Figure 9-8. *Array representation of a binary tree.*

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 0^{th} 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 represents 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 represented using arrays.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <windows.h>

struct node
```

```
{
    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 ( "In-order Traversal:\n" ) ;
    inorder ( root ) ;

    return 0 ;
}

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 -> 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
```

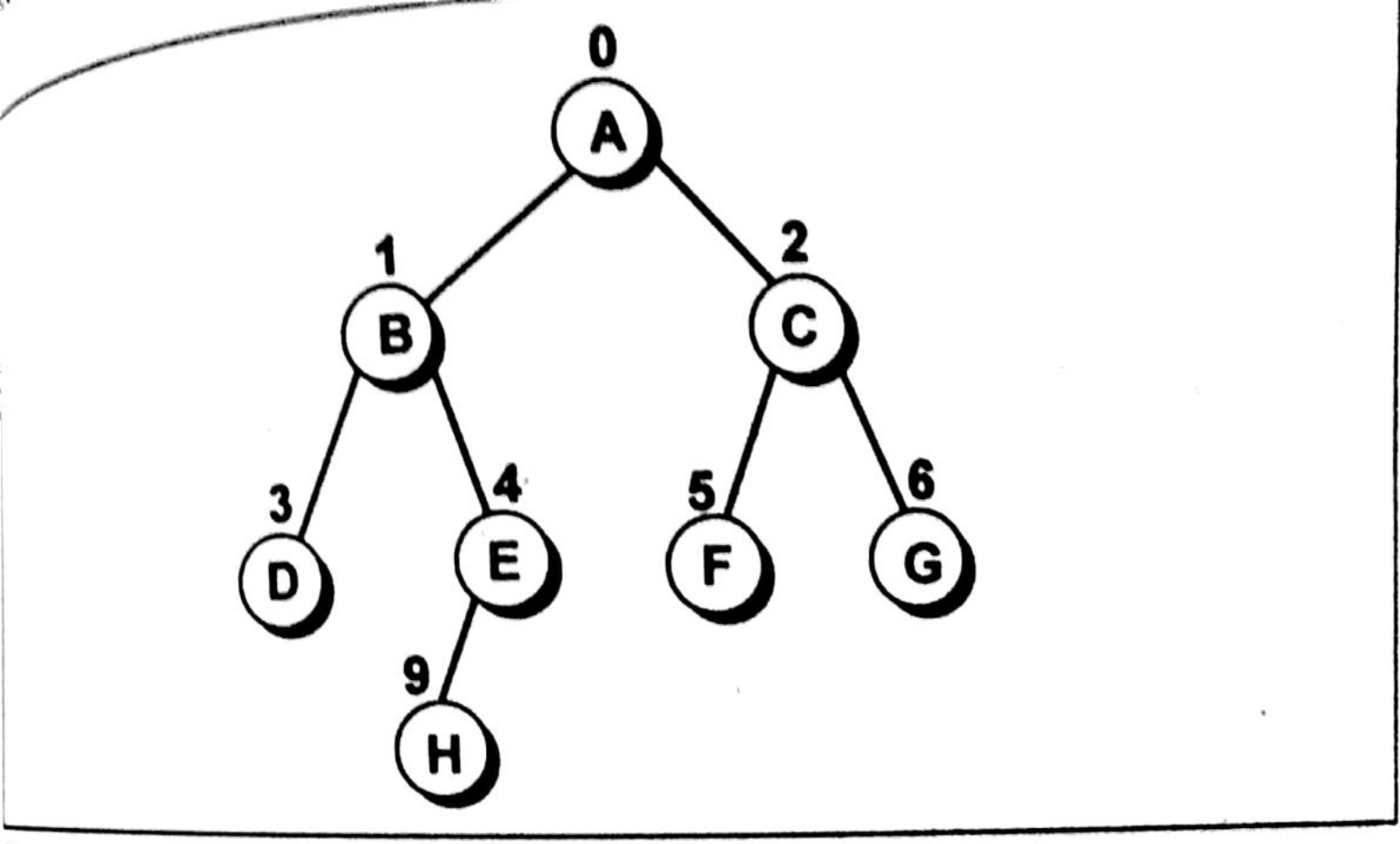

Figure 9-9(a). *Numbering of nodes.*

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

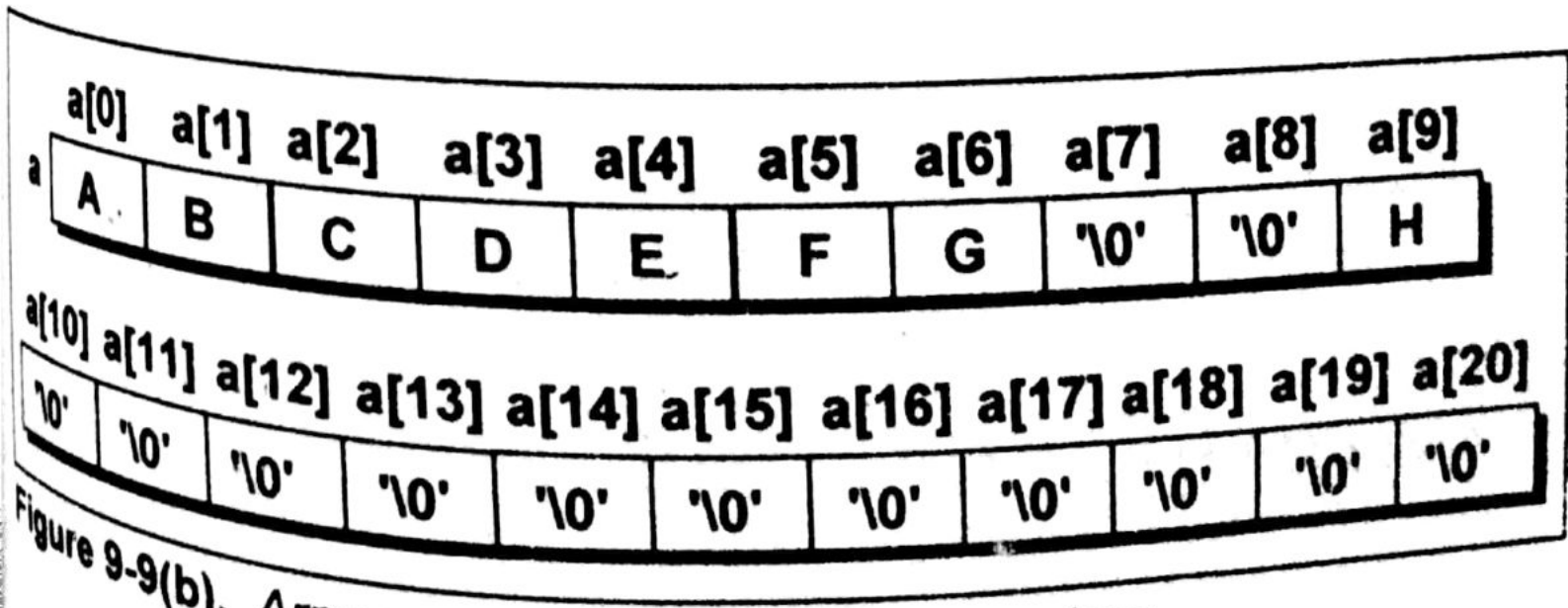

Figure 9-9(b). *Array representation of a Binary tree.*

Following program shows how a binary tree can be represented in an array using the method 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 * buildtree ( int ) ;
void inorder ( struct node * ) ;

char a[ ] = {
              '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
```

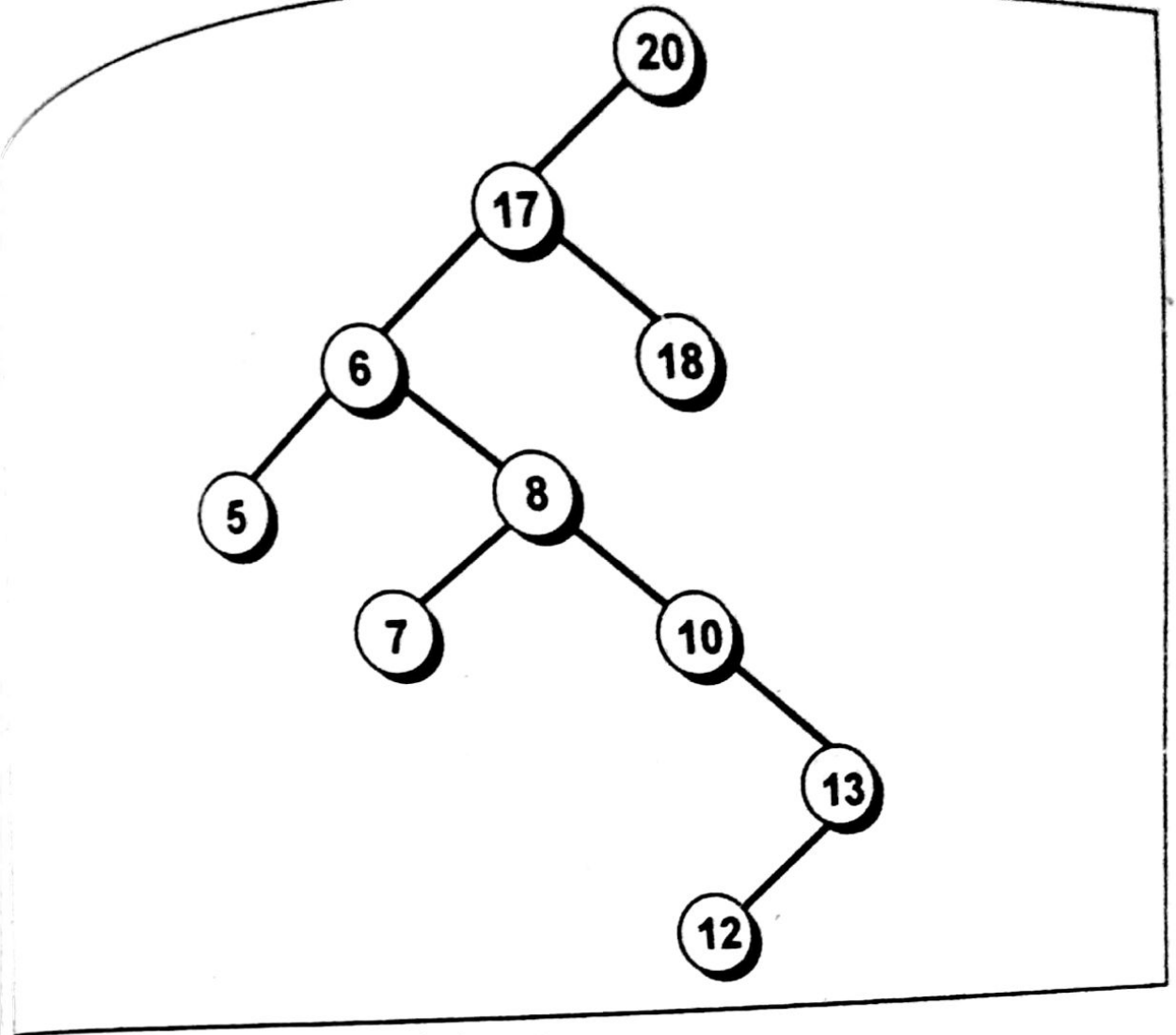

Figure 9-10. *Binary search tree.*

Such a binary 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-order Traversal:\n" ) ;
    postorder ( bt ) ;

    return 0 ;
}

/* 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 ;
        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 ) -> leftchild ), num ) ;
        else
            /* else traverse to right */
            insert ( &( ( *sr ) -> rightchild ), num ) ;
    }
    return ;
}

/* 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 traversed */
        printf ( "%d\t", sr -> data ) ;

        inorder ( sr -> rightchild ) ;
    }
    else
        return ;
}

/* traverse a binary search tree in a DLR (Data-Left-right) fashion */
void preorder ( struct btreenode *sr )
{
    if ( sr != NULL )
    {
        /* print the data of a node */
        printf ( "%d\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 (Left-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
```

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 TRUE 1
#define FALSE 0

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, a[ ] = { 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 ) ;

    del ( &bt, 8 ) ;
    printf ( "\n" ) ;
    printf ( "Binary tree after deletion:\n" ) ;
    inorder ( bt ) ;

    del ( &bt, 13 ) ;
    printf ( "\n" ) ;
    printf ( "Binary tree after deletion:\n" ) ;
    inorder ( bt ) ;

    return 0 ;
}

/* 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" ) ;
        return ;
    }

    parent = x = NULL ;

    /* call to search function to find the node to be deleted */
    search ( root, num, &parent, &x, &found ) ;

    /* if 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 = x ;
    xsucc = x -> rightchild ;

    while ( xsucc -> leftchild != NULL )
    {
        parent = xsucc ;
        xsucc = xsucc -> leftchild ;
    }

    x -> data = xsucc -> data ;
    x = xsucc ;
}

/* 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 -> leftchild == 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 && x -> rightchild == NULL )
    {
        if ( parent -> leftchild == x )
            parent -> leftchild = x -> leftchild ;
        else
            parent -> rightchild = x -> leftchild ;

        free ( x ) ;
        return ;
    }
}

/*returns 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
        btreenode **x, int *found )
{
    struct btreenode *q ;

    q = *root ;
    *found = FALSE ;
    *par = NULL ;

    while ( q != NULL )
    {
        /* if the node to be deleted is found */
        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 traversed */
      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    13    15
Binary tree after deletion:
7    9    11    12    15
```

nodes.

has (n + 1) external

Threaded Binary Tree

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

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

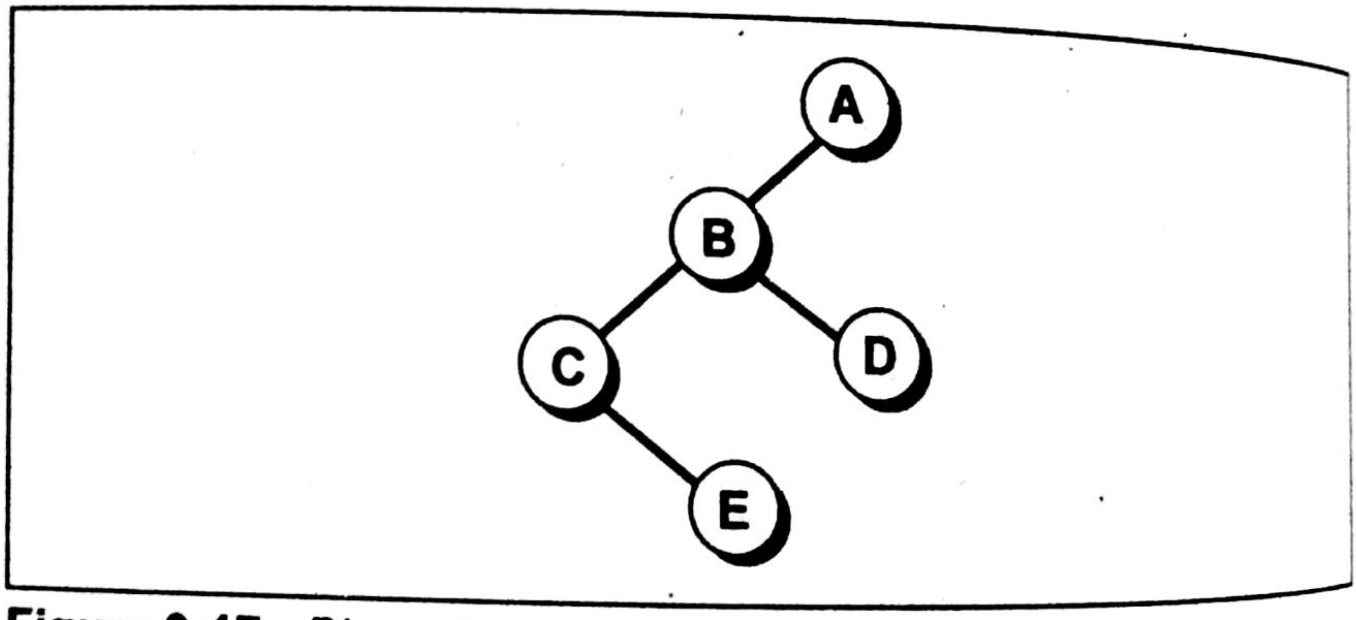

Figure 9-17. *Binary tree.*

Here, first we follow the left pointers until we reach node C, without, however, pushing the pointers to **A**, **B** and **C** onto a stack. For in-order traversal the data for node **C** 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-order 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 node **B**. We could then easily follow this pointer back to node **B**. This is shown in Figure 9-18.

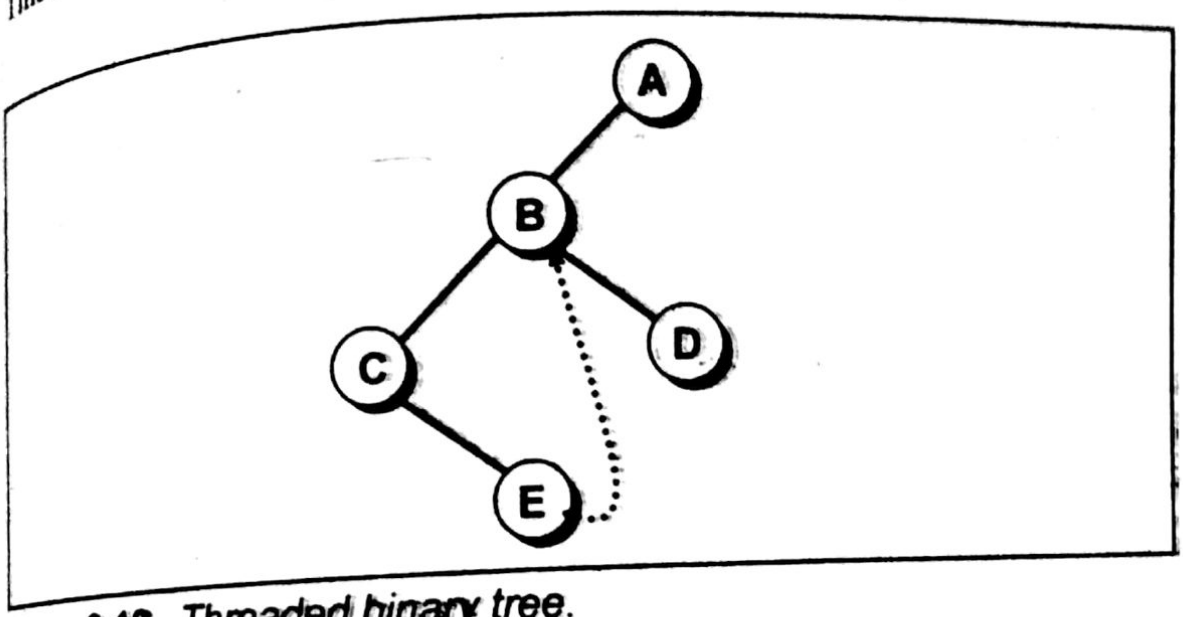

Figure 9-18. *Threaded binary tree.*

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.

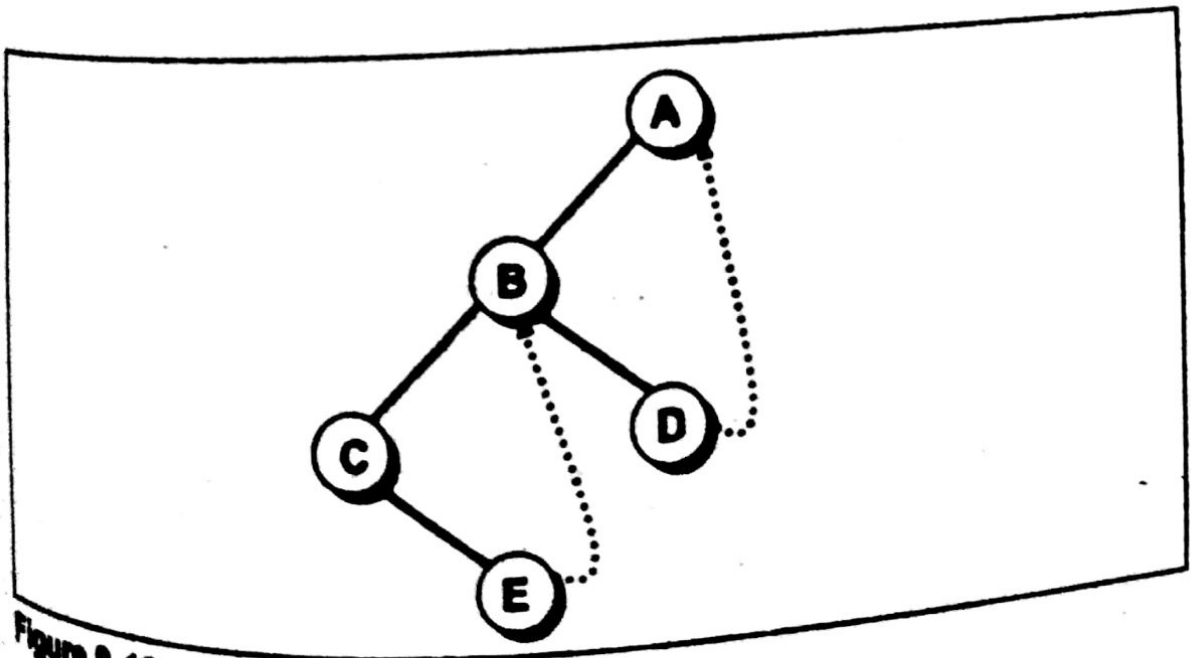

Figure 9-19. *Threaded binary tree.*

The program to implement a threaded binary tree is given below. The program shows how to insert nodes in a threaded 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>

enum boolean
{
    false = 0,
    true = 1
};

struct thtree
{
    enum boolean isleft ;
    struct thtree *left ;
    int data ;
    struct thtree *right ;
```

```
    enum boolen isright ;
} ;

void insert ( struct thtree **, int ) ;
void del ( struct thtree **, int ) ;
void search ( struct thtree **, int, struct thtree **,
                    struct thtree **, int * ) ;
void inorder ( struct thtree * ) ;
void deltree ( struct thtree ** ) ;

int main( )
{
    struct thtree *th_head ;

    th_head = NULL ; /* empty tree */

    insert ( &th_head, 11 ) ;
    insert ( &th_head, 9 ) ;
    insert ( &th_head, 13 ) ;
    insert ( &th_head, 8 ) ;
    insert ( &th_head, 10 ) ;
    insert ( &th_head, 12 ) ;
    insert ( &th_head, 14 ) ;
    insert ( &th_head, 15 ) ;
    insert ( &th_head, 7 ) ;

    system ( "cls" ) ;
    printf ( "Threaded binary tree before deletion:\n" ) ;
    inorder ( th_head ) ;

    del ( &th_head, 10 ) ;
    printf ( "\n" ) ;
    printf ( "Threaded binary tree after deletion:\n" ) ;
    inorder ( th_head ) ;

    del ( &th_head, 14 ) ;
    printf ( "\n" ) ;
    printf ( "Threaded binary tree after deletion:\n" ) ;
```

```
    inorder ( th_head ) ;

    del ( &th_head, 8 ) ;
    printf ( "\n" ) ;
    printf ( "Threaded binary tree after deletion:\n" ) ;
    inorder ( th_head ) ;

    del ( &th_head, 13 ) ;
    printf ( "\n" ) ;
    printf ( "Threaded binary tree after deletion:\n" ) ;
    inorder ( th_head ) ;

    deltree ( &th_head ) ;

    return 0 ;
}

/* inserts a node in a threaded binary tree */
void insert ( struct thtree **s, int num )
{
    struct thtree *p, *z, *head = *s ;

    /* allocating a new node */
    z = ( struct thtree * ) malloc ( sizeof ( struct thtree ) ) ;

    z -> isleft = true ;  /* indicates a thread */
    z -> data = num ;  /* assign new data */
    z -> isright = true ;  /* indicates a thread */

    /* if tree is empty */
    if ( *s == NULL )
    {
        head = ( struct thtree * ) malloc ( sizeof ( struct thtree ) ) ;

        /* the entire tree is treated as a left sub-tree of the head node */
        head -> isleft = false ;
        head -> left = z ;  /* z becomes leftchild of the head node */
        head -> data = -9999 ;  /* no data */
```

```
        head -> right = head ;  /* right link will always be pointing
                                   to itself */
        head -> isright = false ;

        *s = head ;

        z -> left = head ;  /* left thread to head */
        z -> right = head ;  /* right 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 -> isright = true ;
                    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 -> isleft = true ;
                        z -> left = p ;
                        return ;
                    }
                }
            }
        }
    }
}

/* deletes a node 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 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 -> 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 -> isleft = true ;
        parent -> left = x -> left ;
```

```
    }

    free ( x ) ;
    return ;
}

/* if the node to be deleted has only rightchild */
if ( x -> isleft == 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 -> left = x -> left ;
            x -> left -> right = parent ;
        }
        else
        {
            parent -> right = x -> left ;
            x -> left -> right = x -> right ;
        }

        free ( x ) ;
        return ;
    }
}

/* returns the address of the node to be deleted, address of its parent and
    whether the node is found or not */
void search ( struct thtree **root, int num, struct thtree **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 in 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:

Threaded binary tree before deletion:

8 9 10 11 12 13 14 15

Threaded binary tree after deletion:

8 9 11 12 13 14 15

Threaded binary tree after deletion:

8 9 11 12 13 15

Threaded binary tree after deletion:

9 11 12 13 15

Threaded binary tree after deletion:

9 11 12 15

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>
#include <windows.h>

#define MAX 101

struct node
{
    struct node *left ;
    int 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], pre[MAX], x ;

int main( )
{
    struct node *t ;
    int req, i, num ;

    t = NULL ; /* empty tree */

    system ( "cls" ) ;
    printf ( "Specify the number of items to be inserted: " ) ;
    while ( 1 )
    {
```

```
        scanf ( "%d", &req ) ;
        if ( req >= MAX || req <= 0 )
            printf ( "Enter number between 1 to 100.\n" ) ;
        else
            break ;
    }

    for ( i = 0 ; i < req ; i++ )
    {
        printf ( "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 = recons ( 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 ;
}

/* inserts 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 ) ;
        preorder ( t -> right ) ;
    }
}

void postorder ( struct node *t )
{
    if ( t != NULL )
    {
         postorder ( t -> left ) ;
         postorder ( t -> right ) ;
         printf ( "%d\t", t -> data ) ;
     }
 }

 void inorder ( struct node *t )
 {
     if ( t != NULL )
     {
          inorder ( t -> left ) ;
          printf ( "%d\t", in[x++]= t -> data ) ;
          inorder ( t -> right ) ;
      }
  }

  struct node * recons ( int *inorder, int *preorder, int noofnodes )
  {
       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++ )
        tempin[j] = inorder[j] ;

    for ( j = 0 ; j < i  ; j++ )
        temppre[j] = preorder[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] ;
        temppre[j - i] = preorder[j + 1] ;
    }
}

right = recons ( tempin, temppre, 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 Traversal:
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
```

AVL Trees

Searching in a binary search tree is efficient if the heights o
left and right sub-trees of any node are equal. However, fr
insertions and deletions in a BST is likely to make it unbal
The efficiency of searching is ideal if the difference betwe
heights of left and right sub-trees of all the nodes in a
search tree is at the most one. Such a binary search tree is ca
a **Balanced Binary Tree**. It was invented in the year 1962 l
Russian mathematicians—G. M. Adelson-Velskii and
Landis. Hence such trees are also known as AVL trees. Fi
27 shows some examples of AVL trees.

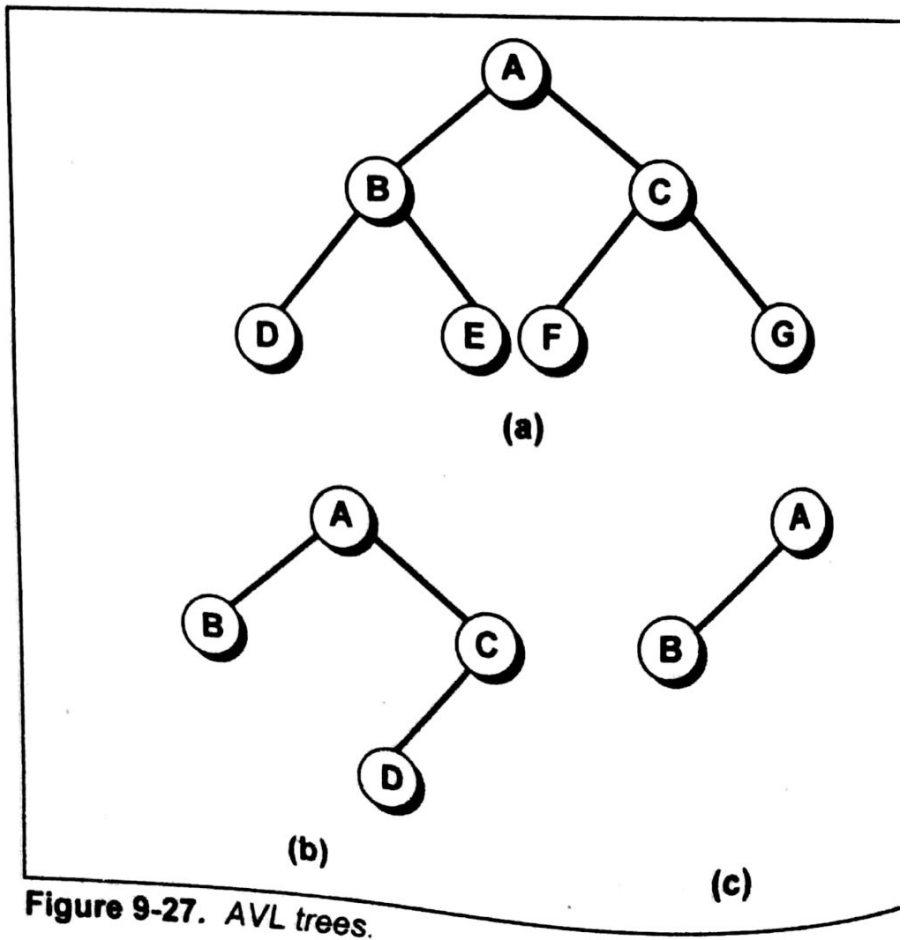

Figure 9-27. *AVL trees.*

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <windows.h>

#define FALSE 0
#define TRUE 1

struct AVLNode
{
    int data ;
    int balfact ;
    struct AVLNode *left ;
    struct AVLNode *right ;
};

struct AVLNode * buildtree ( struct AVLNode *, int, int * ) ;
struct AVLNode * deldata ( struct AVLNode *, int, int * ) ;
struct AVLNode * del ( struct AVLNode *, struct AVLNode *, int * ) ;
struct AVLNode * balright ( struct AVLNode *, int * ) ;
struct AVLNode * balleft ( struct AVLNode *, 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, 12, &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, 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 ) ;

    return 0 ;
}

/* inserts an element into tree */
struct AVLNode * buildtree ( struct AVLNode *root, int data, int *h )
{
    struct AVLNode *node1, *node2 ;
```

```
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:
                node1 = root -> left ;
                if ( node1 -> balfact == 1 )
                {
                    printf ( "Right rotation along %d.\n", root -> data ) ;
                    root -> left = node1 -> right ;
                    node1 -> right = root ;
                    root -> balfact = 0 ;
                    root = node1 ;
                }
                else
                {
                    printf ( "Double rotation, left along %d",
                                node1 -> data ) ;
                    node2 = node1 -> right ;
                    node1 -> right = node2 -> left ;
                    printf ( " then right along %d.\n", root -> data ) ;
                    node2 -> left = node1 ;
```

```
                    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 ;
                }
                root -> balfact = 0 ;
                *h = FALSE ;
                break ;

            case 0:
                root -> balfact = 1 ;
                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 )
    {
        switch ( root -> balfact )
        {
            case 1:
                root -> balfact = 0 ;
                *h = FALSE ;
```

```
        break ;

    case 0:
        root -> balfact = -1 ;
        break;

    case -1:
        node1 = root -> right ;
        if ( node1 -> balfact == -1 )
        {
            printf ( "Left rotation along %d.\n", 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 ;
            node1 -> left = node2 -> right ;
            node2 -> right = node1 ;
            printf ( " then left along %d.\n", 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 ;
        }
        root -> balfact = 0 ;
```

```
                    *h = FALSE ;
                }
            }
        }
        return ( root ) ;
}

/* 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 = balleft ( 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 = TRUE ;
        }
        return ( succ ) ;
    }

    /* balances the tree, if right sub-tree is higher */
    struct AVLNode * balright ( struct AVLNode *root, int *h )
    {
        struct AVLNode *node1, *node2 ;

        switch ( root -> balfact )
        {
            case 1:
                root -> balfact = 0 ;
                break ;

            case 0:
                root -> balfact = -1 ;
                *h  = FALSE ;
                break ;

            case -1:
                node1 = root -> right ;
                if ( node1 -> balfact <= 0 )
                {
                    printf ( "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.\n", 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 ;

    case 0:
        root -> balfact = 1 ;
        *h = FALSE ;
        break ;

    case 1:
        node1 = root -> left ;
        if ( node1 -> balfact >= 0 )
        {
            printf ( "Right rotation along %d.\n", 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 ) ;
}

/* displays 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 ) ;
}
```

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:

(a) 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.

(b) The level of all the leaf nodes must always be the same.

(c) One single node can contain either one or two values.

(d) If any node has two children (left and right) then that node contains 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.

(e) If any node has three children (left, middle and right), then that node contains two data values, 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**.

Figure 9-41 shows a 2-3 tree.

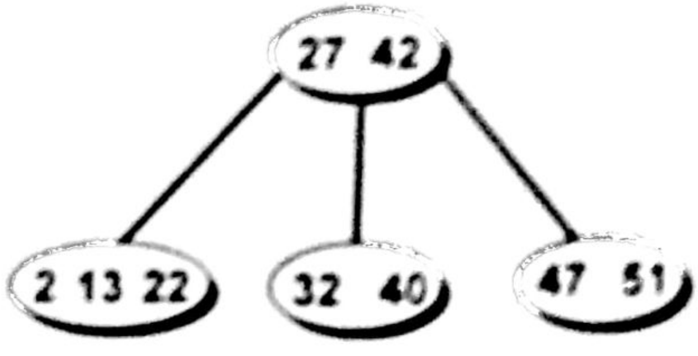

Figure 9-46. *Multi-way tree of order 4*

Definition Of B-Tree

B-tree is a multi-way search tree of order **n** that [illegible] following conditions:

(a) All the non-leaf nodes (except the root node) have at [illegible] children and at the most **n** children.

(b) The non-leaf root node may have at the most **n** [illegible] child and at least two child nodes.

(c) A B-tree can exist with only one node [illegible] containing no child.

(d) If a node has **n** children then it must have **n** - **1** [illegible] values of a particular node are in increasing order.

(e) All the values that appear on the left-most [illegible] smaller than the first value [illegible] [illegible]

Let us now put all the theory that we learnt into practice. Here is a program that implements B-tree of order 5.

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

#define MAX 4
#define MIN 2

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 ** ) ;
struct btnode * search ( int, struct btnode *, int * ) ;
int searchnode ( int, struct btnode *, int * ) ;
void fillnode ( int, struct btnode *, struct btnode *, int ) ;
void split ( int, struct btnode *, struct btnode *,
                    int, int *, struct btnode ** ) ;
struct btnode * del ( int, struct btnode * ) ;
int delhelp ( int, struct btnode * ) ;
void clear ( struct btnode *, int ) ;
void copysucc ( struct btnode *, int ) ;
void restore ( struct btnode *, int ) ;
void rightshift ( struct btnode *, int ) ;
void leftshift ( struct btnode *, int ) ;
void merge ( struct btnode *, int ) ;
void display ( struct btnode * ) ;

int main( )
```

```
{
    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 ) ;

    printf ( "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 ;
}

/* inserts 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 ;
    }
    return root ;
}

/* sets the value in the node */
int setval ( int val, struct btnode *n, int *p, struct btnode **c )
{
    int k ;
    if ( n == NULL )
    {
        *p = val ;
        *c = NULL ;
        return 1 ;
    }
    else
    {
        if ( searchnode ( val, n, &k ) )
            printf ( "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 ;
            }
        }
```

```
            return 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 )
    {
        if ( val < n -> value [1] )
        {
            *pos = 0 ;
            return 0 ;
        }
        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 -> child [i] ;
    }
    n -> value [k + 1] = val ;
    n -> child [k + 1] = c ;
    n -> count++ ;
}

/* 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 -> 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
            clear ( root, i ) ;
    }
    else
        flag = delhelp ( val, root -> child [i] ) ;

    if ( root -> child [i] != NULL )
    {
        if ( root -> child [i] -> count < MIN )
            restore ( root, i ) ;
    }
    return flag ;
  }
}

/* removes the value from the node and adjusts the values */
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 -> child [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[0] )
                temp = temp -> child [0] ;

        node -> value [i] = temp -> value [1] ;
}

/* adjusts the node */
void restore ( struct btnode *node, int i )
{
        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 -> child [i] ;
    }

    temp -> child [1] = temp -> child [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-- ;
}
```

```
* adjusts the values and children while shifting the value from parent to left
   child */
void leftsh ( struct btnode *node, int k )
{
    int i ;
    struct btnode *temp ;

    temp = node -> child [k - 1] ;
    temp -> count++ ;
    temp -> value [temp -> count] = node -> value [k] ;
    temp -> child [temp -> count] = node -> child [k] -> child [0] ;

    temp = node -> child [k] ;
    node -> value [k] = temp -> value [1] ;
    temp -> child [0] = temp -> child [1] ;
    temp -> count-- ;

    for ( i = 1 ; i <= temp -> count ; i++ )
    {
        temp -> value [i] = temp -> value [i + 1] ;
        temp -> child [i] = temp -> child [i + 1] ;
    }
}

/* merges two nodes */
void merge ( struct btnode *node, int k )
{
    int i ;
    struct btnode *temp1, *temp2 ;

    temp1 = node -> child [k] ;
    temp2 = node -> child [k - 1] ;
    temp2 -> count++ ;
    temp2 -> value [temp2 -> count] = node -> value [k] ;
    temp2 -> child [temp2 -> count] = node -> child [0] ;

    for ( i = 1 ; i <= temp1 -> count ; i++ )
    {
```

```
            temp2 -> count++ ;
            temp2 -> value [temp2 -> count] = temp1 -> value [i] ;
            temp2 -> child [temp2 -> count] = temp1 -> child [i] ;
        }
        for ( i = k ; i < node -> count ; i++ )
        {
            node -> value [i] = node -> value [i + 1] ;
            node -> child [i] = node -> child [i + 1] ;
        }
        node -> count-- ;
        free ( temp1 ) ;
}

/* displays the B-tree */
void display ( struct btnode *root )
{
    int i ;

    if ( root != NULL )
    {
        for ( 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 11 not found.

After deletion of values:
2    13    27    32    40    42    47    51
```

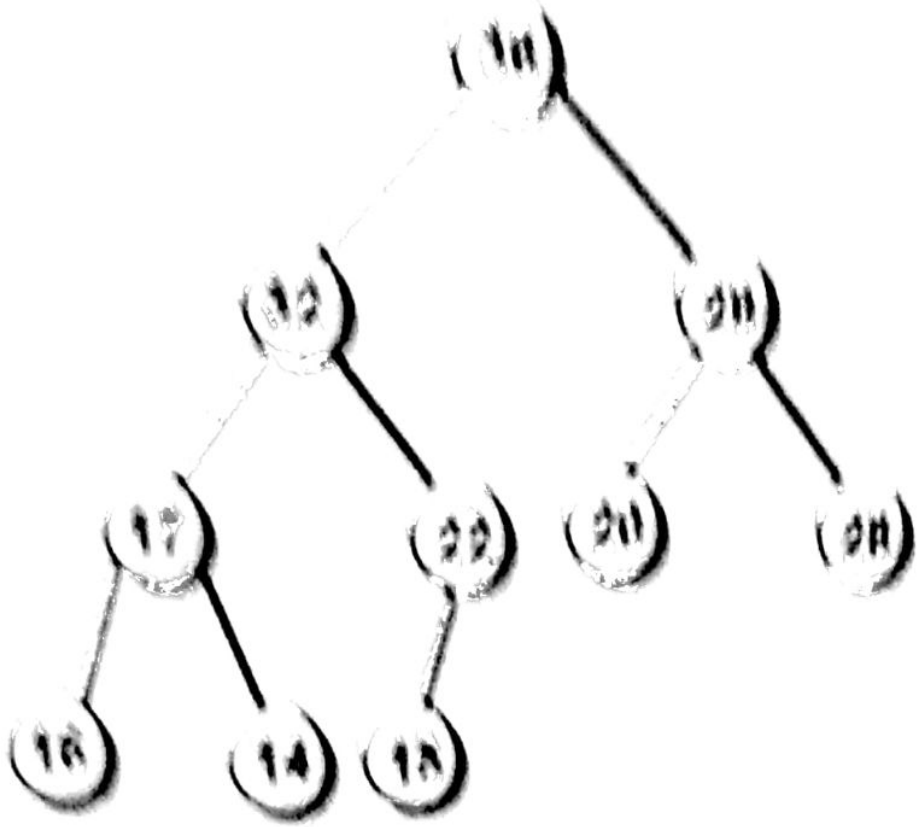

Figure 9-55. Max heap or (Maximum/Min) heap

Priority Queue Represented As A Heap

We have seen how a binary tree can be represented by one dimensional 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^{th} node its left and right child exist at $(2i + 1)^{th}$ and $(2i + 2)^{th}$ position respectively. Same is the case with heap, if the index of the root is considered as 1 then the left and right child of i^{th} node are present at $(2i)^{th}$ and $(2i + 1)^{th}$ position respectively and the parent node is present at $(i/2)^{th}$ index in the array. Figure 9-56 shows an array a that represents the heap shown in Figure 9-55.

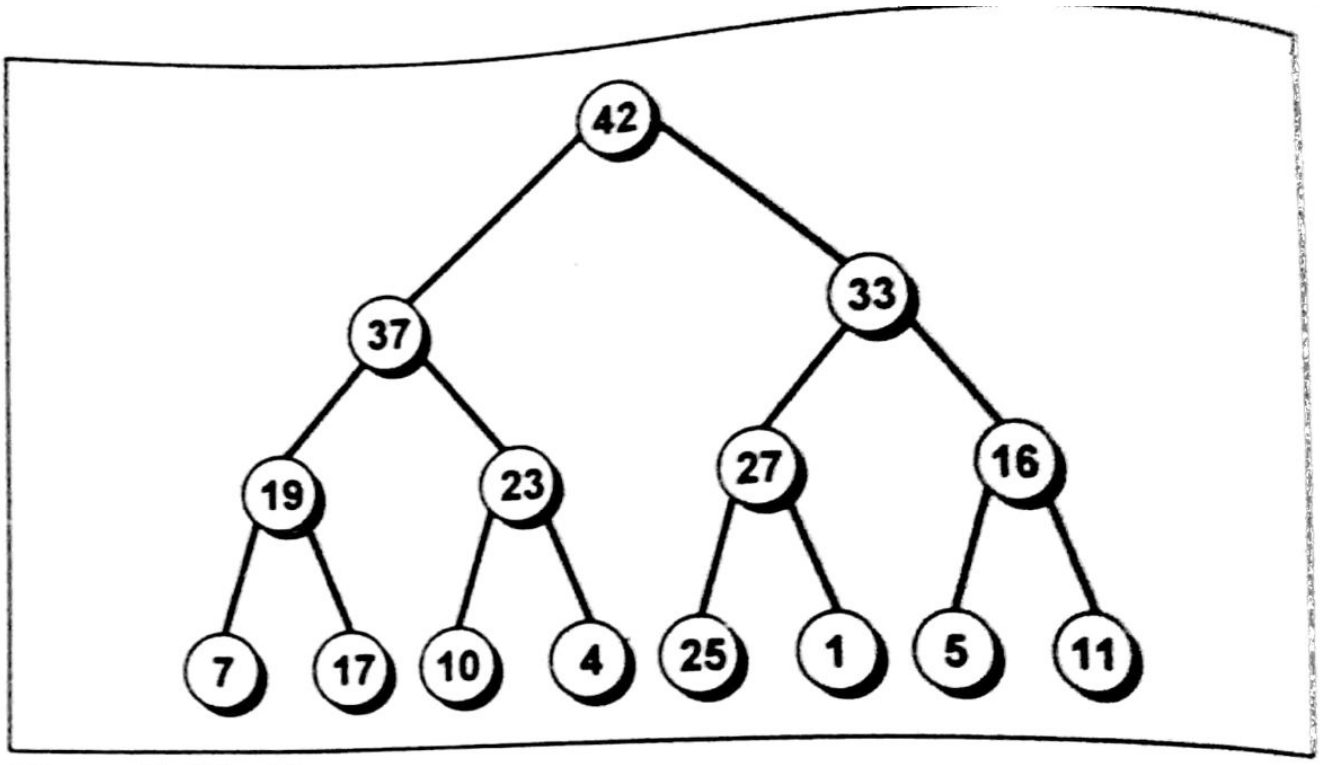

Figure 9-65. *Heap.*

Following program implements all the operations that can be performed on a heap.

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

void restoreup ( int, int * ) ;
void restoredown ( int, int *, int ) ;
void makeheap ( int *, int ) ;
void add ( int, int *, int * ) ;
int replace ( int, int *, int ) ;
int del ( int *, int * ) ;

int main( )
{
```

```
    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 ( "%d\t", arr [i] ) ;

    i = 24 ;
    add ( 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 ) ;
    printf ( "\n\n" ) ;
    printf ( "Element replaced %d.\n\n", i ) ;
    printf ( "Heap after replacement of an element:\n" ) ;
    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] ) ;

    return 0 ;
}

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, int n )
{
    int i ;
    for ( i = n / 2 ; i >= 1 ; i-- )
        restoredown ( i, arr, n ) ;
}

void add ( int val, int *arr, int *n )
{
    ( *n ) ++ ;
```

```
    arr [*n] = val ;
    restoreup ( *n, arr ) ;

}
int replace ( int i, int *arr, int n )
{
    int r = arr [1] ;
    arr [1] = i ;
    for ( i = n / 2 ; i >= 1 ; i-- )
        restoredown ( i, arr, n ) ;
    return 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:
```