

# Data Structures Using C++

## 1.1

## Introduction

Data structure is an implementation of an abstract data type having its own set of data elements along with functions to perform operations on that data. Arrays are considered as good example for implementing simple data structures. Arrays can be effectively used for random access of fixed amount of data. However, if the data structure requires frequent insertions and deletions then arrays are not much efficient because even a single insertion or deletion operation forces many other data elements to be moved to new locations. Also, there is always an upper limit to the number of elements that can be stored in the array-based data structure.

Dynamic data structures such as linked lists, which are based on dynamic memory management techniques, remove these limitations of static array-based data structures. They provide the flexibility in adding, deleting or rearranging data items at run time.

Data structures are categorized into two types: linear and nonlinear. Linear data structures are the ones in which elements are arranged in a sequence, such as arrays, linked lists, etc. Alternatively, nonlinear data structures are the ones in which elements are not arranged sequentially, such as trees and graphs.

There are certain linear data structures (e.g., stacks and queues) that permit the insertion and deletion operations only at the beginning or at end of the list, not in the middle. Such data structures have significant importance in systems processes such as compilation and program control.

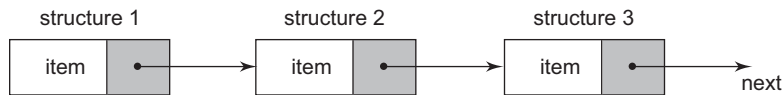
Data structures in C++ are implemented in the form of classes. The class members form the elements of the data structures while member functions are used to perform operations on the data elements.

## 1.2

## Linked lists

In contrast to arrays, a completely different way to represent a list is to make each item in the list part of a structure that also contains a “link” to the structure containing the next item, as shown in Fig. 1.1. This type of list is called a linked list because it is a list whose order is given by links from one item to the next.

## 2 Object Oriented Programming with C++



**Fig. 1.1**

Each structure of the list is called a node and consists of two fields, one containing the item, and the other containing the address of the next item (a pointer to the next item) in the list. A linked list is therefore a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented in Example 1.1:

### Example 1.1 Pointer to Structure Type

```
struct node
{
    int item;
    node *next;
};
```

Since linked list is an example of an ADT, we must define the associated functions that are required to manipulate the linked list structure. The typical operations performed on a linked list are

- **Insert:** Adding an element into the list
- **Delete:** Deleting an element from the list
- **Search:** Searching an element in the list
- **Print:** Displaying the list

Example 1.2 shows the implementation of a linked list along with its associated operations:

### Example 1.2 Linked List and Its Problems

```
class linkedlist
{
private:
    struct node //Declaring the structure of the linked list
                element
    {
        int element;
        node *next; //Linked list element pointing to another element
                    in the list
    }*first, *last; //Declaring the linked list pointers
```

```

public:
    node *n;
    linkedlist();
    void insert(int); //Inserting an element into the linked list
    int del(int);     //Deleting an element from the linked list
    void display(void); //Displaying the elements of the linked
                        list
    node *search (int); //Searching an element in the linked list
    ~linkedlist();
};

linkedlist:: linkedlist()
{
    first=NULL;
    last=NULL;
}

//Insert Function
void linkedlist:: insert(int value)
{
    node *ptr = new node; //Dynamically declaring a list
                           element
    ptr->element = value; //Assigning value to the newly
                           allocated list element

    //Updating linked list pointers
    if(first==NULL)
    {
        first = last = ptr;
        ptr->next=NULL;
    }
    else
    {
        last->next = ptr;
        ptr->next = NULL;
        last = ptr;
    }
}

//Delete function
int linkedlist:: del(int value)
{
    node *loc,*temp;
    int i;
    i=value;

    loc=search(i); //Searching the element to be deleted

    if(loc==NULL) //Element not found
        return(-9999);
}

```

#### 4 Object Oriented Programming with C++

```
        if(loc==first) //Element is found at first location in the list
        {
            if(first==last) //Found element is the only element in the list
            first=last=first->next;
            else
            first=first->next;
            return(value);
        }

        for(temp=first;temp->next!=loc;temp=temp->next)
            ;
        temp->next=loc->next; //Updating link pointers for element
        deletion
        if(loc==last)
            last=temp;
        return(loc->element);
    }

//Display function*/
void linkedlist:: display()
{
    node *ptr;

    if(first==NULL) //Checking whether the linked list is empty
    {
        cout <<"\n\tList is Empty!!";
        return;
    }

    //Printing the linked list elements
    cout<<"\nElements present in the Linked list are:\n";
    if(first==last) //Only one element present in the list
    {
        cout<<"\t"<<first->element;
        return;
    }

    for(ptr=first;ptr!=last;ptr=ptr->next) //Traversing the linked
                                            list
        cout<<"\t"<<ptr->element;
    cout<<"\t"<<last->element;
}

//Search function
linkedlist:: node* linkedlist:: search (int value)
{
    node *ptr;

    if(first==NULL) //Checking whether the list is empty
        return(NULL);
```

```

        if(first==last && first->element==value) //Condition where list
                                                    has only one element
            return(first);

        for(ptr=first;ptr!=last;ptr=ptr->next) //Traversing the linked
                                                    list
            if(ptr->element==value)
                return(ptr); //Returning the location where search element is
                found

        if(last->element==value)
            return(last);
        else
            return(NULL);

    }

    linkedlist:: ~linkedlist()
    {
        node *temp;
        if(first==last)
        {
            delete first;
            return;
        }
        while(first!=last)
        {
            temp=first->next;
            delete first;
            first=temp;
        }
        delete first;
    }

```

## 1.3

## Stacks

A stack is a linear data structure in which a data item is inserted and deleted at one end. A stack is called a Last In First Out (LIFO) structure because the data item that is inserted last into the stack is the first data item to be deleted from the stack. To understand this, imagine a pile of books. Usually, we pick up the topmost book from the pile. Similarly, the topmost item (i.e., the item which made the entry last) is the first one to be picked up/removed from the stack. Stacks are extensively used in computer applications. Their most notable use is in system software (such as compilers, operating systems, etc.).

In general, writing a value to the stack is called as a push operation, whereas reading a value from it is called as a pop operation. Example 1.3 shows the implementation of a stack along with its operations:

**Example 1.3 Implementation of Stack and Its Operations**

```

class stack
{
private:
    struct stk          //Declaring the structure for stack elements
    {
        int element;
        stk *next;      //Stack element pointing to another stack
                        element
    }*top;

public:
    stack();
    void push(int);      //Declaring a function prototype for
                        inserting an element into the stack
    int pop();           //Declaring a function prototype for removing
                        an element from the stack
    void display();      //Declaring a function prototype for
                        displaying the elements of a stack
    ~stack();
};

stack:: stack()
{
    top=NULL;
}

void stack:: push(int value)
{
    stk *ptr;
    ptr= new stk;        //Dynamically declaring a stack element

    ptr->element=value; //Assigning value to the newly allocated
                        stack element

    //Updating stack pointers
    ptr->next=top;
    top=ptr;
    return;
}

int stack:: pop()
{
    if(top==NULL)        //Checking whether the stack is empty
    {
        cout<<"\n\STACK is Empty.";
        getch();
    }
}

```

```

        exit(1);
    }
    else
    {
        int temp=top->element;
        top=top->next;
        return (temp);
    }
}

void stack:: display()
{
    stk *ptr1=NULL;
    ptr1=top;
    cout<<"\nThe various stack elements are:\n";
    while(ptr1!=NULL)
    {
        cout<<ptr1->element<<"\t";        //Printing stack elements
        ptr1=ptr1->next;
    }
}

stack::~ ~stack()
{
    stk *ptr1;
    while(top!=NULL)
    {
        ptr1=top->next;
        delete top;
        top=ptr1;
    }
}

```

## 1.4

## Queues

A queue is very similar to the way we (are supposed to) queue up at train reservation counters or at bank cashiers' desks. A queue is a linear, sequential list of items that are accessed in the order First In First Out (FIFO). That is, the first item inserted in a queue is also the first one to be accessed, the second item inserted in a queue is also the second one to be accessed, and the last one to be inserted is also the last one to be accessed. We cannot store/access the items in a queue arbitrarily or in any random fashion.

Similar to stacks, queues also have their application areas in systems processes. In fact, we may use queue data structure in all those situations where FIFO principle is to be implemented. Example 1.4 shows the implementation of a queue along with its operations:

**Example 1.4 Implementation of a Queue and Its Operations**

```

class queue
{
private:
    struct que //Declaring the structure for queue elements
    {
        int element;
        que *next; //Queue element pointing to another queue element
    }*front,*rear;

public:
    queue();
    void add(int); //Declaring a function prototype for inserting an
                    element into the queue
    int rem();     //Declaring a function prototype for removing an
                    element from the queue
    void display(void); //Declaring a function prototype for
                        displaying the elements of a queue
    ~queue();
};

queue:: queue()
{
    front=rear=NULL;
}

void queue:: add(int value)
{
    que *ptr = new que; //Dynamically declaring a queue element
    ptr->element = value; //Assigning value to the newly allocated
    queue element

    //Updating queue pointers
    if(front==NULL)
    {
        front = rear = ptr;
        ptr->next=NULL;
    }

    else
    {
        rear->next = ptr;
        ptr->next = NULL;
        rear = ptr;
    }
}

int queue:: rem()
{

```



```

    int i;

    if(front==NULL) //Checking whether the queue is empty
        return(-9999);

    else
    {
        i=front->element;
        front = front->next;
        return(i);
    }
}

void queue:: display()
{
    que *ptr=front;
    if(front==NULL)
    {
        cout<<"\n\tQueue is Empty!!";
        return;
    }

    else
    {
        cout<<"\nElements present in the Queue are:\n";
        //Printing queue elements
        while(ptr!=rear)
        {
            cout<<"\t"<<ptr->element;
            ptr=ptr->next;
        }
        cout<<"\t"<<rear->element;
    }
}

queue::~ ~queue()
{
    que *temp;
    if(front==rear)
    {
        delete front;
        return;
    }
    while(front!=rear)
    {
        temp=front->next;
        delete front;
        front=temp;
    }
    delete front;
}

```

## 1.5 Trees

A tree data structure arranges the data elements in a hierarchical tree format. The data elements of the tree are referred as nodes. A node is called parent node if it has child nodes descending from it. The node at the top of the hierarchy is called as root node.

A binary tree is a special form of a tree in which a parent node can have a maximum of two child nodes. A binary tree in sorted form holds great significance as it allows quick search, insertion and deletion operations. Figure 1.2 shows the general form of a binary tree:

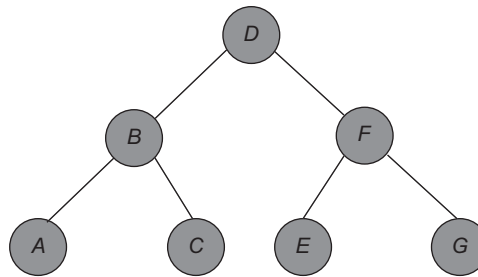


Fig. 1.2

There are three ways of traversing a binary tree:

- **Inorder:** In this tree traversal method, the left subtree is traversed first, followed by the root node and then the right subtree.
- **Preorder:** In this tree traversal method, the root node is traversed first, followed by the left subtree and then the right subtree.
- **Postorder:** In this tree traversal method, the root node is traversed first, followed by the right subtree and then the left subtree.

Example 1.5 shows the general form of representing a binary tree node:

### Example 1.5 Binary Tree Node

```

struct node
{
    int element; //Stores the data value of a node
    node *left; //Pointer to the left subtree
    node *right; //Pointer to the right subtree
};
  
```

We can embed the above data structure inside a class in C++ and build functions around it to implement different tree traversal techniques.

## 1.6

## Graphs

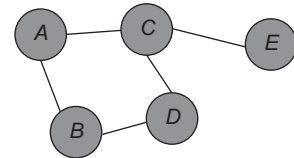
A graph is a set of nodes (called vertices) connected with each other by arcs (called edges). There is no specific pattern to how the nodes are arranged in a graph. A tree can also be considered as a specific instance of a graph.

Figure 1.3 shows a simple graph.

The set of vertices and edges in the above graph are:

$V = \{A, B, C, D, E\}$

$E = \{(A, C), (A, B), (C, D), (B, D), (C, E)\}$



**Fig. 1.3**

Graph data structure is used for representing and solving various problems in the field of computer science.

One of the common tasks associated with graphs is finding the shortest path between two nodes. A path constitutes the sequence of edges that need to be traversed to move from one node to another. Obviously, a graph must be connected for the possible existence of such a path.

To search a particular node in a graph, we may use the depth first search (DFS) or breadth first search (BFS) technique. The DFS technique performs the search operation across the depth of the graph while the BFS technique searches a node amongst the neighbouring nodes i.e., across the breadth of the graph.

A graph can be implemented with the help of adjacency list concept, which maintains a list of nodes adjacent to a particular node in the graph. Example 1.6 shows the general form of representing a graph node:

### Example 1.6 Graph Form in General Form

```

struct node
{
    int data; //Stores the data value of a node
    int STATUS; //Used as a flag to determine whether a node has
                //been visited or not
    node *next; //Pointer to the next node
    adj *link; //Adjacency link pointer
};

struct adj
{
    node *next;
    adj *link;
};
  
```

We can embed the above data structure inside a class in C++ and build functions around it to implement the different searching techniques in a graph.

