in the air ,computer scientist almost universally portray tree data structure with the root at the top and leave at the bottom.The direction from the root to the leave is “down”and the opposite direction is “up”.Going from the leave to the root is called climbing the tree,and going from the root to the leave is called descending the tree.

(Figure 9-4 Strlctly binary tree.)

The number of nodes connected to a partlcular node is called the degree of a partlcular node. For example, in figure 9-4 the node containing data D has a degree 3.The degree of a leaf node is always one.

The level of a node in a binary tree is defind as follows:

The root of the tree has level 0,and the level of any other node in the tree is one more than the level of its father.For example ,in the binary tree shown in figure9-2,node E is at level 2and node H is at level 3.The depth of a binary tree is the maximum level of any leaf in the tree.This equal the length of the longest path from the root to any leaf.Thus the depth of the tree shown in figure 9-2 is 3.A complete binary tree of depth of d is a strlckly binary tree all of whose leaves are at the same level d.Figure 9-5 illustrates the complete binary tree of depth 2.

Traversal of a Binary Tree

The traversal of a binary tree is to visit each node in the tree exactly once. Binary tree traversal is useful in many applications. The order in which nodes of a liner list are visited is clearly from first to last .However, there is no such natural liner order for the nodes of a tree. The methods differ primarily in the order in which they visit the nodes. There are three popular methods of binary tree traversal. These methods are known as **in –order traversal** **,Pre-order traversal** and **post-order traversal**. In each of these methods nothing need be done to traverse an empty binary tree. The function used to traverse a tree using these methods can be kept quite short if we understand the recursive nature of the binary tree. Recall that a binary tree is recursive In that each sub-tree is really a binary tree itself. Thus traversing a binary tree involves visiting the root node and traversing its left and right sub-trees. The only difference among the methods is the order in which these three operations are performed.

To traverse a non-empty binary tree in pre-order, we perform the following three operations:

1.Visit the root

2.Traverse the left sub-tree in pre-order

3.Treverse the right sub-tree in pre-order

To traverse a non-empty binary tree in in-order (or symmetric order):

1.Traverse the left sub-tree in in-order

2.Visit the root

3.Traverse the right sub-tree in-order

To traverse a non-empty binary tree in post-order:

1.Traverse the left sub-tree in post –order

2.Traverse the right sub-tree in post –order

3.Visit th eroot

Representation Of A Binary Trees In Memory

The structure of each node of a binary tree contains the data field and since a binary tree is recursive in nature it contains the pointer to the same structure .Instated of one pointer it contains two pointers, one for the left child and another for the right child. Figure 9-6 shows the structure of a node of a binary tree.

The structure that defines a node of a binary tree is as follows:

Struct tnode

{

Struct toned \*left;

Int data;

Struct tnode\*right;

};

There are two ways by whlch we can represent a binary tree.

1. Linked representation of a binary tree
2. Array representation of binary tree

Both these ways are discused below.

Linklist Representation Of Binary Trees

Binary trees can be represented by links, where each node contains the address of the left child and the right child. The lesf node contains a NULL value in its links fields, as there is no child(left or right)to a leaf node. Some node contains a NULL value in its left or right link field because the respective child of that particular node could be empty .Figure 9-7 shows the linked representation of a binary tree.

I figure 9-7,the link fields of node C contains the address of the nodes F and G .The left link field of node E contains the address of thee node H. Similarly the right link contains a NULL value as there is only one (left)child or node E.

The node D,F,G and H contains a NULL value in both their link fields, as these are the leaf nodes.

Array Representation Of Binary Trees

When a binary tree is represented by arrays three separate arrays are required .One array arr stored the data fields of the trees. The other two arrays Lc and rc repents the left child and right child of the nodes.Figure9-8 shows these three arrays, which represents the tree that is shown I figure9-7.

The array lc and rc contains the index of the array arr where the data is present. If the node does not have any left child or right child then the element of the array lc or rc contains a value -1.The 0th element of the array arr that contains the data is always the root node of the tree. Some elements of the array arr contain’\0’ which repents an empty child.

Let us understand this with the help of an example .Suppose we want to find the left and right child of the node E. Then we

Need to find the value present at index 4 in array lc and rc since E is present at index 4 in the array arr. The value present at index 4 in the array lc is 9, which is the index position of node H in the array arr.So the left child of the node E is H.The right child of the node E is empty because the value present at index 4 in the array rc is-1.

Following program shows how a binary tree can be reprented using arrays.

#include <stdio.h>

#include<conio.h>

#include<mulloc.h>

{

Struct node\*left;

Char data ;

Struct node\*right;

}

Struct node\*buildtree(int);

Void inorder(struct node\*);

Char arr[]={‘A’,’B’,’C’,’D’,’E’,’F’,’G’,’\0’, ’\0’,’H’,};

Int lc[]={1,3,5,-1,9,-1, -1, -1 ,-1,-1,};

Int rc []={2,4,6,-1 ,-1, -1, -1 ,-1,-1,-1};

Int main()

{

Struct node \*root;

System (“cls”);

root=buildtree(0);

printf(“on-order Traversal:\n”);

inorder(root);

return0;

}

Struct node\*buildtree (int index)

{

Struct node\*buildtree(int index)

{

Struct node\*temp=NULL;

If (index!=-1)

{

Temp=(struct node\*)malloc(sizeof(struct node));

Temp->left=buildtree(lc[index]);

Temp ->data=arr[index]);

Temp->data+arr[index];

Temp->right+buildtree(rc[index]);

}

Return temp;

}

Void inorder (struct node\*root)

{

If (root !=NULL)

{

Inorder (root ->left);

Printf(“%c\t”,root->data);

Inorder(root ->right);

}

}

Output :

In-order Traversal:

D B H E A F C G

The function build tree () is called by passing a value 0 and is received in an integer variable index which indicates the 0th element of the array arr(the root node).In the function a conditional is checked ,whether index is -1.If it is, then it indicates that the particular node doesn’t have any child. But if the index is not-1 then memory is allocated for a new node. The data from the array arr is stored in data part of the new node. Family, a recursive call is made for the left and right child of the new node, by passing the corresponding index values from the arrays lc and rc.

The function inorder() is called to traverse the tree in in-order traversal .This function receives only one parameter as the address of the root node.Then a condition is checked whether the pointer is NULL. If the pointer is not NULL then a recursive call is made first for the left child and then for the right child. The values passed are the addresses of the left and right children that are present in the pointer left and right respectively. In-between these two calls the data of the current node is printed.

We can also represent a binary tree using one single array. For this, numbers are given to each node starting from the root node-0 to root node,1 to left node of the first level, then 2 to the second node from left of the first level and so on. In other words, the nodes are numbered from left to right level by level from top to bottom. Figure 9-9 (a)shows the numbers given to each node in the tree. Note that while numbering the nodes of the tree, empty nodes are also taken into account.

Figure 9-9(b) shows the array representation of the tree that is shown in figure 9-9(a)

It can be observed that of n is the number given to the node then its left child occurs at the position (2n+1)and right child at (2n+2).

If any node doesn’t have a left or right child then an empty node is assumed and a value ‘\0’ is stored at that index in the array.

Following program shows how a binary tree can be represented as a array using the m etjod discussed above.

#include<stdio.h>

#include<conio.h>

#include<malloc.h>

#include<windows.h>

Struct node

{

Struct node\*left;

Char data;

Struct node\*right;

};

Struct node\* builtree(int);

Void inorder(struct node\*);

Chara[]={

‘A’,’B’,’C’,’D’,’E’,’F’,’G’,’\0’, ’\0’,’H’, ’\0’,

’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’,

};

Int main()

{

Struct node \*root;

System(“cls”);

Root+buildtree(0);

Printf(“in-order Traversal :\n”);

Inorder(root);

Return 0;

}

Struct node\*buildtree(int n)

{

Struct node\*temp=NULL;

If (a[n]!=’\0’)

{

Temp+(struct node\*)malloc(sizeof (struct node));

Temp ->left=buildtree (2\*n+1);

Temp->data=a[n];

Temp-.right=buildtree(2\*n+2);

}

Return temp;

}

Void inorder (struct node\*root)

{

If (root !=NULL)

{

Inorder(root ->left);

Printf(“%c\t”,root ->data);

Inorder(root ->right);

}

}

Output:

In-order Traversal:

D B H E A F C G

In the above program the working of functions inorder() and buildtree ()is same as what we saw in the last program except for a small change in the function buildtree (). Here, the condition is checked that the value present at particular index is’\0’.If it is not ‘\0’then the recursive calls are made for the left and right child.

Instead of three arrays only one array is maintained and hence the value of index passed to recursive call is 2\*n+1 and 2 \* n+2 for the left and right child respectively.

Binary Search Trees

Many algorithms that use binary trees proceed in two phases. The first phase builds a binary tree, and the second traverses the tree. As an example of such an algorithm, consider the following sorting method. Given a list of numbers in an input file, we wish to print them in ascending order. As we read the numbers, they can be inserted into a binary tree such as the one shown I figure 9-10.

When a number is compared with the contents of a node in the tree, a left branch is taken if the number is smaller than the contents of the node. Thus if the input list is

20 17 6 8 10 7 18 13 12 5

Then the binary tree shown in figure 9-10 is produced.

Such a binery tree has the property that all the elements in the left sub-tree of a node n are less than the contents of n. And all the elements in the right sub-tree of n are greater than or equal to the contents of n.

A binary tree that has these properties is called a binary search tree. If a binary search tree is traversed in in –order (left,root,,and right)and the contents of each node are printed as the node is visited ,the numbers are printed in ascending order. Convince yourself that this is the case for the binary search tree shown in figure 9-10.The program to implement this algorithm is as follows:

#include<stdio.h>

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

#include<windows.h>

Struct btreenode

{

Struct btreenode\*leftchild;

Int data;

Struct btreenode\*rightchild;

};

Void insert (struct btreenode\*\*,int);

Void inorder(struct btreenode\*);

Void preorder(struct btreenode\*);

Void postorder(struct btreenode\*);

Int main()

{

Struct btreenode\*bt;

Int req,i=1,num;

Bt=NULL;/\*empty tree\*/

System (“cls”);

Printf(“specify the number of items to be inserted:”);

Scanf(“%d”&req);

While (i++<+req)

{

Printf(“Enter the data:”);

Scanf(“%d”,&num);

Insert (&bt,num);

}

Printf(“\n”);

Printf(“in-order Traversal:\n”);

Inorder(bt);

Printf(“\n”);

Printf(“pre-order Traversal:\n”);

Preorder(bt);

Printf(“\n”);

Printf(“post-orderTraversal:\n”);

Postorder(bt);

Return0;

}

/\*inserts a new node in a binery seach tree \*/

Void insert(struct btreenoode\*\*sr,intnum)

{

If (\*sr==NULL)

{

\*sr=(struct btreenode\*)malloc(sizeof(struct btreenode));

(\*sr)-.leftchild=NULL;

(\*sr)->data=num;

(\*sr)->rightchild=NULL;

Return;

}

Else/\*search the node to which new node will be attached \*/

{

/\*if new data is less ,traverse to left\*/

If (num<(\*sr)->data)

Insert (&((\*sr)->rightchild),num);

}

Return;

}

/\*traverse a binery search tree in a LDR (left –data-right)fashion\*/

Void inorder(struct btreenode\*sr)

{

If (sr!=NULL)

{

Inorder (sr->leftchild);

/\*print the data of the node whose leftchild is NULL or the path

Has already been traversed\*/

Printf(“%d\t”,sr->data);

Inorder(sr->rightchild);

}

Else

Return;

}

/\*traverse a binery search tree in a DLR (data-left-right)fashion\*/

Void peroreder(struct btreenode\*sr)

{

If (sr!=NULL)

{

/\*print the dat of a node \*/

Printf (“%\t”,sr->data);

/\*traverse till leftchild is not NULL\*/

Preorder(sr->leftchild);

/traverse till rightchild is not NULL \*/

Preorder(sr->rightchild);

}

Else

Return;

}

/\*traverse a binary search tree in LRD(ledt-right-data)fashion\*/

Void postorder(struct btreenode\*sr)

{

If (sr!+NULL)

{

Postorder(sr->leftchild);

Postorder(sr->rightchild);

Printf(“%d/t”,sr->data);

}

Else

Return;

}

Output:

Specify the number of items to be inserted:5

Enter the data :1

Enter the data :2

Enter the data :3

Enter the data :4

Enter the data :5

In-order Traversal:

1 2 3 4 5

Pre-order Traversal:

1 2 3 4 5

Post-order Traversal:

5 4 3 2 1

In the above program the working of the function inorder() is exactly the same as the last program. The functiom preorder()

And postorder ()work in the same manner except for a small difference. In case of the function preorder() initially data is printed then the recursive calls are made for the left and right children. On the other hand, in case of postorder() firstly the recursive calls a for left and right children are made and then the data is printed.

In the function insert() two arguments are passed-one is the pointer to the node. of the tree and the other is the data that is to be inserted. Initially, the pointer to the node contains a NULL value, which indicates an empty tree. Then a condition is checked whether the pointer is NULL. If it is NULL then a new node is created and the data that is to be inserted is stored in its data part.

The left and right child of this new node is set with a NULL value,

as this is always going to be the leaf node. In the else block, i.e. if the current node is not empty then the dataof the current node is compared with the data that is to be inserted If the data that is to be inserted is found to be smaller than the dataof the current node then a recursive call is made to insert( ) function by passing the address of the node of the sub-tree, otherwise the address of the node of right sub-tree is passed. So at one stage in the recursive call the node is found to empty, which is the place where the new node is to be inserted.

Operations On A Binary Search Tree

There are many operations that can be performed on binary searchtrees. Searching, insertion and deletion of a node are thebasic operations that are required to maintain a tree. Let us nowdiscuss these operations in detail.

Searching Of A Node In A BST

To search any node in a binary tree, initially the data that is to be searched is compared with the data of the root node. If the data is equal to the data of the root node then the searching is successful. If the data is found to be greater than the data of the root node the if the searching process proceeds in the right sub-tree of there node, otherwise, searching process proceeds in the left sub-tree the root node. Same procedure is repeated for the left or right sub-tree until data is found. While searching the data if the leaf node of tree

reached and the data is not found then it is concluded that the is not present in the tree.

Consider Figure 9-10 and suppose the data that is to be searched 8. Initially 8 is compared with root node which holds a value 20and since 8 is less than 20 the searching would proceed in the leftsub-tree. Now 8 is compared 17 and since 8 is less than 17 these searching would proceed in the left sub-tree of the node which holds a value 17. Next, 8 is compared with 6 and since 8 is greater than 6 the searching proceeds in the right sub-tree of the node which holds a value 6. Now 8 is compared with the node which holds a value 8 and since 8 is found the searching process ends here.

Insertion Of A Node In A BST

To insert any' node ,into a BST, initially the data that is to be inserted is compared with the data of the root node. If the data is found to be greater than or equal to the data of the root node then the new node is inserted in the right sub-tree of the root node, otherwise, the new node is inserted in the left sub-tree of the root node. Now the root node of the right or left sub-tree is taken and its data is compared with the data that is to be inserted and the same procedure is repeated. This is done till the left or the right sub-tree where the new node is to be inserted is found to be empty. Finally, the new node is made the appropriate child of this current node.

Figure 9-12 shows how a new node is inserted into a binary search

tree.

Suppose the new node that is to be inserted holds a value 7 in its data ﬁeld. To ﬁnd the appropriate position of this "new node in the tree, initially it is compared with the root node, which holds a value 20. Since 7 is less than 20, the searching of appropriate position of new node will be done in the left sub-tree. Now 7 is compared with 17 and since 7 is less than 17 the searching of appropriate position of- new node will be done in the. left sub-tree of the node that holds a value 17. Next 7 is compared with 6 and since 7 is greater than 6 the searching of appropriate position of new node is done in the right sub-tree of the node that holds a value 6. Now 7 is compared with 8 and since 7 is less than 8 the searching of appropriate position of the new node is done in the left sub-tree of the node that holds a value 8. But since this left sub-tree is empty the new node is made the left child of the node that holds a value 8.

Deletion From A Binary Tree

In addition to techniques for inserting data in a binary tree and

traversing the tree, practical examples call for deleting data from

the binary tree. Assuming that we will pass the speciﬁed data item

that we wish to delete to a del( ) function, there are four possible

cases that we need to consider:

(a) No node in the tree contains the speciﬁed data.

(b) The node containing the data has no children.

(c) The node containing the data has exactly one child.

(d) The node containing the data has two children.

Case (a):

Here, we merely need to print the message that the data item is not

Present in the tree

Case (b):

In this case since the node to be deleted has no children the memory occupied by this should be freed and either the left link 0r the right link of the parent of this node should be set to NULL.Which of these to set to NULL depends upon whether the NULL.being deleted is a left child or a right child of its parent.

Case (c):

In this case since the node to be deleted has one child the solution

is again rather simple. We have to adjust the pointer of the parent

of the node to be deleted such that after deletion it points to the

child of the node being deleted. This is shown in Figure 9-13.

Case (D)

In this case since the node to be deleted has two children solution is more complex. Consider node 8 shown in Figure 9-14 .Before Deletion. The in—order successor of the node 8 is node 9.The data of this in-order successor should now be copied into node to be deleted and a pointer should be set up pointing to ,in-order successor (node 9). This in-order successor would always 9have one or zero child. This in-order successor should then deleted using the same procedure as for deleting a one child or zero child node. Thus the whole logic of deleting a node with children is to locate the in-order successor, copy its data reduce the problem to a simple deletion of a node with one or zero child. This is shown in Figure 9-14.

A program to implement the operations performed on a binary search tree is given below

#include<stdio.h>

#include<conio.h>

#include<malloc.h>

#include<windows.h>

#define TRUE1

#defineFALSE0

Struct btreenode

{

Struct btreenode\*leftchild;

Int data;

Struct btreenode\*rightchild;

};

Void insert(struct btreenode\*\*,int);

Void del(struct btreenode\*\*,int);

Void search(struct btreenode\*\*,int,struct btreenode\*\*,

Struct btreenode\*\*,int\*);

Void inorder (struct btreenode\*);

Int main()

{

Struct btreenode\*bt;

Int I =0,[]={11,9,13,8,10,12,14,15,7};

Bt=NULL;/\*empty tree\*/

System(“cls”);

While (i<=8)

{

Insert (&bt,a[i]);

I++;

}

System (“cls”);

Printf(“Binary tree before deletion:\n”);

Inorder(bt);

Del (&bt,10);

Printf(“\n);

Printf(“Binary tree after deletion:\n”);

Inorder(bt);

Del (&bt,14);

Printf(“\n”);

Printf(“Binary tree after deletion:\n”);

Inorder(bt);

Return0;

}

/\*inserts a new node in a binary search tree\*/

Void insert (struct btreenode\*\*sr,int num)

{

If (\*sr ==NULL)

{

\*sr =(struct btreenode\*)malloc (sizeof(struct btreenode));

(\*sr)->leftchild=NULL;

(\*sr)->data=num;

(\*sr)->rightchild =NULL;

}

Else /\*search the node to which new node will be attached\*/

{

?\*if new data is less,traverse to left \*/

If (num<(\*sr)->data)

Insert (&((\*sr)->leftchild),num);

Else

/\*else traverse to right \*/

Insert (&((\*sr)->rightchild),num);

}

}

/\*deletes a node from the binary search tree\*/

Void del (struct btreenode\*\*root,int num)

{

Int found;

Struct btreenode \*parent, \*x,\*xsucc;

/\*if tree is empty \*/

If (\*root ==NULL)

{

Printf(“Tree is empty.\n”);

}

P=aren’t =x=NULL;

/\*call to search function to find the node to be deleted\*/

Search (root,num,&parents,&x,&found);

/\*of the node to deleted is not found\*/

If (found==FALSE)

{

Printf(“Data to be deleted,not found.\n”);

Return;

}

/\*if the node to be deleted has two children\*/

If (x->leftchild!=NULL&&x->rightchild!=NULL)

{

Parent=xsucc;

Xsucc= Xsucc->leftchild;

}

/\*if the node to be deleted has no child\*/

If (x->leftchild==NULL&&x->rightchild==NULL)

{

If (parent->rightchild==x)

Parent ->rightchild=NULL;

Else

Parent->leftchild=NULL;

Free(x);

Return;

}

/\*if the node to be deleted has only rightchild\*/

If (x->leftchild==NULL&&x->rightchild!=NULL)

{

If (parent->left child==x)

Parent ->leftchild=x->rightchild;

Else

Parent ->rightchild=x->rightchild:

Free(x);

Return;

}

/\*if the node to be deleted has only left child\*/

If(x->leftchild !=NULL&7x->rightchild==NULL)

{

If (parent->leftchild==x)

Parent->leftchild=x->leftchild;

Free(x);

Return;

}

/\*return the address of the node to be deleted,address of its parent and whether the node is found or not \*/

Void search (struct btreenode\*\*root,int num,struct btreenode\*\*par,struct

Btreenode88x,int\*found)

{

Struct btreenode\*q;

Q=\*root;

\*found=FALSE;

\*par=NULL;

While(q!=NULL)

{

/\*if the node to be deleted isfound \*/

If(q->data==num)

{

\*found=TRUE;

\*x=q;

Return;

}

\*par =q;

If(q->data>num)

Q=q-.leftchild;

Else

Q=q->rightchild;

}

}

/\*traverse a binary search tree in a LDR(left-data-right)fashion\*/

Void inorder(struct btreenode\*sr)

{

If (sr!=NULL)

{

Inorder)sr->leftchild);

/\*print the data of the node whose leftchild is NULL or the path has already been traverse\*/

Printf(“%d\t”,sr->data);

Inorder(sr->rightchild);

}

}

Output:

Binary tree before deletion:

7 8 9 10 11 12 13 14 15

Binary tree after deletion:

7 8 9 11 12 13 14 15

Binary tree after deletion:

7 8 9 11 12 13 15

Binary tree after deletion:

7 9 11 12 15

In the order above program the working of the function insert()and inorder()is exactly the same as what we saw in the previous program. The function search() searches for the given number in the tree and returns the address of the node where the number is found, address of its parent and an integer value which holds either TRUE or FALSE (0 or 1) depending upon whether the number found or not.

The function search( ) receives ﬁve parameters. The ﬁrst parameter root is the address of the root node of the tree, second parameter num is the number that 15 to be searched, the third fourth parameters (par and x) are the pointers to the addresses the parent of the node where data 15 found and the address of node itself, respectively, and the last parameter found is integer value indicating whether the element IS found or not.

In the insert( ) function initially the variable pointed by found is set to a FALSE value and the value of par is set to NULL because if the node 15 not found then the pointer pointed by per should hold a NULL value. Then a while loop 15 executed with condition q!= NULL where q holds the address of the root noInside the while loop a condition is checked for the data to be searched. If the data is found then a TRUE value is stored in

variable pointed by found and the address of the current node stored 1n x. If the data 15 found 1n the ﬁrst iteration then the value of par is NULL, since it has no parent.

If the data 1s n0t found 1n the ﬁrst iteration then inside the while loop the address of the current node 1s stored 1n par and then data 13 compared with the data present in the current node .If data of the current node is greater than the data Which is tosearched then q will hold the address of the left sub- tree of the current node otherwise it will hold the address of the right sub-tree of the current node. This way any function that calls search( ) gets the address of the node where the node 1s present and the address of its parents.

In the function del( ) two parameters are received. The ﬁrst is the

address of the root node and other is the number that is to be

deleted. Initially, a condition is checked whether the root node is empty. If it is then a message will be displayed and the control will be returned back, otherwise the function search( ) is called with NULL values stored in the two pointer variables parent and x.

If the data that is to be deleted is not found then the last argument

passed to function search( ) holds a value FALSE. Hence, after

the call to function search( ) a condition is checked whether

found = FALSE. If it is equal to FALSE then it indicates that

the data is not present in the tree. So an appropriate message will

be displayed and control 'will be returned back.

If the data that 15 to be deleted 15 found then the following four

cases would arise.

(a) the node has two children

(b) the node has no child

(c) the node has only right child

((1) the node has only left child

Case (a): Node to be deleted has two children

In this case, initially the address of the node that is to be deleted is

stored in the pointer parent. But by doing so the address of the

parent node that is to be deleted is lost. We don’t mind doing so,

because we are not interested in storing any address or a NULL

value in the left or right child of the parent node. What we need to

do is to ﬁnd out the in-order successor of the node that is to be

deleted. For this the address of the right child of the node that is to

be deleted is stored in a pointer variable xsucc. Then a while loop

is executed where a condition is checked whether the left child of

xsucc is NULL. If it is not then the address of the xsucc is stored

in the parent and the left child of the xsucc is stored in the xsucc.

As a result, at the end of the while loop the value present in xsucc

is the address of the in-order successor of the node that is to be

deleted and will always either have only one child or no child.

After the while loop the value parent in the in-order succrssor is copied into the data that is to be deleted.Finally,the address of the in-order successor is stored in x,which is the node that is to be deleted.So the logic of deleting the node which has two children is converted to the case of deleting the node which has only one or no child,which are discussed below:

Case (b):Node to be deleted has no child

Since noth the children of node that is to be deleted hold a NULL.value what needs to be done is store a NULL value in the respective child of the parent.This is done by checking the condation parent-> rightchild==x.If the rightchild of the parent node is equal to the value in the node to be deleted,then a NULL value is stored in the rightchild of the parent,otherwise,it is stored in the leftchild.Finally the memory occupied by the noed to be deleted is released and control is returned.

Case(c) :Node to be deleted has only right child

Here a condition is checked whether the node that is to be deleted is the lefdt child of its parent,this is done yhrough the statement

If (parent->leftchild==x)

If this condation is true then the address of the rightchild of the node that is to be deleted is stored in the leftchild of the parent node,otherwise in the rightchild of the parent node.Then the memory occupied by the node to be deleted is released and control is returned.

Casr(d ):Node to be deleted has only left child

The action taken here is similar to case ( c) discussed above

Applications Of Binary Trees

There are many application of a boinary tree.For example whenever we want to taketh etwo –way decesions,binary trees are the best option.One such application is a b inary search tree,which we have already discussed.Binary trees can also be used to represent expressions as discussed below.

Representing Expressions In Binary trees

The arithmetic expression represented as binary tree known as expression trees.The root node is operator and the left and right children are operands.In case of unary operators the left child is absent and the right child is the operand.Since there is no operator available for exponent.Generally,nodes appearing in circular shape denote the operators and nodes appearing inb square shape are used for operands.For example,Figure 9-15 shows a tree for the expression

A\*B+C\*D+E

When the exprerssion tree is traversed in pre-order then the prefix from of the expression is obtained.Similarly ,when the expression tree is traversed in post-order then the postfix form of the expression is obtained.In the same way we get the infix form of the expression is obtained.In the same way we get the infix form of the expression when the tree is traversed in in-order .But the parentness parent in the original arithmetic expression cannot be obtained back from the in-order traversal of the binary tree.Thisis because the structure of the binary tree itself decides the order of the evaluation of the tree.

Extended Binary Tree

A binary tree can be converted to an extended binary tree by adding new nodes to its leaf nodes and to the nodes that have only one child.These new nodes are added in such a way that all the nodes in the resultant tree have either zero or two children.The extended tree is also known as a 2-tree.The nodes of the original tree are called internal nodes and the new nodes that are added to binary tree,to make it an extended binary tree are called external nodes.

Digure 9-16shows how extended binary tree(figure9-16( b))can be obtained by adding new notes to original tree(Figure9-06(a)).

I figure 9-16 ,all the nodes with circular shape are internal nodes and all the nodes with square shape arer external nodes.

A few points to note…

1. If a tree has n nodes then the number of branches it has is(n-1).
2. Except the root node every node in a tree has exactly one parent.
3. Any two nodes of a tree are connected by only one single path.
4. For a binary tree of height h the maximum number of nodes can be 2h+1-1.
5. Any binary tree with n internal nodes has(n+1)external nodes.

Threaded Binary Tree

Both the recursive and non- recursive procedures for binary tree traversal require that pointer4 to all of the free nodes be kept temporarily on a stock.It is possible to write binary tree traversal procedure that dose not require any pointer to the nodes be put on the stock.Sucj procedures eliminate the overhead (time and memory)involved in initializing,pushing and popping the stock.

In order to get an idea of how such binary-tree-traversal procedures work,let us look at the three shown in figure 9-17.

Here,first we follow the left pointer until we reach node C,without ,however ,pushing the poi8nter to A ,B is then printed ,after which C’s right pointer is followed to node E.Then the data from node E is printed .The next step in our in-oreder traversal is to go back to node B and print its data;however ,we did not save any pointers.But suppose that when we created the tree we had replaced the NULL right pointer of node E with a pointer back to nodeB.We could then easily follow this pointer back to node B.This is shown in figure 9-18.

Similarly ,suppose we replace the NULL right pointer of D with a pointer back up to A,as shown in figure 9-19.Then after printing the data in D,we can easily jump up to A and print its data.

The pointer that point in-oreder successor of a node are called right threads.Each right thread replace a normal right pointer in a tree node.Likewise ,we can have left threads that point to the in-order predecessor of a node.

The only problem with threads is that the coding requires that we know whether a pointer is a normal pointer to a child or a thread that point back to an in-order successor or in-order predecessor node.One solution to this problem is to add the data in each tree node two fields that indicates whether the left and right pointers in that node are normal pointer threads.For example ,these fields might be Boolean variable right is true if the right pointer is a thread and false if it is a pointer to right child.Likewise ,the variable left is true if the left pointer is a thread and false if it is a pointer to left child.If we add these Boolean variable to each tree node,we would make the following structure declaration for a node.

Struct thtree

{

Enum Boolean left;

Struct thtree\*leftchild;

In data;

Struct thtree\*rightchild;

Enum Boolean right;

};

The struct of a node of a threaded binary tree is shown in Figure9-20.

Thus each node would contain data,a left pointer ,a true or false value for left thread,a right pointer and a true or false value for right thread.

The program to implement a threaded binary tree is given below.The program shows how to insert nodes in a threases binary tree,delete nodes from it and traverse it in in-order traversal.

#include<stdio.h>

#include<conio.h>

#include(malloc.h>

#include<windows.h>

Ennum Boolean

{

False=0,

True=1

};

Struct thtree

{

Enum Boolean isleft;

Struct thtree\*left;

In data;

Struct thtree\*right;

Head->isright=false;

\*s =head;

z->left=head;/\*left thread to head \*

z->right=head;/\*left thread to head \*/

}

Else/\*if tree is non-empty\*/

{

P=head->left;

/\*traverse till the thread is found attached to the head\*/

While (p!=head)

{

If (p->data>num)

{

If (p->isleft!=true)/\*checking for a thread\*/

P=p->left;

Else

{

z->left=p->left;

p->left=z;

p->isleft=false;/\*indicates a link\*/

z->right=p;

return;

}

}

Else

{

If (p->data<num)

{

If (p->isright!=true)

P=p->right;

Else

{

z->right=p->right;

p->right=z;

p->isright=false;/\*indicates a link\*/

z->itleft=true;

z->isleft=p;

return;

}

}

}

}

}

}

/\*deletes a nide from the binary search tree\*/

Void del(struct thtree\*\*root,int num)

{

Int found;

Struct thtree \*parent,x,\*xsucc;

/\* if tree is empty\*/

If (\*root==NULL)

{

Printf(“Tree is empty./n”);

Return;

}

Parent =x=NULL;

/\*call to search function to find the node to be deleted\*/

Search (root,num,&parent,&x,&found);

If(found==false)

{

Printf(“Data to be deleted,not found.\n”);

Return;

}

/\*if the node to be deleted has two children\*/

If (x->isleft==false&&x->isright==false)

{

Parent=x;

Xsucc=x->right;

While(xsucc->isleft==false)

{

Parent =xsucc;

Xsucc=xsucc->left;

}

x->data=xsucc->data;

x=xsucc;

}

/\*if the node to be deleted has no child\*/

If (x->isleft==true&&x->isright==true)

{

/\*if node to be deleted is a root node\*/

If (parent==NULL)

{

(\*root)->left=\*root;

(\*root)->isleft=\*true;

Free(x);

Return;

}

If (parent->right==x)

{

Parent ->isright=true;

Parent->right=x->right;

}

Else

{

Parent->islefft=true;

Parent->left=x->left;

}

Free(x);

Return;

}

/\*if the node to be deleted has only rightchild\*/

If (x->isleeft==true &&x->isright==false)

{

/\*node to be deleted is a root node\*/

If (parent ==NULL)

{

(\*root)->left=x->right;

Free(x);

Return;

}

If(parent->left==x)

{

Parent->left=x->right;

x->right->left=x->left;

}

Else

{

Parent ->right=x->right;

x->right->left=parent;

}

Free(x);

Return;

}

/\*if the node to be deleted has only left child\*/

If (x->isleft==false&&x->isright==true)

{

/\*the node to be deleted is a root node\*/

If (parent==NULL)

{

Parent=x;

Xsucc=x->left;

While(xsucc->isright==false)

Xsucc= Xsucc->right;

Xsucc->right=\*root;

(\*root)->left=x->left;

Free(x);

Return;

}

If (parent->left==x)

{

Parent->right=x->right;

}

Else

{

Free(x);

Return;

}

}

/\*returns the address of the node to be deleted,address of its parent and

Whether the node is found or n ot \*/

Void search(struct thtree\*\*root,int num,structthtree\*\*par,

Struct thtree\*\*x,int \*found)

{

Struct thtree\*q;

Q=(\*root)->left;

\*found=false;

\*par=NULL;

While(q!=\*root)

{

/\*if the node to be deleted is found\*/

If(q->data==num)

{

\*found=true;

\*x=q;

Return;

}

\*par=q;

If (q->data>num)

{

If(q->isleft==true)

{

\*found=false;

X=NULL;

Return;

}

Q=q->left;

}

Else

{

If (q->isright==true)

{

\*found=false;

\*x=NULL;

Return;

}

Q=q->right;

}

}

}

/\*traverses the threaded binary tree inorder\*/

Void inorder(struct thtree\*root)

{

Struct thtree\*p;

P=root->left;

While(p!=root)

{

While(p->isleft==false)

P=p->left;

Printf(“%d\t”,p->data);

While(p->isright==true)

{

P=p->right;

If (p==root)

Break;

Printf(“%d\t”,p->data);

}

P=p->right;

}

}

Void deltree(struct thtree\*\*root)

{

While ((\*root)->left!=\*root)

Del (root,(\*root)->left->data);

}

Output:

Threded binary tree before deletion:

7 8 9 10 11 12 13 14 15

Threded binary tree after deletion:

7 8 9 11 12 13 14 15

Threaded binary tree after deletion:

7 8 9 11 12 13 15

Threded binary tree after deletion:

7 9 11 12 13 15

Threaded binary tree after deletion:

7 9 11 12 15

Now ,a brief explanation about the program.We have used an enumerated data type Boolean to store information whether the thread is present or not.If left is true it means tha there is a left thread and the node has no left child,if right is true it shows the presence of right thread and the node has no right child.

To insert a new node in a threaded tree,the insert()function is called which first checks for the empty tree.If the tree is found to be empty a head node is created and the node is joined as its left sub-tree with both true and leftchild and rightchild pointing back to the head node.Otherwise,the node is inserted into the tree so that a threaded binary se4arch tree is created.

Deletiom of a node from the threaded binary tree is similar to that of a normal binary tree.That,is we have to identify the four possibilities about the node being deleted:

a) No node in the tree contains the specified data.

b) The node has no children.

c) The node has exactly one child.

d) The node has two children.

The treatment given to these possibilities is same as the one discussed in the previous section on binary trees except for some minor readjustment of threads.

The threaded binary tree’s in-order traversal is different than a normal tree in the sense that we do not have to stack the pointers to nodes visited earlier so as to reach them later.This is avoided by using the threads to ancestors.The procedure to achive this is as follows:

This procedure begins by first going to the left sub-tree of the head node using the statement

P=root->leftchild;

Then through a while loop we follow the leftmost pointers until a thread to a predecessor is found .On encountering this thread we print the data for the leftmost node.Next,through another while loop we check the Boolean value of the right thread.If this value is true,we follow the thread back up to the ancestor node and print this ancentor node’s data.This way we continue to move up till the right thread is true.When the right thread is found to be false we again proceed by going to the right child and checking it’s left sub-tree.

As we follow these steps we are sometimes to reach the head node,and that is the time to stop the procedure.This is what is being achiveved from the statement.

If(p==root)

Break;

**Reconstruction Of A Binary Tree**

If we know the sequence of nodes obtained through in-order/pre-order/post-order traversal it may not be feasible to reconstruct the binary tree.This is because two different binary trees may yield same sequence of nodes when traversed using post-order traversal.

Similarly in-order or pre-order traversal of different binary trees may yield the same sequence of nodes.However ,we can construct a unique binary tree if the result of in-order and pre-order traversal are available .Let us understand this with the help of following set of in-order and pre-order traversal result:

In-order traversal:4,7,2,8,5,1,6,9,3

Pre-order traversal:1,2,4,7,5,8,3,6,9

We know that the first value in the pre-order traversal gives us the root of the binary tree.So the node with data 1 becomes the root of the binary tree.In in-order traversal ,initially the left sub-tree is traversal then the root node and the right sub-tree.So the data before 1 in the in-order list(i.e.4,7,2,8,5)forms the left sub-tree and the data after 1 in the in-oreder list(i.e.6,9,3)forms the right sub-tree.In figure 9-21(a) the structure of tree is shown after seprating the tree in left and right sub-trees.

Now the next data in pre-order list is 2 so the root node of the left sub-tree is2 .Hence the data before 2 in the in-order list(i.e.4,7)

Forms the left sub-tree of the node that contains a value 2.The data that comes to the right of 2 in the in-order list (i.e.8,5)forms the right sub-tree of the node with value2 .Figure 9-21(b) shows

Structure of tree after expanding the left and right sub-tree of the node that contains a value2.

Now the next data in pre-order list is 4,so the root node of the left sub-tree of the node that contains a value 2 is 4.The data before 4 in the in-order list forms the left sub-tree of the node that contains a value4.But as there is no data with value 4 in in-order list(i.e.7)

Forms the right sub-tree of the node that contains a value 4.Figure 9-21 ( c) shows the structure of tree after expanding the left and right sub-tree of the node that contains a value 4.

Since we are now left with only one value 7 in both the pre-order and in-order from we simply represent it with a node as shown in figure 9-21(d) .

In the same way one by one all the data are picked from the pre-order list and are placed and their respective sub-trees are constructed.As a result,the whole tree is constructed .Figure 9-21(e ) to 9-21(i) shows each step of construction of nodes one by one.

Here is the program that shows how to reconstruct a binary tree from the in-order and pre-order list.

#include<stdio.h>

#include<conio.h>

#include <malloc.h>

#defineMAX 101

Struct node

{

Struct node\*left;

In data ;

Struct node\*right;

}:

Void insert(struct node\*,int);

Void preorder(struct node\*);

Void postorder(struct node\*);

Void inorder(struct node\*);

Struct node\*recons(int\*,int\*,int);

Void deltree(struct node\*);

Int in [MAX],x,

Int main()

{

Struct node\*t;

Int req,I,num;

T=NULL;/\*empty tree\*/

System(“cls”);

Printf(“specify the number of item to be inserted:”);

While(1)

{

Scanf (“%d”,&req);

If (req>=max ll req <=0)

Printf(“Enter number between 1 to 100.\n”);

Else

Break;

}

For (i=0;i<req;i++)

{

Print)”Enter the data:”);

Scanf(“%d”,&num);

Insert(“&t,num);

}

Printf(“\n”);

Printf(“in-order Traversal:\n”);

X=0;

Inorder(t);

Printf(“\n”);

Printf(“pre-order Traversal:\n”);

X=0;

Preorder(t);

Printf(“n\”);

Printf(“post-order Traversal:\n”);

X=0;

Postorder(t );

Deltree( t);

T=NULL;

T=rercons(in,pre,req);

Printf(“\n”);

Printf(“After reconstruction of the binary tree:\n”);

X=0;

Printf(“\n”);

Printf(“in-order Traversal :\n”);

Inorder(t);

X=0;

Printf(“\n”);

Printf(pre-order Traversal :\n”);

Preorder(t);

X=0;

Printf(“\n”);

Printf(“post-order Traversal:\n”);

Postorder(t);

Deltree(t);

Return 0;

}

/\*insert a new node in a binary search tree\*/

Void insert (struct node\*\*sr,int num)

{

If (\*sr==NULL)

{

\*sr=(struct node\*) malloc (sizeof(struct node));

(\*sr)->left =NULL;

(\*sr)->data =num;

(\*sr)->right =NULL;

Return;

}

Else /\*search the node to which new node will be attached\*/

{

/\*if new data is less.traverse to left\*/

If(num<(\*sr)->data)

Insert(&((\*sr)->left),num);

Else

/\*else traverse to right \*/

Insert(&(\*sr)->right),num);

}

}

Void preorder(struct node\*t)

{

If(t!=NULL)

{

Printf(“%d\t”,pre[x++]=t->data);

Preorder(t->left);

Postorder(t->right);

}

}

Void postorder(struct node\*t)

{

If(t!=NULL)

{

Postorder(t->left);

Postorder(t->right);

Printf(%\t,t->data);

}

}

Void inorder (struct node\*t)

{

If (t!=NULL)

{

Inorder(t->left);

Printf(“%d\t”,in[x++]t->data);

}

}

Struct node\*recons(int\*inorder,int\*preorder,int noonfnodes)

{

Struct node\*temp,\*left,\*right;

Int tempin[100],temppre[100],I,j;

If (noofnodes==0)

Return NULL;

Temp=(struct node\*)malloc (sizeof (struct node));

Temp->data=preorder[0];

Temp->left=NULL;

Temp ->right=NULL;

If (noofnodes==1)

Return temp;

For(i=0;inorder[i]!=preorder[0];)

I++;

If (i>0)

{

For (j=0;j<=I;j++)

Temppre[i]=inorder[j+1];

}

left=recons(tempin,temppre,i);

temp->left=left;

if(i<noofnodes-1)

{

For(j=I;j<noofnodes-1;j++)

{

Tempin[j-i]=inorder[j+1];

Tempin[j-i]=preorder[j+1];

}  
}

Right =recons (tempin,temper,noofnodes-i-1);

Temp->right=right;

Return temp;

}

Void deltree(struct node\*t)

{

If (t!=NULL)

{

Deltree(t->left)

Deltree(t->right);

}

Free(t);

}

OUTPUT:

Specify the number of items to be inserted :5

Enter the data :1

Enter the data :2

Enter the data :3

Enter the data :4

Enter the data :5

In-order Traversal:

1 2 3 4 5

Pre-order Traversal:

1 2 3 4 5

Post-order

5 4 3 2 1

After reconstruction of the binary tree:

In-order Traversal:

1 2 3 4 5

Pre-order Traversal:

1 2 3 4 5

Post-order Traversal

5 4 3 2 1

Initially data is received, tree is constructed and is traversed in in-order,pre-order and post-order traversal. While traversing the tree two arrays pre[] and in [] are constructed to store the sequence of nodes. Then through the function deltree()the tree is deleted and a NULL value is stired in t which is pointer to tree. Then a function recons()is called to reconstructs the tree from the arrays pre[] and in [].This function returns a pointer to the root node of the reconstructed tree.Finally,again the tree is traverased in in-order,pre-order and post-order traversal to verify the reconstruction of the tree.

The working of the function deltree()is straight –forward.It checks whether the value of t is NULL.If it is not,two recuesive calls are made one for the left child of the current node and the other for the right child.After the control returns from the recursive calls the memory occupied by the current node is released.

The function recons()receives three parameters.First and second are the pointer to the in-order and pre-order arrays respectively.The third is theh number of nodes that a particular tree or sub-tree has.Inside the function two local arrays tempin and temper are created,to store the values of in-order and pre-order sub-trees respectively.Then a conditional is checked whether whether the number of nodes is 0.If it is then a NULL value is returned.If it iis not 0 then memory is allocated for a new node and its address is stored in a pointer temp.The element present at the 0th index n pre-order list is going to be the root node .A NULL value is stored in the data part of the new node.A NULL value is stored in left and right child of the new node.Then a condational is checked whether number of node is equal to 1.If it is 1,then the value of temp is returned which holds the address of the new node.If the sub-tree contains more than one node of the pre-order list in the in-order list.This is done because the left and right sub-trees are needs to be seprated and stored in tempin and temper arrays.

I order to seprate the left and right sub-tree it is checked whether the index of the first node of the pre-order list in nthe in-order list is more than 0.If it is then the respective children of the left sub-tree from the in-order and pre-order list in tempin and temppre respectively .This time the address returned by the function recons() is going to be the address of the root node of the right sub-tree for the current node.Hence it is stored in thr right part of the current node is returned which is stored in temp.

**General Trees**

A general tree can have any number of nodes. The children of a node are called as sibling of each other .In other words, if a particular node has four children,then theh second ,third and fourth child of that node are the sibiling of the first child.Figure 9-22 shows a general tree.

In figure 9-22 ,the node containing values 9,2,6 are sibiling of the node that contains a value 5.

While representing any tree it is necessary to have as many pointer in the node as the number of children that node has. However, each node in a general tree is likely to have different number of children. For example, some nodes may have 2 children, some may have 5 and more may have 4 children. Hence the structure of a node of a general tree should be declared in such a way that it contains variable number of pointer.This is parctially impossible.

To overcome this problem the maximum children occurring in any node in a general tree can be counted and then the structure with those many pointer can be decleareed.This however would lead to wastage of a large number of pointers.A better solution for this problem is to represent the general trees as binary trees.

General Trees Represented As Binary Trees

To represent the general trees as binary trees,two points are maintained.One pointer point to first child of a anode and another pointer point to the sibiling of the node.The structure of the nodes can be defined as follows:

Struct general

{

Int data ;

Struct general\*firstchild;

Struct general\*sibiling;

};

The in-memory view of a node of the tree is shown in figure 9-23.

While representing a general tree as a binary tree,the pointer to the first child of sa particular node is made the left child of that node and the right child is the pointer to the sibiling of the node.Thus while representing a general tree any node has only two children.Hence,a general tree can be represented as a binary tree.Figure9-24 shows the binary presenting of general tree in figure 9-22.

Though this tree dose’t appear as a binary tree unless it is rotated by450 as shown in figure 9-25.

From figure it can be observed that for every general tree represented as a binary tree,the righth sub-tree of the root node is always empty,as there aer no sibiling of the root node.

**Forest**

Forest is a set of several trees that are not linked to each other in any way.Forest can be represented as a binary tree.Initially the left most tree is represented as a binary tree exactly in the same way as explained in the previous section .Then the second tree is made the right child of the root node of the first tree and the third tree is made the right child of the root node of the second tree.Figure 9-26 shows how a forest can be represented as a binary tree.

AVL Trees

Searching in a binary search tree is efficient if the heights of both left and right sub-tree of any node are equal .However ,frequent insertion and deletions in a BST is likely to make it unbalanced.The efficiency of searching is ideal if the difference between the heights of left and right sub-trees of all the nodes in a binary search tree is at the most one.Such a binary search tree is called as a Balanced Binary Trees.It was invented in the year 1962 by two Russian mathematicians-G.M.Adelson –Velskii and E.M.Landis.Hence such trees are also known as AVL trees.Figure 9-27 shows some examples of AVL trees.

To represent a node of an AVL tree four fields are required-one for data,two for storing the address of the left and right child and an additional field is required to hold the balance factor of any node is calculated by subtracting the height of the right sub-tree of the node from the height of the left sub-tree.The structure of a node of an AVL tree is given below:

Struct AVL

{

Struct AVL\*left;

Int data ;

Struct AVL\*right;

Int balfact;

};

The value of balfact of any node is-1,0 or 1.If it is other than these three value then the tree is not balanced or it is not an AVL tree.

-If the value of balfact of any node is-1,then the height of the right sub-tree of that node is one more than the hreight of its left sub-tree.

- If the value of balfact of any node is 0 then the height of its left and right sub-tree is exactly same.

- If the value of balfact of any node is 1 then the height of the left sub-tree of that node is one more than the height of its right sub-tree.

The balfact value can also be represented\,-,/instead of -1,0,1.Figure 9-28 shows trees with value of balfact of each node.

Here,the tree( a) is an AVL tree,whereas ( b) is not .

Insertion of A Node In An AVL Tree

We can insert a new node in an AVL tree by finding its appropriate position.But insertion of new node involves certain overheads since the tree may become unbalanced due to the increase in its height.

If the new node is inserted as a child of any non-leaf node then there will be no effect on its balance ,as the height of the tree dose not increase .This is shown in figure 9-29.

If the new node is inserted as a child of leaf node than there is possibility that the tree may become unbalanced .This depends upon whether the node is inserted to a leaf sub-tree or to a right sub-tree ,which in turn changes the balance factor of the nodes.

If the new node is inserted as a child of leaf node of sub-tree(leaf or right )of shorter height then there will be no effect on the balance of an AVL tree.Figure 9-30shows the insertion of new node in the sub-tree of shorter height.

If the height of the left and the right sub-tree is same then the insertion of a new node on any of the leaf node dose not disturb the balance of an AVL tree.In this case even if the height of a sub-tree increase by one there will be no effect on the balance of the tree .Figure 9-31 shows how increase in height of one sub-tree dosen’t affect the balance of the tree.

If the new node is inserted as a child of the leaf node of taller sub-tree (left or right)theb the AVL tree becomes unbalanced and the tree no longer remains an AVL tree.This is shown in figure 9-32.

To re-balance abd make it an AVL tree the nodes need to be properly adjusted.So after insertion of a new node the tree is traversed starting from the new node to the node whose balance factor is disturbed.The nodes aer adjusted in such a way that the resultant tree becomes a balanced tree.

This can be explained with help of an example .Consider an AVL tree that is shown in figure 9-32.Suppose we want to insert a node with a value 13.This node would get inserted as the left child of thr node containing 15.On insertion the tree no longer remains an AVL tree as the balance factor of one of the nodes becomes-2.In general this condation can be represented as shown in figure 9-33.

Here the balance factor of P is -1 and that od Q is 0.ST1is the left child of the node P.ST2 and ST3 are the left and right child of the node Q.After insertion of a new node there are two cases that can make the tree an unbalanced tree.These are discussed below.

(a ) The new noe is inserted as a chikd (left or right )of the leaf node of sub-treeST3 This Shown in figure 9-34.

(b) The new node is inserted as a child (left or right )of the leaf node of sub-tree ST2 This shown in figure 9-35.

Let us now see how to achieve the balance in both these cases.

Case (a):

As seen from figure 9-32 ,on insertion of the new node the balance factor of the node containing the data 6 voilates the condation of an AVL tree.To re-balance the tree we are required to make left –rotation of the tree along the node containing the data 6. Left –rotation makes the node containing the data 6 as the left child of the node containing the data 12 and the node containing the data 10 as a right child of the node containing the data 6.This is shown in figure9-36.a

Case (b):

Now suppose instead of 13 we insert a node with value 11.This new node would get inserted as the right child of the node containing s value10.After this the tree no longer remains a balanced tree as the balance factor of node containing value 6 violates the rule of AVL tree.This is shown in figure 9-37.

To re-balance the tree we are required to make initially a right –rotation of the tree along the node containing a value 12.Right –rotation makes node 10 the right child of node 6.Node 12 becomes the right child of node 10 and node 11 becomes the left child of node 12.This is shown in figure 9-38.

But even now the tree is not balanced and hence,the tree is rotated to left aliong the node 6.As a result node 10 becomes the left child of the node20. The node 6 becomes the left child of the node 10.Since,there is no left child for node 10 the right child of node 6 is empty .Thus finally the three becomes a balanced binary tree or an AVL tree.This procedure of rotating the tree,first to right and then to the left is known as double rotation.Figure 9-39 shows the resultant tree that is an AVL tree.

There are two more possibilities wherer an AVL tree may become unbalanced due to insertion of new nodes.These are shown in figure 9-40( a) to figure 9-40( c).

To balance the tree shown in figure 9-40 (b) only a right rotation is required and to balance the tree that is shown in figure 9-40(c) a double rotation is required –initially a left rotation and then a right rotation.

**Deletion OF A Node from an AVL Tree**

The deletion of a node from an AVL tree is exactly the same as deletion of a node from a BST .Initially we need to search the node to be deleted.The node to be deleted could be a leaf node,a node with one child or a node with two children.We have already discussed the procedure to adjust the links in such cases.Only thing that remains to be done is to check the balance factor of each node of the tree after the deletion of the node .The process to re-balance the tree is exactly the same as we discussed in case of inserting a node in an AVL tree.Following program implememts an AVL tree.

#include <stdio.h>

#include<conio.h>

#include<malloc.h>

#include<windows.h>

#defineFALSE 0

# define TRUE 1

Struct AVL Node

{

Int data;

Int dalfact;

Struct AVLNode\*left;

Struct AVLNode\*right;

};

Struct AVLNode\*buildtree(struct AVLNode\*,int,int\*);

Struct AVLNode\*deldata(struct AVLNode\*,int,int\*);

structAVLNode\*del(structAVLNode\*,structAVLNode\*,int\*);

structAVLNode\*balright(structAVLNode\*,int\*);

structAVLNode\*balleft(structAVLNode\*,int\*);

void display (struct AVLNode\*);

void deltree (struct AVLNode\*);

int main()

{  
 struct AVLNOde \*avl=NULL;

Int h;

System (“cls”);

Avl=buildtree(avl,20,&h);

Avl=buildtree(avl,6,&h);

Avl=buildtree(avl,29,&h);

Avl=buildtree(avl,5,&h);

Avl=buildtree(avl,25,&h);

Avl=buildtree(avl,32,&h);

Avl=buildtree(avl,10,&h);

Avl=buildtree(avl,15,&h);

Avl=buildtree(avl,27,&h);

Avl=buildtree(avl,20,&h);

Avl=buildtree(avl,13,&h);

Printf (“AVL tree:\n”);

Display (avl);

Avl=deldata(avl,20,&h);

Avl=deldata(avl,12,&h);

Printf(“\n”);

Printf(“Avl tree after deletion of a node:\n”);

Display(avl);

Deltree (avl);

Return0;

}

/\*insert an element into tree\*/

Struct AVLNode \*buildtree(struct AVLNode \*root,int,data,int\*h)

{

Struct AVLNode \*node1,\*noe2;

If(!root)

{

Root =(struct AVLNode \*)malloc (sizeof(struct AVLNode));

Root ->data=data;

Root->left=NULL;

Root->right=NULL;

Root->balfact=0;

\*h=TRUE;

Return(root);

}

If (data<root->data)

{

Root ->left=buildtree(root->left,data,h);

/\*if left subtree is higher\*/

If(\*h)

{

Switch (root ->balfact)

{

Case:1

Node=1root->left;

If (node1->balfact==1)

{

Printf(“Right rotation along%d.\n”,root->data);

Root ->lrft=node1->right;

Node1->right=root;

Root->balfact=0:

Root=node1;

}

Else

{

Print(“Double rotation,left along %d”,noed1->data);

Node2=node1->right;

Node 1->right=node2->left;

Printf(“then right along%d.\n”,root->data);

Node2->left=node1;

Root ->left=node2->right;

Node2->right=root;

If(node2->ba;coat==1)

Root->balfact=-1;

Else

Node1->balfact=0;

Root=node2;

}

Root->balfact=0;

\*h=FALSE;

Break;

Case0;

Root->balfact=0;

Break;

Case-1;

Root-.balfact=0;

\*h=FALSE

}

}

}

If (data>root->data)

{

Root ->right=buildtree(root->right,data,h);

/\*If the right subtree is higher\*/

If (\*h)

{

Root ->right=buildtree(root->right,data,h);

/\*if the right subtree is higher\*/

If (\*h)

{

Switch (root->balfact)

{

Case1:

Root->balfact=0;

\*h=FALSE;

Break;

Case0:

Root->balfact=-1;

Break;

Case-1:

Node1=root->right;

If (node1->balfact==-1)

{

Printf(“Left rotation along %d.\h”,root->data);

Root->right=node1-.left;

Node1->left=root;

Root->balfact=0;

Root=node1;

}

Else

{

Printf (“Double rotation ,right along %d”,node1->data);

Node2=node1->left;

Node2->left=node2->right;

Node2->right=node1;

Printf(“then left along %d.\h”,root->data);

Node2->left=root;

If (node2->balfact==-1)

Root->balfact=1;

Else

Root->balfact=0;

If(node2->balfact==1)

Node1->balfact=-1;

Else

Node1->balfact=0;

Root=node2;

}

Root->balfact=0;

\*h=FALSE;

}

}

}

/\*deletes an item from the tree\*/

Struct AVLNode \* deldata(struct AVLNode\*root,int data,int\*h)

{

Struct AVLNode \*node;

If(!root)

{

Printf(“No such data.”);

Return(root);

}

Else

}

If (data<root->data)

{

Root->left=deldata(root->left,data,h);

If (\*h)

Root=balright(root,h);

}

Else

{

If(data>root->data)

{

Root->right=deldata(root->right,data,h);

If(\*h)

Root=belleft(root,h);

}

Else

{

Node=root;

If(node->right==NULL)

{

Root=node->left;

\*h=True;

Free(node);

{

Else

{

If(node-.left==NULL)

{

Root=node->right;

\*h=TRUE;

Free(node);

}

Else

{

Node->right=del(node->right,node,h);

If (\*h)

Root=balleft(root,h);

}

}

}

}

}

Return(root);

}

Struct AVLNode\*del(struct AVLNode\*succ,struct AVLNode\*node,int\*h)

{

Struct AVLNode\*temp=succ;

If (succ->left!=NULL)

{

Succ->left=del(succ->left,node,h);

If(\*h)

Succ=balright(succ,h);

}

Else

{

Temp=succ;

Node->data=succ->data;

succ=succ->right;

free(temp);

\*h=TRUEW;

}

Return(succ);

}

/\*balance the tree,if right sub-tree is higher\*/

Struct AVLNode\*balright(struct AVLNode\*root,int\*h)

{

Struct AVLNode\*node1,\*node2;

Switch(root->balfact)

{

Case1:

Root->balfact=0;

Break;

Case0;

Root->balfact=0;

Break;

Case-1;

Node1=root->right;

If (node1->balfact<=0)

{

Print(“Left rotation along%d.n”,root->data);

Root->right=node1->left;

Node1->left=root;

If (node1->balfact==0)

{

Root->balfact—1;

Node1->balfact=1;

\*h=FALSE;

}

Else

{

Root ->balfact=node1->balfact=0;

}

Root=node1;

}

Else

{

Printf(“Double rotation,right along%d”,node1->data);

Node2=node1-.left;

Node1->left=node2->right;

Node2->right=node1;

Printf(“then left along %d.\h”.root->data);

Root->right=node2->left;

Node2->left=root;

If (node2->balfact==-1)

Root-.balfact=1.

Else

Root->balfact=0;

If (node2->balfact==1)

Node1->balfact=-1;

Else

Node1->balfact=0;

Root=node2;

Node2->balfact=0;

}

}

Return(root);

}

/\*balances the tree,if left sub-tree is higher \*/

Struct AVLNode\*balleft(struct AVLNode\*root,int\*h)

{

Struct AVLNode\*node1,\*node2;

Switch (root->balfact)

{

Case-1:

Root->balfact=0;

Break;

Case0;

Root->balfact=1;

\*h=FALSE;

Break;

Case 1:

Node1=root->left;

If (node1->balfact>=0)

{

Printf(“Right rotation along%d.\h”,root->data);

Root->left=node1->right;

Node1->right=root;

If (node1->balfact==0)

{

root->balfact=1;

node1->balfact=-1;

\*h=FALSE;

}

Else

{

Root->balfact=node1-.balfact=0;

}

Root=node1;

}

Else

{

Printf(“Double rotation,left along%d”,node1->data);

Node2=node1->right;

Node1->right=node2->left;

Node2->left=node1;

Printf(“then right along %d.\n”,root->data);

Root->left=node2->right;

Node2->right=root;

If (node2->balfact==1)

Root->balfact—1;

Else  
 root->balfact=0;

If (node2->balfact==-1)

Node1->balfact=1;

Else

Node1->balfact=0;

Root=node2;

Node2->balfact=0;

}

}

Return(root);

}

/\*displys the tree in-order\*/

Void display (struct AVLNode\*root)

{

If(root!=NULL)

{

Display (root->left);

Printf(“%d\t”,root->data);

Display(root->right);

}

}

/\*deletes the tree\*/

Void deltree (struct AVLNode\*root)

{

If (root!=NULL)

{

Deltree(root->left);

Deltree(root->right);

}

Free(root);

}

Output:

Left rotation along 6.

AVL tree:

5 6 10 12 13 15 20 25 27 29

32

AVL tree after deletion of a node:

5 6 10 13 15 20 25 27 29 32

In the program initially eleven nodes are created and then two nodes are deleted.After deletion of the node since the tree become unbalanced,it is balanced by doing appropriate rotations.The function buildtree(),del(),balleft()and balright().Finally,a function deltree()is called that deletes the entire tree by releasing the memory occupied by tree.

The function buildtree() is used to add a new node to the tree. it receive three parameters;the first is the address of root node of the tree or sub-tree to which the new node is to be added.Thesecond is an integer that holds the data of the node that is to be added and the third is the address of a variable that is used as a flag to check whether there is a need for balancing the tree after addition of the new node.

In the finction buildtree() it is checked whether root is NULL.If it is,then the tree is empty and the new node is going to be the first node.Now memory is allocated for a new node.Next,data is stored in the data part,NULL in the left and right part of the new node it going tobe the leaf node.

If the tree is non-empty then the new node is added child of the leaf node.To determine whether the new node should be a leaf child or right child another it is checked whether the data of the new node is less than the data of the current node.If it is ,then a recursive call is made to function buildtree () by passing the address of the left sub-tree.If the left sub-tree is empty then the new node is made the left child of the current node.Then using if (\*h) it is checked whether there is a need for balancing.

If balancing needs to be done then a switch-case is applied on the balfact of the current node.If the balfact of current node is 1 (left sub-tree of current node is higher)then it is checked whether the balfact of left child of current node is 1.If it is,then simply a right rotation is required along the scurrent node,otherwise a double rotation is required.After rotation,balfact of current node is assigned a value 0 and a FALSE value is stored in the flag variable pointed to by h.If balfact of current node is 0,balfact is simply assigned a value 1.If balfact of current node is -1,then balfact is assigned a value 0 and FALSE value is stored in the flag variable pointed to by h.

There is one more possibility –the data of new node is greater than data of the current node.If it is thena recursive call is made to function buildtree() by passing the address of the right child of the current node.Here too the switch-case is applied if the height of the right sub-tree is higher,and appropriate rotation is applied,if needed .Finally ,the address of the current node is returned.

The function deldata ()works similar to the function buildtree to the function buldtree().Here also the recursive call is made foe either left or right child depending upon the data to be deleted.If data is found then its in-order successor is searched and a call is made to function del() which deletes the node.The function balleft()and balright ( )are called to balance the tree after deletion of the node.

The function display () is nothing but in-order traversal of the tree which we have already discussed.

2-3 Trees

The basic idea behind maintaining a search tree is to make the insertion ,deletion and searching operations efficient .In AVL trees the searching operation is efficient .However,insertion and deletion involves rotation that makes the operation complicated.To eliminate this complication a data structure called 2-3 tree can be used .To build a 2-3 tree there are certain rules that need to be followed.These rules are as follows:

1. All the non-leaf nodes in a 2-3 tree must always have two or three non-empty child nodes that are again 2-3 trees.
2. The level of all the leaf nodes must always be the same.
3. One single node can contain either one always be the same.
4. If any node has two children (left and right)then that node contain single data.The data occurring on left sub-tree of that node is less than the data of the node and the data occurring on right sub-tree of that node is greater than the data of the node.
5. If any node has three children (left,middle and right),then thatnode contains two data value ,let say I and j,Where i<j.The data of all the nodes on the left sub-tree are less than i.The data of all the nodes on the middle sub-tree are greater than I but less than j and the data of all the nodes on the right sub-tree are greater than j.
6. Figure 9-41 shows a 2-3 tree.

The structure of node of a 2-3 tree is as follows:

Struct twothree

{

Int count;

Int data[3];

Struct twothree\*child[4];

};

Here,the array data has 3 elements even though any node of the tree contains maximum two values.Also,the array child has 4 elements even thoughg any node of the tree has maximum three children.The reason behind this is explained in the section “Insertion Of A Value In A 2-3 Tree”later.

**Searching For A Value In A 2-3 Tree**

The process of searching data in a 2-3 tree starts from the root node of the tree.Cosider figure 9-41 .Suppose we want to search the value 17.It is first compared with the root node of the tree i.e.20.Since 17 is less than 20 the comparison will proceed in the left sub-tree of the tree,i.e.the node which contains two data values,6and 15.Since 17 is greater than 6,it is compared with 15

Anfd as 17 is greater than 15 too,the comparison process will process in the right sub-tree of the node containing 6 anmd 15.Here ,since the right sub-tree contains the value 17,the search is successful.

**Insertion Of A Value In 2-3 Tree**

Let us now try to understand the process of insertion of a value in a2-3 tree.To insert a value in a 2-3 tree we first need to search the position where the value can be inserted,then the value and the node in which the value is to be inserted are adjusted .The 2-3 tree grows in the reverse direction.This is a bit odd,hence let us understand this with the help of an example.

Consider the tree shown in figure 9-41 and suppose we want to insert a value 2.To insertthe value,first we need to search the appropriate position for the value.To search the position the method that we had already seen is followed.The searching process will terminate at the leaf node that contains the data 3 and 5.The actual insertion process atarts here.The value that is to be inserted i.e.2,is added to this node.So the node now contains the values2,3and 5.This is shown in figure 9-42

Adding 2 to the node that already contains 3 and 5 violates the definition of a 2-3 tree.To again make it a 2-3 tree,the median of the three values 2,3 and 5 is taken and that value is shifted to the parent of this node.In our case,the median value happens to be 3,which is moved up to the parent node.As a result the parent node now contains 3,6 and 15.Also ,the node containing 2 and 5 is now split into two different nodes containing value 2and 5.These two nodes are then attached as first and second child of the parent node containing the data 3,6and 15. This is shoen in figure9-43.

Now the node containing value 3,6, and 15 violates the rule of a 2-3 tree.So again the same process of finding the median of the three values and shifting that value to the parent node is repeated.In our case 6 is shifted to its parent node that contains a value 20.This time the node that contains a value 3 and 15 is split in such a way that 3 forms the left sub-tree and 15 forms the middle sub-tree of the root node that contains the values 6 and 20.

Also,the old child that contains a value 5 and 9 now becomes the right and left child of the nodes that contains values 3 and 15 respectively.The process of inserting the new value to the 2-3tree cases here as all the nodes satisfy the condations of a 2-3 tree.The final 2-3 tree is shown in figure 9-44.

From figure 9-44 it is clear that during the process of insertion some node may contain three data values and four children.This is merely an intermediate step.Hence ,while defining the structure of the node of a 2-3 tree it is necessary to declare an array of three elements for data value and an array of four elements for pointer to child nodes.

**Deletion Of A Value From A 2-3 Tree**

Deletion of a value from a 2-3 tree is exactly opposite to insertion.In case of insertion the node where the data is to be inserted is split if it already contains maximum number of values(i.e.two values).But in case of deletion,two nodes are merged if the node of the value to be deleted contains minimum number of values(i.e.only one value).

Let us understand this with the help ofan example .Consider the tree shown in figure 9-41.Suppose the node that contains aa value 17 is to be deleted.Its parent holds the values 6 and 15,and 15 is predecessor of 17.Hence 17 is replaced by 15 and is then merged with its sibiling ,i.e.with 9.Then the node that contains 9 and 15 is made the right child of the node that initially contained the values 6 and 15,and now contains only a single value 6 (as 15 is shifted to the place of deleted node.)This is shown in figure 9-45.

**B-Trees**

The number of values that a particular node of a binary tree or an AVL tree can hold is only one.On the other hand a 2-3 tree can contain at the most two values per node.To improve the efficiency of operations performed on a tree we need to reduce the height of the tree .Another problem arises when the data is stored in secondary storage medium is very high.Hence if we access the data less number of times,less would be the time required to perform an operation .If a node contains more number of value then at a atime more values can be accessed from the secondary medium.To improve the efficiency of tree operations Multi-way Search trees can be used.

**Multi-Way Search Trees**

A multi –way tree of order n is a tree in which any node may contain maximum n-1 value and can have maximum of n children.Order of a tree as we have seen earlier ,is the maximum number of the child nodes that a particular node has.In a multi-way tree of order 4(or a 4-way tree)any node can contain maximum three value and four children .Figure 9-46 shows a multi-way tree of order 4.

**Defination Of B-Tree**

B-Tree is a multi-way search tree of order n that satisfies the following condations:

1. All the non-leaf nodes (except the root node)have at least n\2 children and at the most n children.
2. The non-leaf root node may have at the most n non-empty child and at least two child nodes.
3. A B-tree can exist with only one node i.e.the root node containing no child.
4. If a node has n children then it must have n-1 values.All the values of a particular node are in increasing order.
5. All the values that appear on nthe left most child of a node are smaller than the first value of that node.Alll the values that appear on the right most child of a node are greater than the last value of that node.
6. If x and y are any two ith and (i+1) th  values of a node,where x <y,then all the values appearing on the (i+1) th  sub-tree of that node are greater than x and less than y.
7. All the leaf nodes should appear on the same level.

Figure 9-47 shows a B-Tree order3.

From Figure 9-47 it can be observrd that a B-tree of order 3 is a 2-3 tree.

The structure of a node of a B-tree is similar to the structure of a node of2-3 tree.This structure is given below:

Struct btnode

{

Int count;

Int value[MAX+1];

Struct btnode\*child [MAX+1];

};

Here count represents the number of children that a particular node has.

The values of a node stored in the array value.The addredd of child nodes are stored in the child array.The MAX macro signifies the maximum number of values that a particular node can contain.

**Searching Of A Value In A B-Tree**

Searching for a value k in a B-tree is exactly similar to searcjhing for value in a 2-3 tree.To begin with the value k is compared with the first value key[0] of the root node.If k is less than key [0]then the search is done in the first child node or sub-tree of the root node .If k is greater than key[0]then it is com[ared with key[1].If k is greater than k[0]and similar than key[1]then k is searched in the second child node or sub-tree of the root node.If k is greater than than last value key [i]of the root node then searching is done in the last child node or sub-tree of the root node.

If k is searched in any of the child nodes or sub-tree of the root node then the same procedure of searching is repeated for that particular node or sub-tree.

If the value k is found in the tree then the search is successful .The address of the node in which k is present and the position of the value k in that node is returned .If the value k is not found in the tree,then the search is successful.

**Insertion Of A Value In A B –Tree**

Let us now try to understand the process of insertion of a value in a B-tree.To insert a value in a B-tree firstly we need to search the position of the node where the value can be inserted and then the value and the node are adjusted if required .The node where the new value is to be inserted would always be the leaf node.

Suppose a value k is to be inserted in a B-tree of order4.Here the maximum number of value that any node may have is3.After searching for the appropriate leaf node to insert the value k,the value present at that particular leaf node is counted.This leads to two possibilities:

1. The leaf node is not full (dosen’t contain 3 values).
2. The leaf node is full (contains 3 values).

Different methods are used for inserting new value in these two cases.these are explained below:

**Case( a) :**

If the leaf node is not full then the value is inserted at its appropriate position in the node and the insertion procedure ends.Let us understand this with the help of an example.

Consider figure 9-46.Suppose we intend to add a value 37 to the tree.for this firstly the node where this value can b e inserted is searched .In our case it is the leaf node that contains the values 32 and 40. Since this node contains only two values the third value can be added and hence the insertion procedure ends here.The value 37 is inserted in between 32 and 40 because the values of any node of a B-tree should always appear in ascending order.Figure 9-48 shows the tree after insertion of the value 37.

**Case ( b):**

If the leaf node is full ,then that node is split into two nodes.If m is the order of the tree then any node is always split after the value m\2-1.As a result,the first part of the node contsind the first m\2-1.values and m\2 children if any ,and the second paprt of the node contain the last m-m/2 values and m-m/2+1 children,if any,Then the (m/2)th value is moved up to the parent node and the new value is appropriately attached to one of the two split nodes.If the parent node is full then the same procedure is repested.

Let us understand this with the help of an example .Consider the tree shown in figyre 9-48.Suppose the value that is to be inserted is 19.To begin with,the node in which 19 can be inserted is searched.In our case it is the node that contains values 2,13 and 22.Since this node already contains three values no more values can be added to this node.hence the node is split after the value 4 /2-1(Since order of the tree is 4).As a result,the first part of the node contains the first value (i.e.2)and has two children that are as yet empty.The second part of the node contains the last value 22 and has two children,which too are empty.The value 13 is moved up to the parent node and the new value 19 is attached to second split node,as it is greater than 13.

Finally ,these two split nodes are made the first and second children of the node containing the values 13,27 and 42 .This is shown in figure 9-49.

**Deletion Of A Value From A B-Tree**

Deletion of a value from the B-tree is similsr to insertion .Initially ,we need to find the node from which the value is to be deleted.After the deletion of the value,we need to check,whether the tree still maintains the property of the B-tree or not.Let us try to understand this with the help of an example .

Consider figure 9-50 that shows a B-tree of order5.To delete a value firstly its node is searched,the value is deleted and the number of remaining value in the node is counted.After counting the values there are two possibilities,which are discussed below:

**Case A:**

The number of values is greater than or equal to be minimum number of value required (i.e.2)for a B-tree of order5.

Consider Figure 9-50 and suppose we want to delete the value 64.On deleting this value the number of values that are left in the node are 2,which satisfy the condation of a B-tree.Hence the deletion process comes to an end .Figure 9-51 shows the B\_tree after the deletion of the value64.

**Case B:**

The left or right sibiling of the node from which the value is deleted contains more than the required minimum number of values.

Here the value of its (node from which the value is deleted)parent is moved to the node and a value from its sibling (left or right which contains more number of values than the required minimum values)is moved to its parent.This can be understood with the help of following example.

Consider the B-tree shown in figure 9-50 and suppose we want to delete the value 92. After deleting the value 92,value 79 is moved from its parent to the node from where the value is deleted.Then the value 71 is moved from its left sibling to its parent.The deletion process ends here as all the nodes satisfy the condation of B-tree.This is shown in figure9-52.

**Case ( b):**

The left or right sibling of the node from which the value is deleteeeed contains exactly the required nminimum number of values.

Here the value of its (node from which the value is deleted)parent is mo0ved to the node and the node is mrged with its sibling.If the parent also contains the minimum number of values then the same procedure of merging the node with its sibling is applied.Let us understand this with the help ofan ecample.

Consider the B-tree that is shown in Figure9-52 and suppose we want to delete the value 42.After deletion of the value 42 the node contains only one value 37.So value 32 is copied from its parentand it is with left sibling that contains values 19 and 27.As a result the node now contains four values 19,27,32 and 37.

This is shown in figure 9-53.

Now the parent node contains only one value 14.So the value 53 is copied from its parent and is merged with its right sibling that contains values 71 and 96.As a result,the node now contains four values 14,53,71 and 96.The deletion process ends here as all the nodes satisfy the condation of B-tree.This is shown in figure9-54.

Let us put all the theory that we learnt into practice.Here is a program that implements B-tree of order5.

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<malloc.h>

#include<windows.h>

#defineMAX4

#defibeMIN2

Struct btnode

{

Int count;

Int value [MAX+1];

Struct btnode\*child[MAX+1];

};

Struct btnode\*insert(int,struct btnode\*);

Int setval (int,struct btnode\*,int\*,struct btnode\*\*);

Int searchnode(int,struct btnode\*,int\*);

Void fillnode(int,struct btnode\*,struct btnode\*,int);

Void split (int,struct btnode\*,struct btnode\*,int);int,int\*,struct btnode\*\*);

Struct btnode\*del(int,struct btnode\*);

Int delhelp(int,struct btnode\*);

Voiod clear (struct btnode\*,int);

Void copysucc(struct btnode\*,int);

Void restore(struct btnode\*,int);

Void rightshift(struct btnode\*,int);

Void leftshift(struct brnode\*,int);

Void merge(struct btnode\*,int);

Void display(struct btnode\*,);

{

Struct btnode\*root;

Root +NULL;

System(“cls”);

Root=insert(27,root);

Root=insert(42,root);

Root=insert(22,root);

Root=insert(47,root);

Root=insert(32,root);

Root=insert(2,root);

Root=insert(51,root);

Root=insert(40,root);

Root=insert(13,root);

Print (“B-tree of order 5:\n”);

Display (root);

Root=del(22,root);

Root=del(11,root);

Printf(“\n”);

Printf(“After deletion of values:\n”);

Display(root);

Return 0;

}

/\*insert a value in the B-tree\*/

Struct btnode\*insert(int val,struct btnode\*root)

{

Int I ;

Struct btnode \*c,\*n;

Int flag;

Flag =setval(val,root,&I,&c);

If(flag)

{

n=(struct btnode\*)malloc(sizeof (struct btnode));

n->count=1;

n->value[1]=I;

n->child[0]=root;

n->child[1]=c;

return n;

}

/\*sets the value in the node\*/

Int setval(int val,struct btnode\*n,int\*p,struct btnode\*8c)

{

Int k;

If(n==NULL)

{

\*p=val;

\*c=NULL;

Return 1;

}

Else

{

If (searchnode(val,n,&k))

Print(“Key value already exists .\n”);

If (setval (val,n->child[k],p,c))

{

If (n->count<MAX)

{

Fillnode(\*p,\*c,n,k);

Return 0;

}

Else

{

Split (\*p,\*c,n,k,p,c);

Return 1;

}

}

Retirn 0;

}

}

/\*searches value in the node \*/

Struct btnode\*search(int val,struct btnode\*root,int \*pos)

{

If (root==NULL)

Return NULL;

Else

{

If(searchnode(val,root,pos))

Return root;

Else

Return search (val,root->child[\*pos],pos);

}

}

/\*searches for the node\*/

Int searchnode(int val,struct btnode\*n,int\*pos)

{

\*pos=0;

Return0;

}

Else

{

\*pos=n->count;

While((val<n->value[\*pos]&&\*pos>1)

(\*pos)--;

If (val==n->value[\*pos])

Return1;

Else

Return0;

}

}

/\* adjusts the value of the node\*/

Void fillnode (int val,struct btnode \*c,struct btnode\*n,int k)

{

Int I;

For (i=n->count;i>k;i--)

{

n->value[i+1]=n->value[i];

n->child[i+1]=n->child[i];

}

/\*splits the node\*/

Void split(int val,struct btnode\*c,struct btnode\*n,int k,int\*y,struct btnode\*\*newnode)

{

Int I,mid;

If (k<=MIN)

Mid=MIN;

Else

Mid=MIN+1;

\*newnode=(struct btnode\*)malloc(sizeof(struct btnode));

For (i=mid+1;i<=MAX;i++)

{

(\*newnode)->value[i-mid]=n->value[i];

(\*newnode)->child[i-mid]=n->value[i];

}

(\*newnode)->count=MAX-mid;

n->count=mid;

if (k<=MIN)

return 0;

}

}

/\*searches value in the node\*/

Struct btnode\*search (int val,struct btnode\*root,innt\*pos)

{

If(root==NULL)

Return NULL;

Else

{

If(searchnode(val,root,pos))

Return root;

Else

Return search(val,root->child[\*pos],pos);

}

}

/\*searches for the node\*/

Int searchnode(int val,struct btnode \*n,int \*pos)

{

If (val<n->value[1])

{

\*pos=0;

Return0;

}

Else

{

\*pos=n->count;

While((val<n->value[\*pos]&&\*pos>1)

(\*pos)--;

If(val==n->value[\*pos])

Return 1;

Else

Return 0;

}

}

/\*adjusts the value of the node\*/

Void fillnode (int val,struct btnode\*c,struct btnode\*n,int k)

{

Int I;

For (i=n->count :i>k;i--)

{

n->value [i\_1]=n->value[i];

n->child [i\_1]=n->value[i];

}

n-.value[k+1]=n->value[i];

n->child[k+1]=c;

n->count++;

}

/\*splits the node\*/

Void splilt (int val,struct btnode\*c,struct btnode\*n,int k,int\*y,struct btnode\*\*newnode)

{

Int I,mid;

If(k<=MIN)

Mid+MIN;

Else

Mid=MIN+1;

\*newnode=(struct btnode\*)malloc(sizeof(struct btnode));

For(i=mid+1;i<=MAX;i++)

{

(\*neewnode)->value[i-mid]=n->value[i];

(\*newnode)->child[i-mid]=n->child[i];

}

(\*newnode)->count=MAX-mid;

n->count=mid;

if (k<=MIN)

fillnode(val,c,n,k);

else

fillnode(val,c,\*newnode,k-mid);

\*y=n->value[n->count];

(\*newnode)->child[0]=n->child[n->count];

n->count--;

}

/\*deletes value from the node\*/

Struct btnode\*del(int val,struct btnode \*root)

{

Struct btnode\*temp;

If(!delhelp(val,root))

{

Printf(“\n”);

Printf(“value%d not found.\n,”val);

}

Else

{

If (root->count==0)

{

Temp=root;

Root=root->child[0];

Free(temp);

]

}

Return root;

}

/\*helper function for del()\*/

Int delhelp(int val,struct btnode\*root)

{

Int I;

Int flag;

If(root==NULL)

Return 0;

Else

{

Flag=searchnode(val,root,&i);

If (flag)

{

If(root->child[i-1])

{

Copysucc(root,i);

Flag=delhelp(root->value[i],,root->child[i]);\

If(!flag)

{

Printf (“\n”);

Printf(“value%d not found.\n”,val);

}

}

Else

Flag =delhelp(val,root->child[i]);

If (root->child[i]->count<MIN)

Restore(root,i);

}

Return flag;

}

}

/\*removes the value from the node and adjusts the value\*/

Void clear (struct btnode\*node,int k)

{

Int I;

For (i=k+1;i<=node->count;i++)

{

Node->value[i-1]=node->value[i];

Node->child[i-1]=node->value[i];

}

Node->count--;

}

/\*copies the successor of the value that is to be deleted\*/

Void copysucc (struct btnode\*node,int i)

{

Struct btnode\*temp;

Temp=node->child[i];

While(temp->child[i];

Temp=temp->child[0];

Node-.value[i]=temp->value[1];

}

If (i==0)

{

If (node->child[1]->count>MIN)

Leftshift(node,1);

Else

Merge(node,1);

}

Else

{

If (i==node->count)

{

If(node->child[i-1]->count>MIN)

Rightshift(node,i);

Else

Merge(node,i);

}

Else

{

If (node->child[i-1]->count>MIN)

Rightshift(node,i);

Else

{

If (node->child[i+1]->count>MIN)

Leftshift(node,i+1);

Else

Merge(node,i);

}  
 }

}

}

/\*adjusts the values and children while shifting the value from parent to right child\*/

Void rightshift(struct btnode\*node,int k)

{

Int I;

Struct btnode\*temp;

Temp=node->child[k];

for(i=temp->count;i>0;i--)

{

Temp->value[i+1]=temp->value[i];

Temp->child[i+1]=temp->value[i];

}

Temp->child[1]=temp->value[0];

Temp->count++;

Temp->value[1]=node->value[k];

Temp=node->child[k-1];

Node->value[k]=temp->value[temp->count];

Node->child[k]->child[0]=temp->child[temp->count];

Temp->count--;

}

/\*adjust the values and children while shifting the value from parent to left child\*/

Void leftshift(struct \*btnode\*node,intk)

{

Int=I;

Struct btnode\*temp;

Temp=node->child[k-1];

Temp->count++;

Temp->value[temp->count]node->value[k];

Temp->child[temp->]=node->child[k]->child[0];

For(i=1;i<temp->count;++)

Temp2->count++

Temp2->value[temp2->count]=temp->value[i];

Temp2->child[temp2->count]=temp->value[i];

}

For (i=k;i<node->count;++)

{

Node->value[i]=node->value[i+1];

Node->child[i]=node->value[i+1];

}

Node->count--;

Free(temp1);

}

/\*display the B-tree\*/

Void display(struct btnode\*root)

{

Int I;

If(i=0;i<root->count;i++)

{

Display(root->child[i];

Printf(“%d\t”,root->value[i+1]);

}

Display (root->child[i];

}

}

Output:

B-tree of order 5:

2 13 22 27 32 40 42 47 51

Value v11 not found.

After deletion of values:

2 13 27 32 40 42 47 51

In the program,from main() three function are called –insert(),del()and display ().These function in turn calls several function like leftshift(),rightshift(),delhelp(),etc.The function insert()is used to insert a value in the tree.The function del()is used to delete a value from the tree and the function display() is used to display the list in ascending order.

The function insert() receives two parameters-the address of the root node and the value that is to inserted.This function in turn calls a function setval() which returns a value 0 if the new value is inserted in the tree,otherwise it returns a value 1.If it returns 1 then memory is allocated for new node,the variable count is assigned a value 1 and the new value is inserted in the node.Then the addresses of the child nodes are stored in child pointer and finally the address of the node is returned.

The function setval()receive four parameters.The first is the value thatis to be inserted,second is the address of the node(root node,if called for the first time),third is an intef=ger pointer that points to a local flag variable defined in the function insert(0and last parameter is a pointer to pointer to the child node that will be set in a function(split()or fillnode())called from this function .

The function setval()returns a flag value that indicates whether the value is inserted or not.If the node is empty then this function returns a value1.If the the node is empty then this function calls a function searchnode()that checks whether the value already exsists in the tree.If the value alreadt exsists then a suitable messages is displayed.Then a recursive call is made to the function setval()for the child of the node.If this time the function returns a value 1 it means the value is not inserted.Then a condation is checked whether the node is full or not.If the is not full then a function fillnode()is called that fills the value in the node henc3e at this point a value 0 is returned .If the node is full then a function split ()called that split the exsisiting node.At this popint value 1 is returned to add the current value to the new node.

The function search() receives three parameters.The first parameters is the value to be searched ,second is the address of the node from where the search is to be performed and third is the address of a variable that is checked wheteher the address of the node being searched is NULL .If it is ,then simply a NULL value is returned.Otherwise ,a function searchnode() is called which actually searches the given value.If the search is successful the address of the node in which the value is found is retrned .If the search is unsuccessful then a recursive call is made to the search(0 function for the child of the current node.

The function searchnode()receive three parameters.The first parameter is the value that is to be searched.The second parameter is the address of the node in which the seqarch is to be performed and third is a pointer pos that holds the address of a variable in which the position of the value that once found is stored.This function returns a value 0 if the search in unsuccessful and 1 if it is successful.In this function initially it is checked whether the value that is to be searched is less than the very first value of the node.If it is then it indicates that we the value is not present in the current node.Hence,a value 0 is assigned in the variable that is pointed to by pos and 0 is returned,as the search is unsuccessful.

If the value to be searched is not less than the first value of the node then the value of the count is stored in the variable pointed to by pos.Then a while loop is executed till the time a value ,let’s say x (which is greater than the value that is to be searched )is found.Inside the loop each time the position of the value that is pointed to by pos is decrement by one.This is done in order to find the position of the value that is less than x.Hence,outside,the loop value present at the position stored in the variable pointed by pos.If it is then a value 1 is returned ,as the search if successful otherwise a value 0 is returned.

The function fillnode() receives four parameters.The first is the value that is to be inserted.The second is the address of the child node of the new value that is to be inserted.The third is the address of the node in which the new value is to be inserted.The last parameter is the position of the node where the new value is to be inserted.In the function fillnode(),initially,a for loop is executed which shifts all the value and their respective children one placeto right .One outside the loop the new value and its child is stored at the appropriate position and the variable count is increment by 1.

The function split () recursive six parameters.The first four parameters are exactly the same as in the case of function fillnode().The fifth parameter is a pointer to variable that holds the value from where the node is split.The last parameter is a pointer to pointer of the new node created whether the new value that is to be inserted is inserted at a position less then or equal to the minimum values required in a node.If the condation is satisfied then the node is split at the position MIN (the minimum required values of a node).Otherwise ,the node is split at the one position more than MIN.Then dynamically memory is allocated for a new node.Next,a for loop is executed which copies into the new node the values and children that occur on the right side of the value from where the old node is split.Outside the loop the value of count of new node and old node are set.The function fillnode()is called by passing the address of the new node or old node .The value from which the node is split is assigned to the variable that is pointed to by y.Finally,the child of the node from which the node is split is made the first child of the new node and then the value of old node is sdecrement by 1.

The function dek()receive two parameters.First is the value thay is to be deleted second is the address of the root node.This function calls another delhelp()which returns a value 0 if the deletion of the value is unsuccessfull ,1 otherwise .If it is unsuccessfull then the appropriate message is displayed.Otherwise ,a condation is checked whether the count is 0.If it is ,then it indicates that the node node from which the value is deleted is that last value.Hence,the first child of the node is itself made the node and the original node is deleted.Finall,the address of the new root node is returned.

The function delhelp() receives two parameters.First is the value to be deleted and the second is the address of the node from which it is to be deleted.If so,then a function copysucc()is called which copies the successor of the value to be deleted then a resusive call is made to the function delhelp()for the value that Is to be deleted and its child.If data is not found then the appropriate message is displayed.If the child is empty then a call to function clear () is made which deletes the value.If the searchnode()function fails then a recursive call is made to function delhelp() by passing the address of the child.Then a condation is checked whether the chiold of the node that is searched is empty or not.If is not empty,then another condation is checked to see whether the value present at this child is less than the minimum required value.If so,then a function restore() is called to merge rhe child with its siblings.Finally,the value of the flag is returned which is set as a returned value of the function searchnode().

The function clear()receive two parameters. First is the address of the node from which the value is to be deleted and second is the position of the value that is to be deleted.This function simply shifts the value one place to the left from the position where the value that is to be deleted is present .Hence, the value that is to be deleted is overwritten its successor value.

The function copysucc()receives two parameters.First is the address of the node where the successor is to be copied and second in the position of the value that is overwritten with its successor.Initally,the address of the first child of current node is copied into a tempraray variable and a while loop is executed till the time the leaf node is not reached. Inside the loop is excuted till the time the leaf node is not reached.Inside the loop each time the address of the first child of a node is assigned to the temprory variable so that the leaf node is reached.Finally,outside the loop the first value of the leaf node is assigned to the value that s to be deleted.

The function restore()receives two parameters.First is the node that is to be restored and second is the position of the value from where the value are restored.If seconed parameter is 0,Then another condation is checked to find out whether the values present at the first child are more than the required minimum number of value .If so,then a function leftshift() is called by passing the address of the node and a value 1 signifying that the value of this node I s to be left shifted from the first value.If the condation is not satisfied then a function merge()is called for merging the two children of the node.

If the node that is to be restored is not restored from the first position then in the else block of outer if another condation is checked.This tome the position of the value from where the values are restored is compared with the maximum numberof values that a particular node can contain.If it matched then another condation is checked whether the child of the node contains is also satisfied then a function rightshift()is called,otherwise,the function merge() is called .This time the second value passed to the function is the not 1 but I which holds the position from where the values are shifted or merges.

If the node is to be restored is not restored from the position 1 and count then there is else block where the condation is checked for the minimum number of the prevous child.If it is satisfied then the function rightshift() is called,otherwise ,the condation for next child is checked.If this is satisfied then the function leftshift() is called ,otherwise ,the function meerge() is called.

The function rightshift ()receives two parameters.First is the address of the node from where the value is shifted to its child ansd second is the position k of the value that is to be shifted.The address of the child of value present at position k is copied into a temprory variable temp.Then the value and the child pointed by temp are shifted one position to the right .The first child of the child node is made the second child after incrementing the count of the child.Finally,the value from the child which is present at the position-1 is shifted to the parent node and the last child of the (k-1)thchild is made the first child of the kth child.Then the count of the (k-1)th child is decremented by one.

The function leftshidt () receive two parameters.Both the parameters are exactly same as that of function rightshift().The working of the function is exactly opposite to the working of rightshift().

The function merge ()receive two parameters.First is the address of the node from which the value is to be copied to the child and second is the position of the value.In this function two temprory variables temp1 and temo 2 are defined to hold the address of the two children of the value that is to be copied .Initiallty the value of the node is copied to its child.Then the first child of the node is made the respective child of the node where the value is copied.Thentwo for loops are executed ,out of which first copies all the values and children of the node from where the value is copied.Thenthe count of the node from where the node is copied is decremented.Finally,the memory occupied by the second node is already by calling free().

The function display()receives only one-parameters-the addrsss of the node.If the tree/node is non-empty then a for loop is executed count number of times,i.e.as many time as the number of values present in the node.Inside the for loop there is a recrusive call to the function display ().The argument passed is the address of the first child node in the first interation,second child node in the second iteration and so on.After the recursive call the value of the current node is printed.Outside the for loop there is again a recrusive call to display ()function.This time the argument passed to this function is the address of the last child of the current node.This is done because the for loop executes only the count number of times and hence the last child is not traversed.

**Priority Queue**

In a priority queue all the elements are assigned some priority .The order in which the elements could be deleted or processed from the priority queue depends upon this priority .The element with the highest priority is accessed then the elements,with the second priority and so on.The elements with the same priority are accessed in the order in which they wrer added to the queue.

Operating system uses priority queue for scheduling jobs where jobs with higher priority are processed first.Priority queue is also used in time-sharing sysytems where the program with high priority are pro cessed first and a standard queue is formed for the programs with the same priority .A priority queue can be implemented using a heap as we would see in a later section.

**Heap**

Heap is a complete binary tree.There are two types of heaps.If the value present at any node is greater than all its children then such a tree called as the max-heap or decending heap.In case of a min-heap or ascending heap the value present in any node is smaller than all its children.Figure9-55 shows a descending heap.

**Priority Queue Represented As A Heap**

We have see how a binary tree can be represented by one-diminsal array.The nodes are numbered as 0 for root node,then from left to right at each level as 1,2 etc.For an I thnodeits leftand right child exist at (2i+1)thand (2i+2)thposition respectively.Same is the case with heap ,if the index of the root is consideredas 1 then the left and right child of ithnode are present at(2i)th and (2i+1)thposition respectively and the parent node is present at (i/2)thindex in the array.Figure 9-56 shows an array a that represent the heap shown in figure9-55.

The root node of the tree starts from the index 1 of the array .The 0th element is called as the sentinel value thatis a maximum value and is not the node of the tree.It can be any number,like say 1000.This value is required because while addition of new node certain operation are performed in a loop and to terminate the loop the sentinel value is used.

Theoperation that can be performed on a heap are insertion of a node,deletion of a node and replacement of a node.On performing these operation the tree may not satisfy the heap properties and hence must be re-constructed.

**Insertion Of A Node In A Heap**

To insert an element to the heap,the node is inserted after the last element in the array and is compared with its parent that is parent at(i/2)thposition.If it is found to be greater than its parent then thay are exchanged .This procedure is repeated till the element is placed at its appropriate place.Let us understand this with the help of an example.Suppose a node that contains a value 24 is inserted in the tree that is shown in figure9-55.

Firstly,the value is inserted at index 11 in the arrya,as the tree has 10 nodes.Then 24 is compared with its parent 22 and since 24 is greater than 22 they are exchanged .This is shoen in figure9-57.

Now the element 24 is compared with its new parent 32 and since 24 is less than 32 they are not exchanged.The insertion process ends here and it is the final position of the node 24 in the tree.

**Replacement Of A Node In A Heap**

To replace the node of highest priority (i.e the root node),the new value is inserted at the root of the node.Thenit is compared with its value children.If it is smaller than it is exchanged with the child that is greater among them.Let us understand this with the help of an example.

Suppose value present in the root node in figure 9-55(i.e.36)is replaced with 5.Then it is compared with its children32 and 29.Since 5 is smaller than these values it is exchandged with the greatest value(i.e.32),so the tree noe becomes as shown in Figure 9-58.

Now 5 is compared with its current children17 and 22 and since 5 is less than these value it is exchanged with the greatest value(i.e22),so the tree now becomes as shoen in figure 9-59..

Now 5 is compared with its children -15 and an empty node.Since 5 is less than 15 it is exchanged with 15.Here process of replacement ends.So the tree now becomes as shown in Figure9-60.

Deletion Of A Node From A Heap

Suppose the tree from which the node is to be deleted contained n nodes.Also the node that is to be deleted is always of the highest priority (i.e.the root node).Firstly the element present at the index n in the array is stored at the 1st position in the array and maximum index value of the array is decremented by one.Then the heap is restored as in the case of replace operation.Let us understand this with the help of example.

Suppose we want to delet a value from the heap shown in Figure 9-55.For this the value 15 is copied at the index 1 in array and then heap is restored.Figures9-61(a) to 9-61(c)shows steps involves in restoring the heap.

Construction Of A Heap

Heap can be constructed from an array.To begin with a tree is constructed .If the tree dosen’t satisfy heap properties then it is converted into a heap by adjusting nodes.

Adjustment of nodes starts from the level one less than the maximum level of the tree(as the leaf node are always heap).Each sub-tree of that particular level is made a heap.Then all the sub-trees at the level two less than the maximum level of three aremade heaps.This procedure is repeated till the root node.As a result the final tree becomes a heap.

Let us understand this with the help of example .Consider an array arr that contains 15 elements given below:

7,10,25,17,23,27,16,19,37,42,4,33,1,5,11

The tree that can be constructed from the array I s shown in figure 9-62.

To make ita heap initially the elements that are present at a level one less than the maximum level of the tree are taken into consideration.In our case,these are 17,23,27and 16.They are converted to heaps in the same way as replacing the node of a heap.The resultant tree is shown in figure9-63.

Now the elements that are present at a level two less than the maximumlevel of the tree are considered .In our case,these are 10 andd 25.These are also made the heap by the same procedure.The resultant tree is shown in figure 9-64.

Similarly ,each time one level is deceremented and all the sub-trees at thay level are converted to heaps.As a result ,finally the entire tree gets converted to a heap as shown in figure9-65.

Following program implements alllthe operations that can ne performed on a heap.

#includestdio.h>

#include<conio.h>

#include<windows.h>

Void restoreup(int,int\*);

Void restiredown(int,int\*,int);

Void makeheap(int\*,int);

Void add(int,int\*,int\*);

Int replace(int,int\*,int\*);

Int del(int\*,int\*);

Int arr[20]={1000,7,10.25,17,23,27,16,19,37,42,4,33,1,5,11};

Int I,n=15;

System(“cls”);

Makeheap(arr,n);

Printf(“heap:\n”);

For(i=1;i<=n;i++)

Printf(“%\t”,arr[i]);

I=24;

Ass(I,arr,&n);

Printf(“\n\n”);

Printf(“element added %d.\n\n”,i);

Printf(“heap after addition of an element:\n”);

For(i=1;i<=n;i++)

Printf(“%d\t”,arr[i]);

I=replace(2,arr,n);

Print(“n\n”);

Printf(“element replaced %d.\n\n”,i);

For(i=1;i<=n;i++)

Printf(“%d\t”,arr[i]);

I=del(arr,&n);

Printf(“n\n”);

Printf(“element deleted %d.\n\n”,i);

Printf(“heap after deletion of an element:\n”);

For(i=1;i<=n;i++)

Printf(“%d\t”,arr[i]);

Return0;

}

Void restoreup(int I,int\*arr)

{

Int val;

Val=arr[i];

While(arr[i/2]<=val)

{

Arr[i]=arr[i/2];

I=i/2;

}

Arr[i]=val;

}

Void restoredown(int pos,int\*arr,int n)

{

Int I,val;

Val=arr[pos];

While(pos<=n/2)

{

I=2\*pos;

If ((i<n)&&(arr[i]<arr[i+1]))

I++;

If (val>=arr[i])

Break;

Arr[pos]=arr[i];

Pos=I;

}

Arr[pos]=val;

}

Void makeheap(int\*arr,intn);

{

Int I;

For(i=n/2;i>=1;i--)

Restoredown(I,arr,n);

}

Void add(int val,int\*arr,int\*n)

{

(\*n)++;

Arr[\*]=val;

Restoreup(\*n,arr);

}

Int replace (inti,int\*arr,int n)

{

Int r=arr[1];

Arr[1]=I;

For(i=n/2;i>=1;i--)

Retoredown(I,arr,n);

Retirn r;

}

Int del(int\*arr,int\*n)

{

Int val;

Val=arr[1];

Arr[1]=arr[\*n];

(\*n)--;

Restoredown(1,arr,\*n);

Return val;

}

Output:

Heap:

42 37 33 19 23 27 16 7 17 10 4 25 1 5 11

Element added 24.

Heap after addition of an element:

42 37 33 24 23 27 16 19 17 10 4 25 1 5 11 7

Element replaced 42.

Heap after replacement of an element:

37 24 33 19 23 27 16 7 17 10 4 25 1 5 11 2.

Element deleted 37.

Heap after deletion of an element:

33 24 27 19 23 25 16 7 17 10 4 2 1 5 11

From main()the function makeheap()is called which builds the heap.Then by calling function like add(),replace()and del()operation like addition like addition ,deletion and replacement of the new node are performed on the heap.

The function maleheap ()receives two parameters-first is the base address of the array which holds the value of the nodes and second is the number of nodes in the tree.In makeheap()a for loop is executed which restores the heap by calling the function restoredown ().The loop is started from the (n/2)thelement as all the nodes present above the index n/2 are leaf nodes and they are always the heap.

The function restoredown ()receive three parameters.First is the index of the node from where the heap needs to be restored.Second is the base address of the array that holds the value of the nodes and third is the totalnumber of the nodes in the heap.The value present at the particular position from where the heap needs to be restored is stored in a temprory variable val.

A while loop is executed till the time the index of the array reaches n/2.As we discussed earlier this is the maximum position above which allthe nodes are heap as all of them are leaf nodes.Inside the while loop for each node the position of its left child is calculated which is stored in i.Then a condation is checked whether I is less than n and the left child of the current node is l;ess than the right child.If both the condation are true then the index I is increased by one,as we intend to store the value of child that is greater amongast them.Then another condation is checked to see whether the value of the node from where the heap needs to be restores is greater than or equal to the value of the left or right child of the current node.If it is,then the while loop is terminated.If the loop continues then the value of the respective child (left or right)is stored in the place of current node.Finally,the index of the child is stored in position of the current node,so that in the next iteration of while loop the whole procedure is repeated for its respective child.

One outside the while loop the value present in the temprory variable val is stored in the array at position pos,where pos is the position of the node for which the heap is restored.

The function add()receives three parameters-first is the value that is to be added,second is the base address of the array and third is a pointer to variable that holds maximum index of the array.

To begin with,using the pointer ,the maximum index of array is incremented by one.Then the new value is stored at the highest position in the array.Finally,a function restoreup()is called which restores the heap up,so that the new value is placed at its appropriate position in the heap.

The function restoreup()is the counter-part of the function restoredown().It receive two parameters-first is the total number of nodes parent in the heap and second is the base address of the array.Initially,the new value that is the base address of the array.Initially ,the new value that is added is stored in a temprory variable val.Then a while loop is executed by checking the condation whether the value present at the parent node is less than the current value.If it is,then the value present at its parent is stored in the current node.Then the index I is assigned a value of its parent’s indexi.e.i/2.As a result ,at the end of while loop the value present in I is the index where the new value needs to be added.Hence,outside the while loop the value present in val is stored in the array at the ithindex.

The function replace()receives three parameters.First is the value that is to be added,second is the base address of the array ,and the third is the total number of the nodes present in the heap.

Initially ,the value present at index 1 is stored in a temporary variable r.This is the root node of the heap,as the node that is replaced is always the root node.Then the new value is added at the root node.Then the new value is added at the root node and a for loop is executed which restores the heap down for each and every node from one level less than the height of the tree.Finally,the value of the node which is replaced is returned.

The function del() receives two parameters-first is the base address of the array and second is the address of the variable that holds the total number of nodes present in the heap.The value that is to be deleted is always the root node,hence the value present at the index 1 is stored in a tempeory variable val,the value of maximum index is deceremented by one and then the heap is restored.Finally,the value that is deleted is returned.