Data Structures through C++
Lab Manual

Prepared By:

K. Ravi Chythanya,
Assistant Professor,
Department of IT

Department of Information Technology

SREE CHAITANYA COLLEGE OF ENGINEERING

L.M.D Colony, Karimnagar-505001

1) LAB OBJECTIVE

Upon successful completion of this Lab the student will be able to:

1. Know about Object oriented programming.
2. Use Abstract Data Types in the programs.
3. Application of Non recursive functions.
4. OOP principles like Encapsulation Inheritance Polymorphism were frequently used.
5. Trees –B and AVL Trees and their operations were used.
6. Different sorting techniques (Quick sort, Merge sort, Heap sort)were used.
7. Hashing Techniques are implemented.

2) Guidelines to Students

- Equipment in the lab is meant for the use of student community. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care. Any damage is punishable.
- Students are required to carry their observation and records with completed exercises while entering the lab.
- Students are supposed to occupy the systems allotted to them and are not supposed to talk or make noise in the lab. The seating should be according to roll number wise.
- Lab can be used in free time/lunch hours by the students who need to use the systems should take prior permission from the lab in-charge.
- Students (mainly BOYS) are instructed to wear neat formal dress with in-shirt and identity cards should be hanged in their necks (including GIRLS).
- Lab records need to be submitted on or before date of submission.
- Students are not supposed to use pen drives in the lab.

Algorithm, Flowchart, Program Development

1) **Algorithm:** An Algorithm is a deterministic procedure that, when followed, yields a definite solution to a problem. An Algorithm is a design or plan of obtaining a solution to the problem. it is logically process of analyzing a mathematical problem and data step by step so as to make it easier to understand and implement solution to the problem .it is composed of a finite set of steps, each of which may require one or more operation. It may have zero or more inputs and produces one or major outputs. it should terminate after a finite number of operations.

Important features of an algorithm:

i. **Finiteness:** An algorithm terminates after a fixed number of steps.
ii. **Definiteness:** Each step of the algorithm is precisely defined.
iii. **Effectiveness:** All the operations used in the algorithm can be performed exactly in a fixed duration of time.
iv. **Input:** An algorithm has certain precise inputs before the execution of the algorithm begins.
v. **Output:** An algorithm has one or more outputs.

2) **Flowchart**

Flowchart is a graphical representation of an algorithm. It makes use of the basic operations in programming. All symbols are connected among themselves to indicate the flow of information and processing. A Flowchart is a diagrammatic representation of the various steps involved in the solution of a problem.

Flowchart is a symbolic diagram of operation sequence, data flow, control flow and processing logic in information processing. The symbols used are simple and easy to learn.

3) **Program Development:**

Step 1: vi example.cpp

```
#include<iostream>
using namespace std;
class example
        {

                public:
                        void ex()
                        {
                        cout<<"This is example program";
                        }
        };
void main()
{
example e;
e.ex();
}
```

Step 2: g++ example.cpp (hit Enter)

If the program has no errors, then type **./a.out** to see the output and the output is as follows.

Output: This is example program

Table of Contents

OOP Concepts:

The object oriented paradigm is built on the foundation laid by the structured programming concepts. The fundamental change in OOP is that a program is designed around the data being operated upon rather upon the operations themselves. Data and its functions are encapsulated into a single entity.OOP facilitates creating reusable code that can eventually save a lot of work. A feature called polymorphism permits to create multiple definitions for operators and functions. Another feature called inheritance permits to derive new classes from old ones. OOP introduces many new ideas and involves a different approach to programming than the procedural programming.

Benefits of Object-Oriented Programming:

- Data security is enforced.
- Inheritance saves time.
- User defined data types can be easily constructed.
- Inheritance emphasizes inventions of new data types.
- Large complexity in the software development can be easily managed.

Basic C++ Knowledge:

C++ began its life in Bell Labs, where Bjarne Stroustrup developed the language in the early 1980s. C++ is a powerful and flexible programming language. Thus, with minor exceptions, C++ is a superset of the C Programming language.

The principal enhancement being the object –oriented concept of a **class.** A Class is a user defined type that encapsulates many important mechanisms. Classes enable programmers to break an application up into small, manageable pieces, or **objects.**

Basic concepts of Object-oriented programming:

1) **Object:** Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.
2) **Class:** The entire set of data and code of an object can be made of a user defined data type with the help of a class in fact Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. A class is thus a collection of objects of similar type.
   For example: mango, apple, and orange are members of the class fruit.
   Ex: fruit mango; // will create an object mango belonging to the class fruit.
3) **Data Abstraction and Encapsulation:** The wrapping up of data and functions in to a single unit is known as **encapsulation**. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those functions which are wrapped in the class can access. This insulation of the data from direct access by the program is called data hiding. **Abstraction** refers to the act of representing essential features without including the background details or explanations. Since the classes use the concept of data abstraction, they are known as **Abstraction Data Type (ADT).**

4) **Inheritance:** Inheritance is the process by which objects of one class acquire the properties of objects of another class.

   For example: The bird 'robin ' is a part of the class 'flying bird' which is again a part of the class 'bird'. The concept of inheritance provides the idea of reusability.

5) **Polymorphism:** Polymorphism is another important OOP concept. Polymorphism means the ability to take more than one form. An operation may exhibit different instances. The process of making an operator to exhibit different behaviors in different instance is known as operator overloading.

6) **Dynamic Binding:** Binding refers to the linking of a function to the code to be executed in response to the call. Dynamic Binding (also known as late binding) means that the code associated with a given function which is to be used is not known until the time of the call at run-time. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

7) **Message Passing:** An object-oriented program consists of a set of objects that communicate with each other. This involves the following basic steps:
   1. Creating classes that define objects and their behavior.
   2. Creating objects from class definitions, and
   3. Establishing communication among objects.

   Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

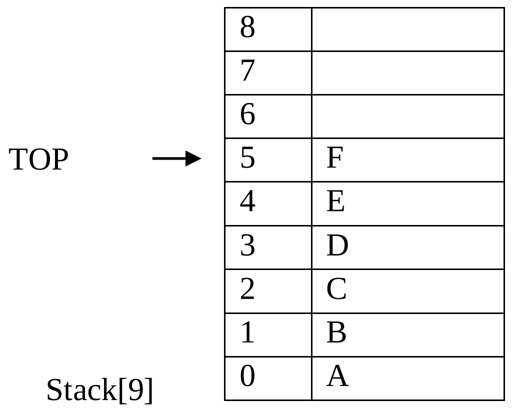
1) Write C++ programs to implement the following data structures using arrays.

a) Stack ADT b) Queue ADT

Aim: A program to implement the Stack ADT using arrays.

STACK: A stack is an ordered collection of data items into which new items may be inserted and from which data items may be deleted at one end. Stack are also called Last-In-First-out (LIFO) lists.

Representation of a Stack



Basic terminology associated with stacks:

1) **Stack Pointer (TOP):** Keeps track of the current position the stack.
2) **Overflow:** Occurs when we try to insert (push) more information on a stack than it can hold.
3) **Underflow:** Occurs when we try to delete (pop) an item off a stack, which is empty.

Basic Operation Associated with Stacks:

1) Insert (Push) an item into the stack.
2) Delete (Pop) an item from the stack.

a) Algorithm For Inserting an Item into the Stack S:

```
Procedure PUSH(S, SIZE, TOP, ITEM)
S     Array
SIZE Stack size
TOP   Stack Pointer
ITEM value in a cell
Step 1: {Check for stack overflow}
          If TOP==SIZE then
          Prints('Stack overflow')
          Return
Step 2: {Increment pointer top}
          TOP=TOP+1
Step 3: {Insert ITEM at top of the Stack}
          S[TOP]=ITEM
```

```
            Return
```

b) Algorithm for Deletion of an Item from the Stack S

```
Procedure POP(S, TOP)
S     Array
TOP   Stack Pointer
Step 1: {Check for stack underflow}
            If TOP==0 then
            Prints('Stack underflow')
            Return
Step 2: {Return former top element of stack}
            ITEM=(S[TOP]);
 Step 3: {Decrement pointer TOP}
            TOP=TOP-1
            Prints('Deleted item is:',item);
            Return
```

c) Algorithm to display the items of a Stack S

```
Procedure POP(S, TOP)
S     Array
TOP   Stack Pointer
Step 1: {Check for stack underflow}
            If TOP==0 then
            Prints('stack is empty')
            Return
Step 2: {display stack elements until TOP value}
            Prints(S[TOP])
            TOP=TOP+1
```

d) Algorithm to display top item of the Stack S

```
Procedure POP(S, TOP)
S     Array
TOP   Stack Pointer
Step 1: {Check for stack underflow}
            If TOP=0 then
            Prints('stack is empty')
            Return
Step 2: {display TOP value into the Stack}
            Prints(S[TOP])
```

Program:

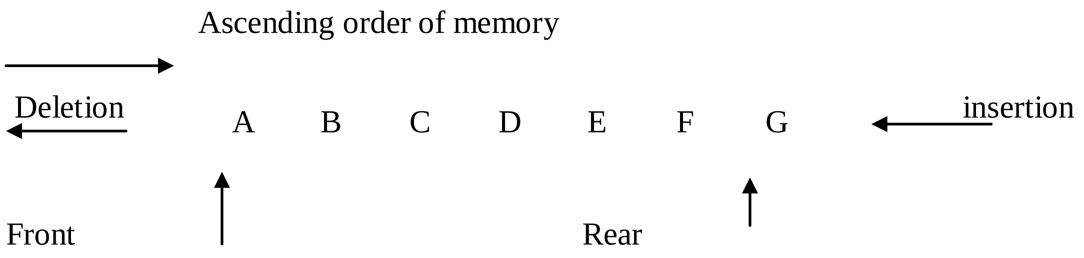
```
#include<iostream>
using namespace std;
#define MAX 10
int top=-1,ch,i;
template <class T>
class StackADT
{
public:
virtual void Push()=0;
virtual void Pop()=0;
virtual void Top()=0;
virtual void Display()=0;
};
template <class T>
class Stack: public StackADT<T>
{
T stk[MAX],ele;
public:
void Push();
void Pop();
void Top();
void Display();
};
template <class T>
void Stack<T>::Push()
{
if(top==(MAX-1))
cout<<"\nThe stack is full";
else
{
cout<<"\nEnter an element:";
cin>>ele;
top++;
stk[top]=ele;
cout<<"\nElement pushed successfully\n";
}
}
template <class T>
void Stack<T>::Pop()
```

```
{
if(top==-1)
cout<<"\nThe stack is empty";
else
{
ele=stk[top];
top--;
cout<<"The deleted element is:"<<ele;
}
}
template <class T>
void Stack<T>::Top()
{
if(top==-1)
cout<<"\nThe stack is empty";
else
cout<<"The top element of the stack is:"<<stk[top];
}
template<class T>
void Stack<T>::Display()
{
if(top==-1)
cout<<"\nThe stack is empty";
else
{
cout<<"\nThe elements in the stack are:";
for(i=top;i>=0;i--)
cout<<"\n"<<stk[i];
}
}
int main()
{
Stack<int> s1;
do
{
cout<<"\n****MENU****";
cout<<"\n1. Push\n2. Pop\n3. Top\n4. Display\n5. Exit";
cout<<"\nEnter ur Choice:";
cin>>ch;
switch(ch)
{
case 1: s1.Push();
        break;
case 2: s1.Pop();
        break;
case 3: s1.Top();
        break;
case 4: s1.Display();
        break;
case 5: exit(1);
default: cout<<"Enter correct Choice";
}
}while(true);
}
```

Aim: A C++ program to implement the Queue ADT using arrays.

QUEUE:

Queue is an ordered collection of data such that the data is inserted at one end and deleted from other end. It is a collection of items to be processed on a First-In-First-Out(FIFO) or First Come First Served(FCFS) basics.



Basic Operation Associated on Queues:

1) Insert an item into the Queue.
2) Delete an item into the Queue.

a) Algorithm to insert an item into a Queue Q:

```
Procudure Insert(Q, SIZE, F, R, ITEM)
Q     Array
SIZE Queue size
F front Pointer
R rear pointer
ITEM: information to be inserted at the rear of queue.
Step 1: {Check for Queue overflow}
        If  R>=SIZE then
        Prints('Queue overflow')
        Return
Step 2: {Increment rear pointer}
        R=R+1
Step 3: {Insert new element at rear end of queue}
        Q[R]=ITEM
Step 4: {If initially the queue is empty adjust the front pointer}
        If F=0, then F=1
```

b) Algorithm to delete an item from a Queue Q:

```
Procedure Delete(Q, F, R)
Q     Array
F front Pointer
R rear pointer
ITEM: information to be inserted at the rear of queue.
Step 1: {Check for Queue underflow}
        If F=0 then
        Prints('Queue underflow')
        Return
Step 2: {Delete the queue element at front end and store it into item}
        ITEM=Q[F]
 Step 3: {If queue is empty after deletion,set front and rear pointers to 0}
        If F=R then
        F=0
        R=0
        {Otherwise increment front pointer}
        Else
        F=F+1
        Return(ITEM)
```

c) Algorithm to display the items from the Queue Q

```
Procedure Dispaly(Q)
Q    Array
Step1: {Check queue values}
        If  F<0
                Prints('Queue is empty')
Step 2: {display Queue values}
        For I value F to R
        Prints(Q[I])
        I=I+1
```

Program:

```
#include<iostream>
using namespace std;
#define MAX 10
int front=0,rear=0,ch,i;
template <class T>
class QueueADT
{
public:
virtual void Insert()=0;
virtual void Delete()=0;
virtual void Display()=0;
};
template <class T>
class Queue: public QueueADT<T>
{
T q[MAX],ele;
public:
```

```
void Insert()
{
if(rear==MAX)
cout<<"\nQueue is full";
else
{
cout<<"\nEnter an element:";
cin>>ele;
q[rear]=ele;
rear++;
cout<<"\nElement inserted successfully\n";
}
}
void Delete()
{
if(front==rear)
cout<<"\nQueue is empty";
else
{
ele=q[front];
front++;
cout<<"The deleted element is:"<<ele;
}
}
void Display()
{
if(front==rear)
cout<<"\nQueue is empty";
else
{
cout<<"\nThe elements in the queue are:";
for(i=front;i<rear;i++)
cout<<q[i]<<" ";
}
}
};
int main()
{
Queue<int> q1;
do
```

```
{
cout<<"\n***MENU***";
cout<<"\n1. Insert\n2. Delete\n3. Display\n4. Exit";
cout<<"\nEnter ur Choice:";
cin>>ch;
switch(ch)
{
case 1: q1.Insert();
        break;
case 2: q1.Delete();
        break;
case 3: q1.Display();
        break;
case 4: exit(1);
default: cout<<"Entered Wrong Choice";
}
}while(1);
}
```

2) Write C++ programs to implement the following data structures using a singly linked list.
a) Stack ADT b) Queue ADT

Aim: A C++ program to implement the Stack ADT using singly linked list.

Program:

```
#include<iostream>
using namespace std;
template<class t>
class node
{
public:
t info;
node *link;
};
template <class t>
class StackADT
{
virtual void Push()=0;
virtual void Pop()=0;
virtual void Top()=0;
virtual void Display()=0;
};
template<class t>
class StackLinkImp:public StackADT<t>
{
public:
t item;
node<t> *temp, *top;
StackLinkImp()
{
top=NULL;
}
void Push()
{
temp=new node<t>;
cout<<"Enter the itemto be insrted on to the stack:";
cin>>item;
if(top==NULL)
temp->link=NULL;
else
temp->link=top;
temp->info=item;
top=temp;
cout<<"Insertion Completed Successfully";
}
void Pop()
{
if(top==NULL)
cout<<"Stack is empty";
else
{
item=top->info;
top=top->link;
cout<<"The deleted element is:"<<item;
}
}
void Top()
{
if(top==NULL)
cout<<"Stack is empty";
else
cout<<"The top element is:"<<top->info;
}
void Display()
{
if(top==NULL)
cout<<"Stack is empty";
else
{
temp=top;
cout<<"The elements in the stack are:";
while(temp!=NULL)
{
cout<<temp->info<<" ";
temp=temp->link;
}
}
}
};
int main()
```

```
{
int ch;
StackLinkImp<int> s1;
do
{
cout<<"\n****Menu****";
cout<<"\n1. Push\n2. Pop\n3. Top\n4. Display\n5. Exit";
cout<<"\nEnter ur choice:";
cin>>ch;
switch(ch)
{
case 1: s1.Push();
        break;
case 2: s1.Pop();
        break;
case 3: s1.Top();
        break;
case 4: s1.Display();
        break;
case 5: exit(1);
default: cout<<"Entered Wrong Choice";
}
}while(1);
}
```

Aim: A C++ program to implement the Queue ADT using singly linked list.

Program:

```
#include<iostream>
using namespace std;
template<class t>
class node
{
public:
t info;
node *link;
};
template<class t>
class QueueADT
{
virtual void Insert()=0;
virtual void Delete()=0;
virtual void Display()=0;
};
template <class t>
class QueueLinkImp: public QueueADT<t>
{
public:
t item;
node<t> *temp,*front,*rear;
QueueLinkImp()
{
front=rear=NULL;
}
void Insert()
{
temp=new node<t>;
cout<<"Enter an element to be inserted in to queue:";
cin>>item;
temp->info=item;
temp->link=NULL;
if(front==NULL)
{
front=rear=temp;
cout<<item<<" inserted Successfully";
return;
```

```
}
rear->link=temp;
rear=rear->link;
cout<<item<<" inserted Successfully";
}
void Delete()
{
if(front==NULL)
cout<<"Queue is empty";
else
{
item=front->info;
front=front->link;
cout<<"Deleted Element is: "<<item;
}
}
void Display()
{
if(front==NULL)
cout<<"Queue is empty";
else
{
cout<<"The elements in the queue are:";
for(temp=front;temp!=NULL;temp=temp->link)
cout<<temp->info<<" ";
}
}
};
void main()
{
int ch;
QueueLinkImp<int> q1;
do
{
cout<<"\n****MENU****";
cout<<"\n1. Insert\n2. Delete\n3. Display\n4. Exit";
cout<<"\nEnter ur choice:";
cin>>ch;
switch(ch)
{
```

```
case 1: q1.Insert();
        break;
case 2: q1.Delete();
        break;
case 3: q1.Display();
        break;
case 4: exit(1);
default: cout<<"Entered Wrong Choice";
}
}while(1);
}
```

3) Write C++ programs to implement the Double Ended Queue (DEQUE) using array.

Aim: A C++ program to implement the Double Ended Queue (DEQUE) using array.

Program:

```
#include <iostream>
using namespace std;
#define MAX 10
int front=-1,rear=-1,c,i;
template<class t>
class dqueADT
{
public:
virtual void addqatbeg()=0;
virtual void addqatend()=0;
virtual void delqatbeg()=0;
virtual void delqatend()=0;
virtual void display()=0;
};
template <class t>
class dque: public dqueADT<t>
{
public:
t q[MAX],item;
dque()
{
front=rear=-1;
for(i=0;i<MAX;i++)
q[i]=0;
}
void addqatbeg()
{
cout<<"Enter an element:";
cin>>item;
if (front==0&&rear==MAX-1)
{
cout<<"\nDeque is full"<<endl;
return ;
}
if(front==-1)
{
```

```
front=rear=0;
q[front]=item;
return;
}
if(rear!=MAX-1)
{
c=count();
int k=rear+1;
for(i=1;i<=c;i++)
{
q[k]=q[k-1];
k--;
}
q[k]=item;
front=k;
rear++;
}
else
{
front--;
q[front]=item;
}
}
void addqatend()
{
cout<<"Enter an element:";
cin>>item;
if(front==0&&rear==MAX-1)
{
cout<<"\nDeque is full"<<endl;
return;
}
if(front==-1)
{
rear=front=0;
q[rear]=item;
return;
}
if(rear==MAX-1)
{
```

```
int k=front-1;
for(i=front-1;i<rear;i++)
{
k=i;
if(k==MAX-1)
q[k]=0;
else
q[k]=q[i+1];
}
rear--;
front--;
}
rear++;
q[rear]=item;
}
void delqatbeg()
{
if(front==-1)
{
cout<<"\nDeque is empty"<<endl;
return;
}
item=q[front];
q[front]=0;
if(front==rear)
front=rear=-1;
else
front++;
cout<<"Deleted item is:"<<item;
}
void delqatend()
{
if(front==-1)
{
cout<<"\nDeque is empty"<<endl;
return;
}
item=q[rear];
q[rear]=0;
rear--;
```

```
if(rear==-1)
front=-1;
cout<<"Deleted item is: "<<item;
}
void display()
{
cout<<endl<<"front-> ";
for(i=0;i<MAX;i++)
cout<<"  "<<q[i];
cout<<" <-rear";
}
int count()
{
int c=0;
for(i=0;i<MAX;i++)
{
if(q[i]!=0)
c++;
}
return c;
}
};
main()
{
int ch;
dque<int> s1;
do
{
cout<<"\n****Menu****";
cout<<"\n1. Insert at Beginning\n2. Insert at End\n3. Delete from Beginning\n4. Delete from
End\n5. Display\n6. Exit";
cout<<"\nEnter ur choice:";
cin>>ch;
switch(ch)
{
case 1: s1.addqatbeg();
        break;
case 2: s1.addqatend();
        break;
case 3: s1.delqatbeg();
```

```
            break;
case 4: s1.delqatend();
            break;
case 5: s1.display();
            break;
case 6: exit(1);
}
}while(1);
}
```