DATA STRUCTURES

MODULE 1

STACK AND QUEUE

1.1 Introduction to different Data Structures

- A ***data structure*** is a method for organizing and storing data which would allow efficient data retrieval and usage.
- Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.
- Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

1.1.1 Efficiency of algorithms, complexity and big O notation

Algorithm : An algorithm is a clearly specified set of simple instructions to be followed to solve a problem. It is a finite set of instructions or logic, written in order to accomplish a certain predefined task.

1.1.1.1 Efficiency of algorithm

Once an algorithm is given for a problem and decided to be correct, the next important step is to determine how much in the way of resources, such as time or space, the algorithm will require.

An algorithm is said to be efficient and fast if it takes ***less time to execute*** and ***consumes less memory space***. The performance of an algorithm is measured on the basis of the following properties:

i. Space Complexity

ii. Time Complexity

1.1.1.2 Complexity

1) Space Complexity: The space complexity of a program is the amount of memory it needs to run to completion i.e, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available. An algorithm generally requires space for instructions, data and environment. The space needed by the programs can be summed up as

a. A fixed part that is independent of the characteristics (eg. number, size) of the inputs and outputs. This part typically includes the instruction space (space for code), space for simple variables (also called aggregate), space for constants, etc.

b. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables and the recursion stack space (depends on instance characteristics)

The space requirement S(P) of any program P may therefore be written as

$S(P) = c + S_p$ (instance characteristics)

where c is a constant.

Estimation of S_p (instance characteristics) is concentrated for any problem for which first, which instance characteristics to be used as a measure of the space requirements is determined.

2) **Time Complexity** : The time complexity of a program is the amount of computer time it needs to run to completion. The time T(P), taken by a program P is the sum of the compile time and the run (execution) time. The compile time does not depend on the instance characteristics and so, the run time of a program is concentrated upon. This run time is denoted by t_p (instance characteristics).

Estimation of t_p is done by knowing the characteristics of the compiler to be used and then proceeding to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores and so on, that would be made by the code for P.

Therefore, $t_p(n) = c_a \text{ADD}(n) + c_b \text{SUB}(n) + c_m \text{MUL}(n) + c_d \text{DIV}(n)$

where n denotes the instance characteristics

c_a, c_b, c_m, c_d denote the time required for an addition, subtraction, multiplication, division etc.

ADD, SUB, MUL, DIV etc., are functions whose value is the number of additions, subtractions, multiplications, divisions, etc., that will be performed when the code for P is used as an instance with characteristic n.

Note: Obtaining an exact formula is an impossible task since the time needed for an addition, subtraction, multiplication, division, etc., often depends on the actual numbers being used for the above operations.

Instead of determining the exact number of operations (+, -, *, / ......) that are needed to solve a problem instance with characteristics given by n, all the operations are lumped together and a count is obtained for the total number of operations i.e, count only the number of program steps. A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

The number of steps of any program statement is to be assigned depends on the ***nature of the statement.*** The various statement types are :

- Comments – Non-executable statements which has a step count of 0.
- Declarative statements – Defines variables and constants, own data types, access specifiers and function definitions, which has a step count of 0.
- Expressions and assignment statements – Most expressions have a step count of 1, except for expressions that contain function calls, where the cost of invoking the functions has to be determined.
- Iteration statements – This class of statements includes the for, while and do statements where the step counts are considered for the control part i.e, the expressions and the initialization statements.
- Switch statement – This statement consists of a header followed by one or more sets of condition – statement pairs and the step counts are considered only for the control statements i.e, expression and conditions.
- If..else statement – This consists of three parts ie.,

  if (< expr >) <statements 1>';

  else <statements 2>;

  Here, each part is assigned the number of steps corresponding to < expr >, <statements 1> and <statements 2> respectively.
- Function invocation – All invocations of functions count as one step, except for the invocation which involves pass-by-value parameters where the size depends on the instance characteristics. If it is a recursive function, then the local variables in the function being invoked are also considered.
- Memory management statements – These include new object, delete object and sizeof(object). The step count associated with each is 1. [For constructor and destructor invocation, the step count is computed as for function invocation.]
- Function statements – These count as 0 steps.
- Jump statements – These include continue, break, goto, return and return <expr>. Each has a step count of 1 except for return <expr> where the cost of <expr> is included.

➢ The best-case step count is the minimum number of steps that can be executed for the given parameters.

➢ The worst-case step count is the maximum number of steps that can be executed for the given parameters.

- The average-case step count is the average number of steps executed on instances with the given parameters.

Example:- (first count = 0)

```
float sum(float *a, const int n)
{
        float s = 0;                    //count++
        for(int i = 0; i< n; i++)
        {                               //count++
                s += a[ i];             // count++
        }                               // n times
        return s;                       // count++
}
```

Therefore, Time T(P) = 1 + 2n + 2 = 2n + 3

1.1.1.3 Big-O notation

The analysis required to estimate the resource use of an algorithm is generally a theoretical issue and therefore, a formal framework is required.

As discussed before about the determination of the step counts, the main motivation behind it is the ability to compare the time complexities of two or more programs that compute the same function and also predict the growth in run time as the instance characteristics change.

Determining the exact step count is a difficult task and is not worthwhile since the notion of a step is itself inexact. So, the exact step count is not very useful for comparative purposes. So, for a comparative study, some terminologies have been introduced that enable to make meaningful (but inexact) statements about the time and space complexities of a program. The idea of the following definitions is to establish a relative order among functions.

(1) Big "Oh" notation

Definition: *f(n) = O(g(n)) [read as "f of n is big oh of g of n"] iff [if and only if] there exists positive constants c and n_0 such that f(n) ≤ cg(n) for all n, n ≥ 0.*

[Note: f and g are non-negative functions]

Although 1000n is larger than n^2 for small values of n, n^2 grows at a faster rate and thus n^2 will eventually be the larger function. The turning point is in the case of n = 1000.

The definition says that eventually there is some point n_0 past which c.g(n) is always at least as large as f(n), so that if constant factors are ignored, g(n) is at least as big as f(n). In the above case, we have

f(n) = 1000n, g'(n) = n^2, n_0 = 1000 & c = 1 [can also use n_0 = 10 & c = 100]

Thus, we can say that 1000n = O(n^2). This notation is known as Big-Oh notation.

Examples:

3n + 2 = O(n) as 3n + 2 ≤ 4n for all n ≥ 2
3n + 3 = O(n) as 3n + 3 ≤ 4n for all n ≥ 3

$10\ n^2 + 4n + 2 = O(n^2)$ as $10\ n^2 + 4n + 2 \leq 11n^2$ for $n \geq 5$
$6*2^n + n^2 = O(2^n)$ as $6*2^n + n^2 \leq 7*2^n$ for $n \geq 4$

O(1) denotes a computing time that is a constant.
O(n) is called linear.
$O(n^2)$ is called quadratic.
$O(n^3)$ is called cubic.
$O(2^n)$ is called exponential.
O(log n) is called logarithmic.

If an algorithm takes time O(log n), it is faster for sufficiently large n, than it had taken O(n). Similarly, O(n log n) is better than $O(n^2)$ but not as good as O(n).

(2) "Omega" notation

Definition: *f(n) = Ω(g(n)) iff there exist positive constants c and n_0 such that f(n) ≥ c g(n) for all*

$n \geq n_0$.

(3) "Theta" notation

Definition: *f(n) = θ(g(n)) iff there exist positive constants c_1, c_2 and n_0 such that c_1 g(n) ≤ f(n) ≥ c_2 g(n) for all n, $n \geq n_0$.*

(4) Small "oh" notation

Definition: *f(n) = o(g(n)) if for all constants c, there exists an n_0 such that f(n) < c g(n) when*

$n > n_0$. Less formally, f(n) = o(g(n)) if f(n) = O(g(n)) and f(n) = O(g(n)) and f(n) ≠

θ(g(n)).

1.1.2 Different types of data structures – linear and non-linear

Definition of Data Structure : A data structure is a method for organizing and storing data which would allow efficient data retrieval and usage.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address. Thus, the record and array data structures are based on computing the addresses of data items with arithmetic operations; while the linked data structures are based on storing addresses of data items within the structure itself. May data structures use both principles, sometimes combined.

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their and space and time cost). Common data structures include array, linked list, hash table, graph, heap, Tree (binary tree, B-tree, red-black tree, avail tree, trie), stack and queue.

Data structures is broadly divided into two as:

- Linear data structure
- Non-linear data structure

Linear data structure

It is a structure that organizes its data elements one after the other (sequentially). It is organized in a way similar to how the computer's memory is organized.

- Organizes their data elements in a linear fashion where data elements are attached one after the other
- Traversed one after the other (data elements), and only one element can be directly reached.
- They are very easy to implement since computer's memory is also organized in a linear fashion.
- Some commonly used linear data structures are:
    i. Array : It is a collection of similar data elements stored in consecutive memory location where each element could be identified using an index (subscript). RDBMS uses arrays.
    ii. Linked List : It is a sequence of nodes, where each node is made up of a data element and a reference (pointer) to the next node in the sequence.
    iii. Stack : It is actually a list where data elements can only be added or removed from the top of the list.
    iv. Queue : It is also a list, where data elements can be added from one end of the list and removed from the other end of the list.

Non-linear data structure

It is a structure constructed by attaching a data element to several other data elements in such a way that it reflects a specific relationship among them. They are organized in a different way than the computer's memory ie., the elements are not organized in a sequential order. It branches to more than one node and cannot be traversed in a single run.

- Here, data elements are not organized in a sequential fashion.
- A data item could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run.
- It is difficult to implement in computer's linear memory.
- Some commonly used non-linear data structures are:
    i. Tree : It is made up of a set linked nodes which can be used to represent a hierarchical relationship among data elements. In hierarchical data model, trees are used as the data structure.
    ii. Graph : It is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that store the data elements. In the network data model, graphs are used as the data structure.

1.1.3 Basic data structure operations – insertion, deletion, search and traverse

The main operations that can be performed on the data structures are:

i. Insertion :- It means inserting a value at a specified position in data structure.

ii. Deletion :- It means deleting a particular value from a specified position in a data structure.

iii. Searching :- It means searching a particular data in a created data structure.

iv. Traversing :- It means reading and processing (visiting) each and every element of a data structure at least once.

1.1.4 Abstract Data Types (ADTs) and C++ classes

An abstract data type (ADT) is a set of objects together with a set of operations. They are mathematical abstractions i.e, how the set of operations is implemented is not mentioned anywhere in the ADT's definition.

Objects such as lists, sets and graphs, along with their operations, can be viewed as abstract data types, such as integers, reals and booleans are data types. For the set ADT, the operations are add, remove, size and contains.

The C++ class allows for the implantation of ADTs, with appropriate hiding of implementation details. Thus, any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate method. If for some reason implementation details need to be changed, it should be easy to do so by merely changing the routines that perform the ADT operations. This change would be completely transparent to the rest of the program.

According o the design decision, there is no rule for which operations must be supported for each other. Error handling and tie breaking are also generally up to the program designer.

Example : Abstract data type: *NaturalNumber* :- This contains the class definition of Natural number. It is assumed that the Boolean type has already been defined elsewhere. *[Natural Number is an ordered subrange of integers starting at 0 and ending as the maximum integer (MAXINT) on the computer.]*

```
class NaturalNumber {
 public:
        NaturalNumber Zero( );                //returns 0
        Boolean IsZero( );             //if *this is 0, return TRUE, otherwise returns FALSE
        NaturalNumber Add(NaturalNumber y);       //return the smaller of *this+y and
MAXINT.
        Boolean Equal(NaturalNumber y);   //return TRUE if *this = = y; otherwise return
FALSE.
        NaturalNumber Successor( ); //if *this is MAXINT return MAXINT; otherwise return
                                    //*this+1.
        NaturalNumber Substact(NaturalNumber y); //if *this < y return 0; otherwise return
*this-y.
};
```

This ADT contains the class definition of NaturalNumber. Here, it uses C++ class to define an ADT. Here, some operators in C++ like operator <<, when overloaded for user-defined ADTs do not exist as member functions of the corresponding class. Rather these operators exist as ordinary C++ functions. Thus, these operations are declared outside the C++ class definition of the ADT even though they are actually part of the ADT.

1.1.5 Use of iterators

An iterator is an object that is used to traverse all the elements of a container class. [A container class is a class that represents a data structure that contains or stores a number of data objects. Objects can usually be added to or deleted from a container class. Ex) Array].

Need for iterators :

Some of the operations that has to be performed on a container class C may include

1. Print all integers in C.
2. Obtain max, min, mean, median or mode of all integers in C.
3. Obtain the sum, product, or sum of squares of integers in C etc.

To solve these problems, it is required to traverse all elements of the container class. So, inorder to traverse all the elements of the container class, iterators are used.

C++ STL : At the core of the C++ **S**tandard **T**emplate **L**ibrary, there are the three following well-structured components.

1. **Containers** – They are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map et.
2. **Algorithms** – They act on containers. They provide the means to perform initialization, sorting, searching and transforming of the contents of containers.
3. **Iterators** – They are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

Iteration is of two types:

i. Index based iteration

ii. Iterator based iteration

i. Index based iteration

This is the normal type of iteration used in C-style code. It is used in the loops.

Example)

```
for ( i = 0; i != size; i++) {
// access elements of v[ i ]
// any code including continue, break, return
}
```

This is only used for sequential random access containers (vector, array, deque). It does not work for list, forward-list or the associative containers.

ii. Iterator based iteration

To know this, it is necessary to know about vector and iterators.

Vector : It is a container, similar to an array with an exception that it automatically handles its own storage requirements in case it grows. “Vector” is a template class that is a perfect replacement for the good old C-style arrays. It allows the same natural syntax that is used with plain arrays that offers a series of services that free the C++ programmer from taking care of the allocated memory and help operating consistently on the contained objects.

Iterators : An iterator is any object that, pointing to some element in a range of elements (such as an array or a container). It has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++)).

Example for iterator based iteration:

```
for ( auto it = v.begin( ); it != v.end( ); ++it) {
// if the current index is needed:
auto i = std :: distance(v.begin( ); it);
// access element is *it
// any code including continue, break, return
}
```

Advantage : It is more generic and works for all containers.

Disadvantage : It needs extra work to get the index of the current element.

The following program shows both the index based iteration and iterator based iteration

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
        // create a vector to store int
        vector<int> vec;
        int i;
        // display the original size of vec
        cout<<"vector size="<<vec.size()<<endl;
        for(i = 0; i < 5; i++)
        {
                vec.push_back(i);
        }
        // display extended size of vec
        cout<<"extended vector size="<<vec.size()<<endl;
        // access 5 values from the vector
        for( i = 0; i < 5; i++)
        {
                cout<<"value of vec ["<<i<<"]="<<vec[i]<<endl;
        }
        // use of iterator to access the values
        vector< int >::iterator v=vec.begin();
        cout<<"v front="<<vec.front()<<endl;
        while(v!=vec.end()){
                cout<<"value of v="<<*v<<endl;
                cout<<"v="<<&v<<endl;
                v++;
        }
return 0;
}
```

1.1.6 Array as an ADT with printArray( ) operation

Although an array is usually implemented as a consecutive set of memory locations, this is not always the case. An array is considered as a set of pairs, <index, value>, such that each index that is defined has a value associated with it. In mathematical terms, this is called as a correspondence or a mapping. When considering array as an ADT, importance is given to the operations that can be performed on an array. The main operations are:

i) Creating a new array

ii) Retrieving a value from the array

iii) Storing a value into the array

iv) Printing the array values

A class definition for an array ADT can be written as follows:

```
class GeneralArray {

// objects: A set of pairs <index, value> where for each value of index in IndexSet, there is a
// value of type float. IndexSet is a finite ordered set of one or more dimensions. For ex,
// [0,1.............,n-1] for one dimension. (0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)
// for two dimensions, etc.
public:
   GeneralArray(int j, RangeList list, float initValue = defaultValue);
    // The constructor General Array creates a j dimensional array of floats; the range of the
   // kth dimension is given by the kth element of list. For each index i in the index set, insert
   // <i, initValue> into the array.

   float Retrieve(index, i);
   // if (i is in the index set of the array) return the float associated with i in the array;
else
   //signal an error.

   void Store(index i, float x);
   // if (i is in the index set of the array) delete any pair of the form <i, y> present in the array
  // and insert the new pair <i, x>; else signal an error.

   void printArray( );
   // if (array is not empty) print the value of each index in the IndexSet.
}; // end of GeneralArray
```

ADT 1.1: Abstract Data Type *GeneralArray*

Constructor *GeneralArray*(**int** *j, RangeList list*, **float** *initValue* = defaultValue) : This produces a new array of the appropriate size and type. All the items are initially set to the floating point variable *initValue*.

Retrieve(*index, i*) : It accepts an index and returns the value associated with the index if the index is valid or an error if the index is invalid.
Store(*index i, float x*) : It accepts an index and a value of type float and replaces the *<index, oldvalue>* pair with the *<index, newvalue>* pair.

printArray( ) : It prints the values in the array ie., for each valid *index* (from the *RangeList*) in the *IndexSet*, the value of that index *<index, value>* is displayed.

The advantage of this ADT definition is that it clearly points out the fact that the array is a more general structure than a "consecutive set of memory locations".

GeneralArray is more general than the C++ array as it is more flexible about the composition of the index set.

Disadvantages of C++ array :

a) The C++ array requires the index set to be a set of consecutive integers starting at 0.

b) C++ does not check an array index to ensure that it belongs to the range for which the array is defined.

1.2 Understanding Stack and its operations

1.2.1 Stack and its operations – Push and Pop

Definition of Stack : Stack is a linear data structure which contains an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only i.e, it uses the principle of Last In First Out (LIFO). Examples that uses this phenomenon

i) Shunting of trains in a rail yard

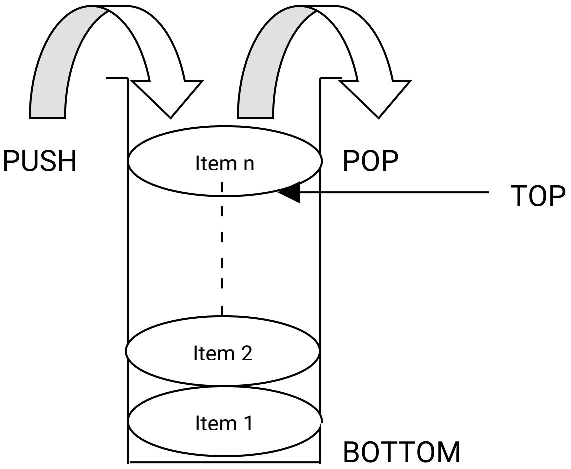
ii) Shipment in a cargo

iii) Order supply in a restaurant

In case of array and linked list, insertions and deletions can be done at any position, but in stack, it can be done only at one end i.e, at the top of the stack.

Basic operations in a stack

The insertion and deletion operations of stack are specially termed as PUSH and POP respectively and the position of the stack where these operations are performed is known as TOP of the stack. PEEP operation is used to show the topmost (data) element in a stack without deleting it. An element in a stack is termed as ITEM. The maximum number of elements that a stack can accommodate is termed as SIZE. The following figure shows a typical view of stack data structure.



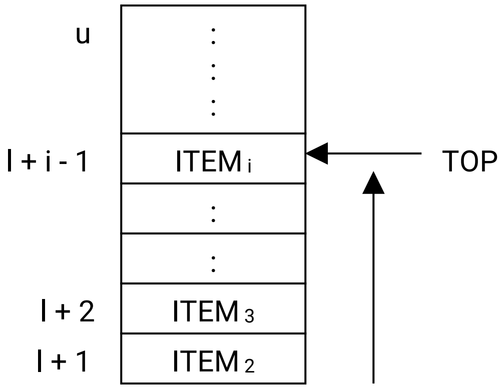
1.2.2 Array representation of stacks

A stack may be represented in the memory in various ways. Mainly there are two ways:

(1) Using one-dimensional array

(2) Using Singly Linked List

In the array representation of stacks, firstly, a memory block of sufficient size is allocated to accommodate full capacity of the stack. Then, starting from the first location of the memory block, items of the stack can be stored in sequential fashion. The following figure shows the array representation of stack.



l	ITEM $_1$	Botto

ITEM $_i$ denotes the ith item in the stack

l and u denote the index range of array in use, usually values of these are 0 and SIZE-1 respectively

TOP is the pointer to point the position of array up to which it is filled with the items of stack.

With this representation, the following two status can be stated.

EMPTY : TOP < l

FULL : TOP >= u

1.2.3 Stack ADT with push(), pop(), stackfull() and stackempty()

The easiest way to implement a stack ADT is by using a one-dimensional array, say *stack*[*MaxSize*], where *MaxSize* is the maximum number of entries. The first, or bottom element of the stack is stored in *stack*[0], the second in *stack*[1] and the *i*th in *stack*[*i*-1]. Associated with the array is a variable *top*, which points to the top element in the stack. Initially, *top* is set to -1 to denote an empty stack. This results in the following stackADT specification with the data member declarations and constructor definition of Stack:

```
template <class KeyType>
class Stack
{
 // objects : A finite ordered list with zero or more elements.
public:
 Stack (int MaxStackSize = DefaultSize);
 // Create an empty stack whose maximum size is MaxStackSize

 Boolean IsStackFull( );
 // if number of elements in the stack is equal to the maximum size of the stack, return
TRUE (1);
 // otherwise, return FALSE (0).

 Boolean IsStackEmpty( );
 //  if number of elements in the stack is 0, return TRUE (1) else return FALSE (0).

 void Push(const KeyType& item);
 // if IsStackFull( ), then StackFull( ); else insert item into the top of the stack.

 KeyType* Pop( );
 // if IsStackEmpty( ), then StackEmpty( ) and return 0; else remove and return a pointer to
the top
 // element of the stack.

KeyType* Peep( );
 // if IsStackEmpty( ), then StackEmpty( ) and return 0; else return the topmost element in
the stack.
};
```

ADT 1.2: Abstract Data Type *Stack*

Data member declarations are:

```
private:
   int top;
   KeyType *stack;
   int MaxSize;
```

The constructor definition of *Stack* is:

```
template <class KeyType>
Stack<KeyType>::Stack (int MaxStackSize) : MaxSize (MaxStackSize)
{
  stack = new KeyType[MaxSize];
  top = -1;
}
```

The member functions *IsStackFull*( ) and *IsStackEmpty*( ) are implemented as follows:

```
template <class KeyType>
inline Boolean Stack <KeyType>::IsStackFull( )
{
 if (top = = MaxSize-1) return TRUE;
 else return FALSE;
}

template <class KeyType>
inline Boolean Stack <KeyType>::IsStackEmpty( )
{
 if (top = = -1) return TRUE;
 else return FALSE;
}
```

The *Push*, *Pop* and *Peep* operations use these functions as shown below:

```
template <class KeyType>
void Stack <KeyType>:: Push(const KeyType& item);
{
 if (IsStackFull( )) StackFull( );
 else stack[++top] = item;
}

template <class KeyType>
KeyType* Stack <KeyType>:: Pop( );
{
 if (IsStackEmpty( )) { StackEmpty( ); return 0; }
 item = stack[top--];
 return &item;
}
```

```
template <class KeyType>
KeyType* Stack <KeyType>:: Peep( );
{
  if (IsStackEmpty( )) { StackEmpty( ); return 0; }
  item = stack[top];
  return &item;
}
```

The value returned by *Pop* and *Peep* is of type *KeyType**, rather than *KeyType* or *KeyType&* to handle the case when the stack is empty. In this case, 0 is returned. *StackFull*( ) and *StackEmpty*( ) are functions that depend on a particular application. Often when a stack becomes full, the *StackFull*() function will signal that more storage needs to be allocated and the program is rerun. *StackEmpty*( ) is often a meaningful condition in the context of the application for which the stack is being used.

1.2.4 Infix, Prefix and Postfix expressions

An expression is made up of operands, operators and delimiters. The following expression

X = A / B – C + D * E – A * C

has five operands: A, B, C, D and E. Though these are all one-letter variables, operands can be any legal variable name or constant in our programming language. In any expression, the values that variables take must be consistent with the operations performed on them. These operations are described by the operators. The basic arithmetic operators are plus, minus, times and divide (+, -, *, /). Other arithmetic operators include unary minus and %. A second class is the relational operators: <. <=, >, >=. = = and < >.

In order to convert such an expression to instruction understandable by the computer, one has to first determine the exact order in which the operators are evaluated. Each operator is given a priority and the expression is evaluated using these priorities. In this evaluation, a stack can be used to store the different operators and operands.

The expressions can be written in three types of notation: Infix, Postfix and Prefix.

i) Infix notation : The conventional method of writing an expression is called Infix.
Example for Infix notation: A+B, C-D, E*F, G/H
Here, the notation is ***<operand> <operator> <operand>*** i.e, the operator is between the operands.

ii) Postfix notation: This uses the convention of writing the operators after the operands.

Example of Postfix notation : AB+, CD-, EF*, GH/
Here, the notation is ***<operand> <operand> <operator>.*** This is also known as reverse Polish notation.

iii) Prefix notation : This uses the convention of writing the operators before the operands.

Example of Prefix notation: +AB, -CD, *EF, /GH
Here, the notation is ***<operator> <operand> <operand>***. This is also known as Polish notation.

Note: In all the above notations, a unary operator precedes its operand.

For the evaluation of an arithmetic expression, the following two steps are followed.

(1) Convert the conventional infix to either postfix or prefix notation.

(2) Evaluate the postfix or prefix notations respectively.

(1) Conversion of infix to

(i) Postfix

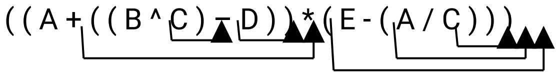
Step 1: Assume the fully parenthesized version of the infix expression.

Step 2: Move all operators so that they replace their corresponding right part of parenthesis.

Step 3: Remove all parenthesis.

Example:1. A fully parenthesized expression ((A+((B^C)-D))*(E-(A/C)))

2.



3. ABC^D-+EAC/-*

(ii) Prefix

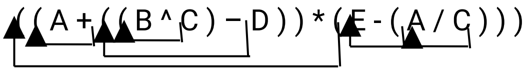
Step 1: Assume the fully parenthesized version of the infix expression.

Step 2: Move all operators so that they replace their corresponding left part of parenthesis.

Step 3: Remove all parenthesis

Example:1. A fully parenthesized expression ((A+((B^C)-D))*(E-(A/C)))

2.



3. *+A-^BCD-E/AC

(2) Evaluate

- This is the main advantage.
- In evaluation in postfix notation, scanning from left to right is required exactly once. [This is possible using stack].

Advantage of postfix or prefix over infix:

i) Need for parenthesis is eliminated.

ii) Priority of operators is eliminated.

iii) Evaluation process is much simpler than attempting a direct evaluation from infix notation.

1.2.5 Infix to Postfix conversion using Stack ADT

General Algorithm for conversion of infix expression to postfix expression

(1) Append a symbol ')' at the end of a given infix expression.

(2) Initialise the stack with '('.

(3) Read one input symbol at a time and decide whether it has to be pushed into the stack. This decision will be governed by the following table.

Symbol	**In-stack priority value** **ISP**	**In-coming priority value** **ICP**
+ -	2	1
* /	4	3
^	5	6
operand	8	7
(	0	9
)	-	0

(4) From the table, two priority values are decided i.e, in-stack priority (ISP) and in-coming priority (ICP).

   a. A symbol will be pushed into the stack if its ICP value is greater than the ISP value of the topmost element.

      i.e, if (ICP(item) > ISP(x)) then PUSH(x) and PUSH(item).

   b. Similarly, a symbol will be popped from the stack if its ISP value is greater than or equal to the ICP value of the incoming element.

      i.e, if (ISP(x) >= ICP(item)) then POP(x).

Assumptions for the algorithm:

READ_SYMBOL( ) : From a given infix expression, this function will read the next symbol.
ISP(x) : Returns the in-stack priority value for a symbol 'x'.
OUTPUT(x) : Append the symbol 'x' into the resultant expression.
SIZE : Capacity of the stack.
TOP : Current pointer pointing to top of the stack.
PUSH(x) : Push operation in stack.
POP( ) : Pop operation in stack

Algorithm INFIX_TO_POSTFIX(E)

Input : E, simple arithmetic expression in infix notation delimited at end by the right parenthesis ‘)’, in-coming and in-stack priority values for all possible symbols in an arithmetic expression.
Output : An arithmetic expression in postfix notation.
Data Structure : Array representation of stack ADT say Stack with TOP as the pointer to the top-most element.
Steps:

```
(1) Stack<char> Stack                 // Create a stack using stack ADT
(2) Stack.PUSH(‘(’)                   // Initialize the stack
(3) While (Stack not empty) do
        1. item = E.READ_SYMBOL( )    // Scan the next symbol in infix expression
        2. x = Stack.POP( )           // Get the next item from the stack
        3. Case : item = operand      // If the symbol is an operand
              i. Stack.PUSH(x)        // The stack will remain the same
             ii. OUTPUT(item)         // Add the symbol into the output expression
        4. Case : item = ‘)’          // Scan reaches to its end
              i. While (x ≠ ‘(’ ) do
                    a) OUTPUT(x)
                    b) x = Stack.POP()
             ii. EndWhile
        5. Case : ISP(x) >= ICP(item)
              i. While (ISP(x) >= ICP(item)) do
                    a) OUTPUT(x)
                    b) x = Stack.POP( )
             ii. EndWhile
            iii. Stack.PUSH(x)
             iv. Stack.PUSH(item)
        6. Case : ISP(x) < ICP(item)
              i. Stack.PUSH(x)
             ii. Stack.PUSH(item)
        7. Otherwise : Print “Invalid Expression”
(4) EndWhile
```

(5) Stop

Example : Let us verify the above procedure INFIX_TO_POSTFIX with the following arithmetic expression:
Input : (A + B) ^ C - (D * E) / F)
Symbol reading : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Read Symbol	Stack	Output
Initial	(	
1	((	
2	((	A
3	((+	A
4	((+	AB
5	(	AB+
6	(^	AB+
7	(^	AB+C
8	(-	AB+C^
9	(-(	AB+C^
10	(-(	AB+C^D
11	(-(*	AB+C^D
12	(-(*	AB+C^DE
13	(-	AB+C^DE*
14	(-/	AB+C^DE*
15	(-/	AB+C^DE*F
16		AB+C^DE*F/-

1.2.6 Evaluation of Postfix expression using Stack ADT

For a given expression in postfix notation, it can be easily evaluated. The following algorithm EVAL_POSTFIX is to evaluate an arithmetic expression in postfix notation using a stack.

Algorithm : EVAL_POSTFIX(E)

Input : E, an expression in postfix notation, with the value of the operands appearing in the expression.
Output : Value of the expression.
Data Structure : Stack ADT using array with TOP as the pointer to the top-most element.

Steps :

```
(1) stack<char> Stack                    // Create a stack using stack ADT
(2) Append a special symbol '#' at the end of the expression
(3) Item = E.READ_SYMBOL( )              // Read the first symbol from E (postfix)
(4) While (item ≠ '#') do
        i.   If (item = operand) then
                 a) Stack.PUSH(item)     // Operand is first pushed into the stack
        ii.  Else
                 a) op = item            // item is an operator
                 b) y = Stack.POP( )     // right-most operand of current operator
                 c) x = Stack.POP( )     // left-most operand of current operator
                 d) t = x op y           // perform the operation with operator 'op' and
                                         //operands x and y
                 e) Stack.PUSH(t)        // Push the result into stack
        iii. EndIf
        iv.  Item = E.READ_SYMBOL( )     // Read the next item from E
(5) EndWhile
(6) value = Stack.POP( )
(7) Return (value)
(8) Stop
```

Example :

Infix : (A+((B*C)/D))
Postfix : ABC*D/+
Input : ABC*D/+ # with A = 2, B = 3, C = 4 and D = 6

Read Symbol	Stack	
A	2	PUSH(A = 2)
B	2 3	PUSH (B = 3)
C	2 3 4	PUSH(C = 4)
*	2 12	POP(4), POP(3), PUSH(T = 12)
D	2 12 6	PUSH(D = 6)
/	2 2	POP(6), POP(12), PUSH(T = 2)

+	4	POP(2), POP(2), PUSH(T = 4)
#		value = POP( ) which is 4

1.3 Understanding Queues and its operations

Definition of queue :

A queue is an ordered collection of homogeneous data elements where insertion and deletion operations take place at two extreme ends i.e, it uses the principle of First In First Out (FIFO). It is a linear data structure like array, stack and linked list where ordering of the elements are in linear fashion.

Here, the queue insertion (called ***enqueue***) operation takes place at the ***"rear"*** end and deletion (called ***dequeue***) operation takes place at the ***"front"*** end. The following figure represents a model of queue structure.

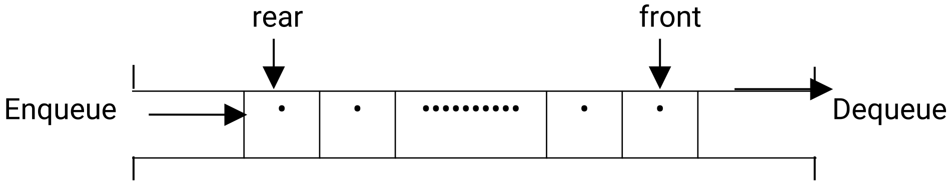
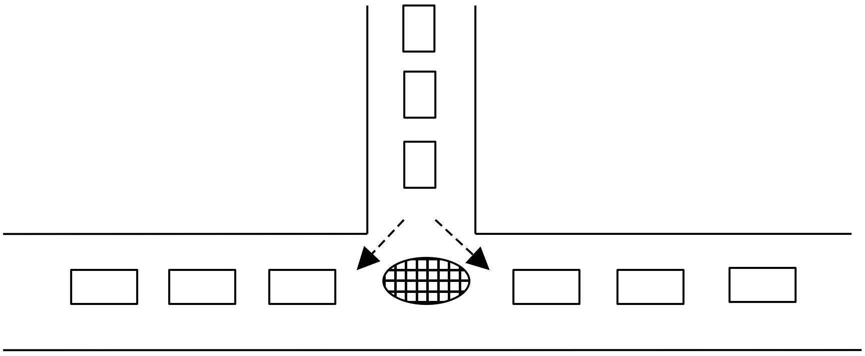


Fig : Model of a Queue

Elements in a queue are termed as ***item***; the number of elements that a queue can accommodate is termed as ***MaxSize***.

Examples:

1. Queuing in front of a counter. Here, the customer who comes first is served first.
2. Traffic control at a turning point. All the traffics will wait on a line till it gets the signal for moving. And on getting the "Go" signal, the vehicles will turn on a first come first turn basis.



3. Job scheduling: Queues are frequently used in computer programming and a typical example is the creation of job queue by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system.

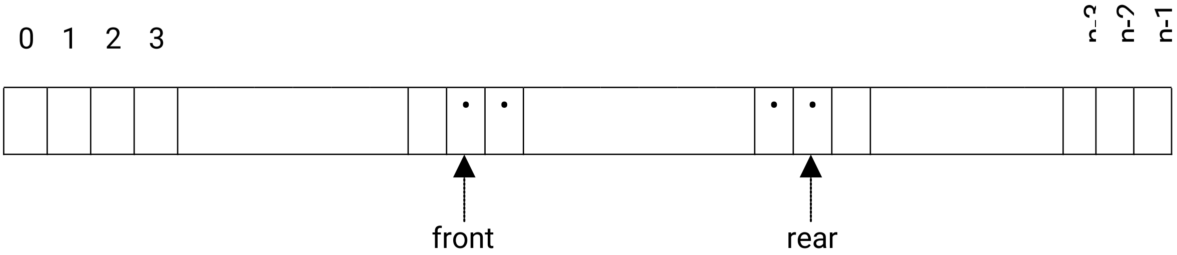
1.3.1 Queue and its operations – Insert and Delete

There are two ways to represent a queue in memory:

1. Using an array :- Used where a queue of fixed size is required.
2. Using Linked List :- Uses doubly linked list and provides a queue whose size can vary during processing.

Representation of queues using arrays

A one-dimensional array, say Q[0 ...... n-1] can be used to represent a queue. The following figure shows an instance of such a queue.



"front" and ***"rear"*** are two pointers, used to indicate two ends of the queue. For insertion, pointer *rear* will be considered and for deletion, pointer *front* will be considered.

Three states of the queue here are:

1. Queue is Empty: front = rear = -1
2. Queue is Full: rear = MaxSize-1
3. Queue contains element >= 0: front <= rear

   No. of elements = rear – front + 1

The Queue Abstract Data Type

```
template <class KeyType>
class Queue
{
// objects: A finite ordered list with zero or more elements.
public:
   Queue(int MaxQueueSize = DefaultSize);
   // Create an empty queue whose maximum size is MaxQueueSize

   Boolean IsQueueFull( );
    // if number of elements in the queue is equal to the maximum size of the queue, return
TRUE(1);
   // otherwise, return FALSE(0).

   Boolean IsQueueEmpty( );
   // if number of elements in the queue is equal to 0, return TRUE(1) else return FALSE(0).
   void Enqueue(const KeyType& item);
   // if IsQueueFull( ), then QueueFull( ), else insert item at rear of the queue.

   KeyType* Dequeue(KeyType&);
   // if IsQueueEmpty( ), then QueueEmpty( ) and return 0; else remove the item at the front
of the
  // queue and return a pointer to it.
};
```

ADT 1.3: Abstract Data Type *Queue*

The representation of a queue in sequential locations is more difficult than that of the stack. The simplest scheme employs a one-dimensional array and two variables, *front* and *rear*. *front* is one less than the position of the first element in the queue, and *rear* is the position of the last element in the queue. This results in the following data member declarations and constructor definition of *Queue*:

```
private:
  int front, rear;
  KeyType* Q;
  int MaxSize;

template<class KeyType>
Queue<KeyType>::Queue (int MaxQueueSize) : MaxSize (MaxQueueSize)
{
   Q = new KeyType[MaxSize];
   front = rear = -1;
}
```

The member functions *IsQueueFull*( ) and *IsQueueEmpty*( ) are implemented as follows:

```
template<class KeyType>
inline Boolean Queue<KeyType> :: IsQueueFull( )
{
   if (rear = = MaxSize-1) return TRUE;
   else return FALSE;
}

template<class KeyType>
inline Boolean Queue<KeyType> :: IsQueueEmpty( )
{
   if (front = rear = -1) return TRUE;
   else return FALSE;
}
```

The member functions *enqueue* and *dequeue* for insertion and deletion respectively are as follows:

```
template<class KeyType>
void Queue<KeyType> :: Enqueue(const KeyType& item)
```

```
// add item to the queue
{
   if (IsQueueFull( )) QueueFull( );
   else if (IsQueueEmpty( )) { front = 0;}
  Q[++rear] = item;
}

template<class KeyType>
KeyType* Queue<KeyType> :: Dequeue( )
// remove front element from the queue
{
   if (IsQueueEmpty( )) { QueueEmpty( ); return 0; }
   item = Q[front];
   if ( front = =  rear) { rear = front = -1; }
   else front++;
   return item;
}
```

1.3.2 Circular Queue and its array representation

For a queue represented using array, when the *rear* pointer reaches at the end, insertion will be denied even if room is available at the *front*. The following table shows this kind of worst case that could happen.

front	*rear*	*CQ*[0]	*CQ*[1]	*CQ*[2]	....	[*n*-1]	Next Operation
0	*n*-1	J1	J2	J3	....	J*n*	initial state
1	*n*-1	▨	J2	J3	....	J*n*	delete J1
1	*n*-1	*(free)*	J2	J3	....	J*n*	add J*n*+1 (not added since queue is full even though there is available space at the *front*)

One way to overcome this disadvantage of queue is to use a ***circular array***. Physically, a circular array is the same as ordinary array, say *CQ*[0 .... *n*-1], but logically, it implies that *CQ*[0] comes after *CQ*[*n*-1]. The following figures show physical and logical representation for a circular array.

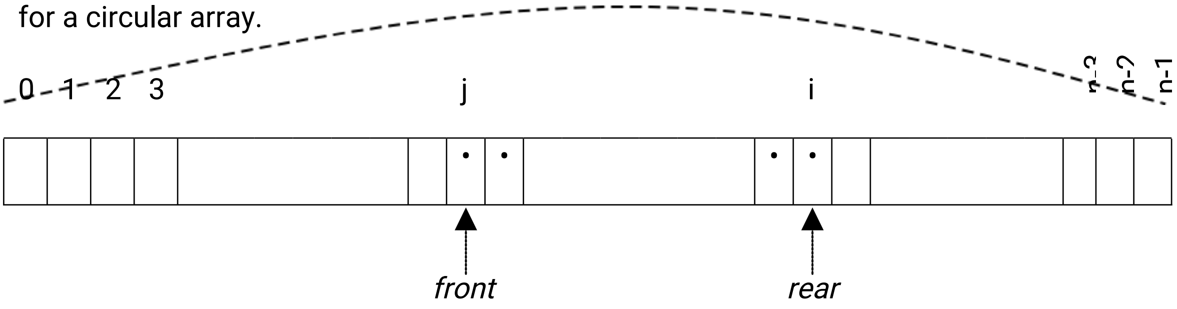


Fig : Circular array (Physical)

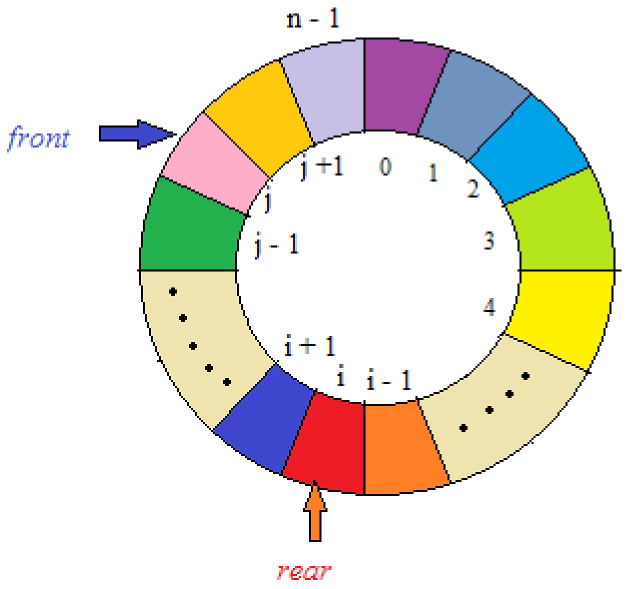


Fig : Circular queue (Logical)

Both pointers (*front* and *rear*) will move in the clockwise direction. This is controlled by the ***mod (%)***function. For example, if the current pointer is at ***i***, then shift to the next location will be (***i*** +1) ***mod*** (*MaxSize*) where 0 ≤ ***i*** ≤ *MaxSize* – 1(*MaxSize* is queue length or queue size). Thus, if ***i*** = *MaxSize* – 1, then the next position is 0.

Two states

1. Circular queue is empty means *front* = *rear* = -1
2. Circular queue is full means *front* = ((*rear* + 1) % *MaxSize*)

1.3.3 Queue ADT (for circular queue) with insert( ) / CEnqueue, delete( ) / CDequeue, QEmpty( ) and QFull( )

A more efficient queue representation is obtained by regarding the array *CQ*[*MaxSize*] as circular. When *rear* = *MaxSize* – 1, the next element is entered as *CQ*[0] in case that spot is free. The queue would be empty or full if it satisfies the above mentioned two states respectively. Initially, we have *front* = *rear* = -1. The following ADT shows the Circular Queue ADT which is the same a Queue ADT except for differences in the definition of the member functions.

```
template <class KeyType>
class CQueue
{
// objects: A finite ordered list with zero or more elements.
public:
   CQueue(int MaxCQueueSize = DefaultSize);
   // Create an empty queue whose maximum size is MaxCQueueSize

   Boolean IsCQueueFull( );
    // if number of elements in the queue is equal to the maximum size of the queue, return
TRUE(1);
```

```
   // otherwise, return FALSE(0).

   Boolean IsCQueueEmpty( );
   // if number of elements in the queue is equal to 0, return TRUE(1) else return FALSE(0).


   void CEnqueue(const KeyType& item);
   // if Is CQueueFull( ), then CQueueFull( ), else insert item at rear of the queue.

   KeyType* CDequeue(KeyType&);
   // if IsCQueueEmpty( ), then CQueueEmpty( ) and return 0; else remove the item at the
front of the
  // queue and return a pointer to it.
};
```

ADT 1.4: Abstract Data Type *Circular Queue*

The data member declarations and constructor definition of *Circular Queue* is as follows:

```
private:
  int front, rear;
  KeyType* CQ;
  int MaxSize;

template<class KeyType>
CQueue<KeyType>::CQueue (int MaxCQueueSize) : MaxSize (MaxCQueueSize)
{
   CQ = new KeyType[MaxSize];
   front = rear = -1;
}
```

The member functions *IsCQueueFull*( ) and *IsCQueueEmpty*( ) are implemented as follows:

```
template<class KeyType>
inline Boolean CQueue<KeyType> :: IsCQueueFull( )
{
   if ((rear + 1) % MaxSize == front) return TRUE;
   else return FALSE;
}

template<class KeyType>
inline Boolean CQueue<KeyType> :: IsCQueueEmpty( )
{
   if (front = rear = -1) return TRUE;
   else return FALSE;
}
```

The member functions enqueue and dequeue for insertion and deletion respectively are as follows:

```
template<class KeyType>
```

```
void CQueue<KeyType> :: CEnqueue(const KeyType& item)
// add item to the queue
{
   if (IsCQueueFull( )) { CQueueFull( ); return; }
   else if (IsCQueueEmpty( )) { front = 0; rear = 0; }
        else rear = (rear + 1) % MaxSize;
   CQ[rear] = item;
}

template<class KeyType>
void CQueue<KeyType> :: CDequeue( )
// remove front element from the queue
{
   if (IsCQueueEmpty( )) { CQueueEmpty( ); return 0; }
   item = CQ[front];
   if ( front = =  rear) { rear = front = -1; }
   else front = (front + 1) % MaxSize;
   return item;
}
```

1.3.4 Priority Queue and Deque

1.3.4.1 Priority Queue

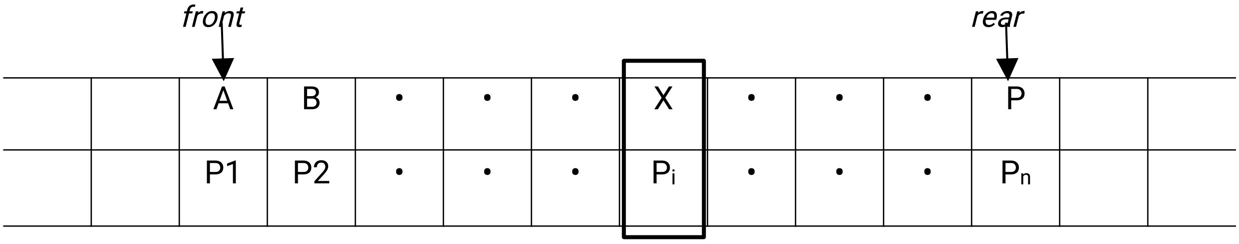
A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules.

1. An element of higher priority is processed before any element of lower priority.

2. Two elements with the same priority are processed according to the order in which they are added to the queue.

A prototype of a priority queue is a time-sharing system: programs of higher priority are processed first, and programs with the same priority form a standard queue.

In a priority queue, each element has been assigned a value, called ***priority*** of the element, and an element can be inserted or deleted not only at the ends, but at any position on the queue. The following figure shows a priority queue.



Here, an element X of priority P_i may be deleted before and element which is at *front*. Similarly, insertion of an element is based on its priority. Instead of adding it after the rear, it may be inserted at an intermediate position dictated by its priority value.

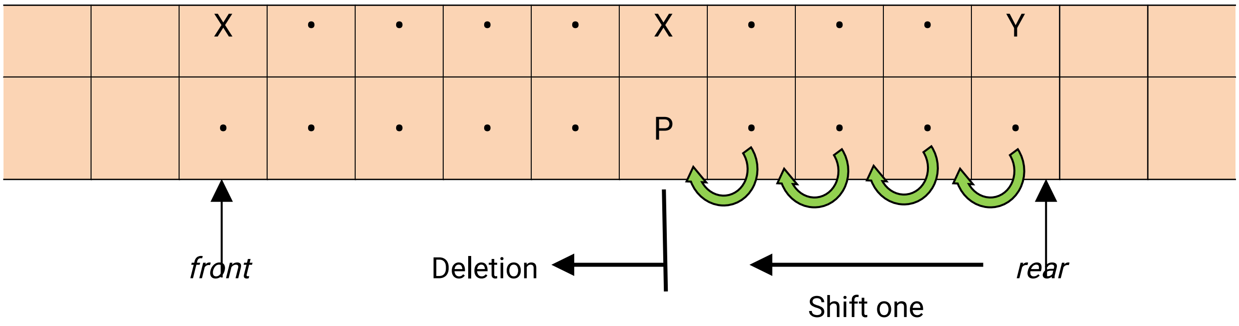
There are various ways of implementing the structure of a priority queue. These are

i. Using a simple / circular array
ii. Multi-queue representation
iii. Using a doubly linked list
iv. Using a heap tree

i. Priority queue using an array

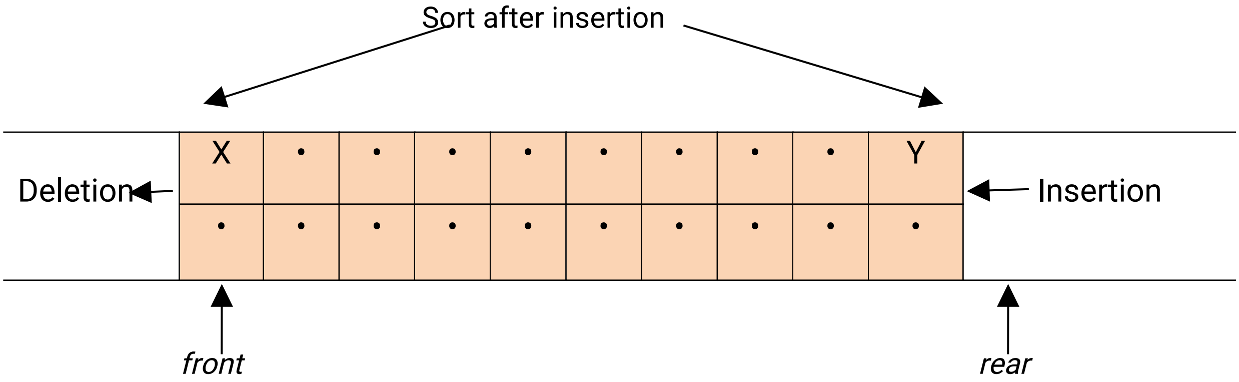
In this representation, an array can be maintained to hold the item and its priority value. The element will be inserted at the *rear* end. The deletion operation will then be performed either of the two following ways:

a) Starting from the *front* pointer, traverse the array for an element of the highest priority. Delete this element from the queue. If this is not the front-most element, shift all its trailing elements after the deleted element one stroke each to fill up the vacant position.



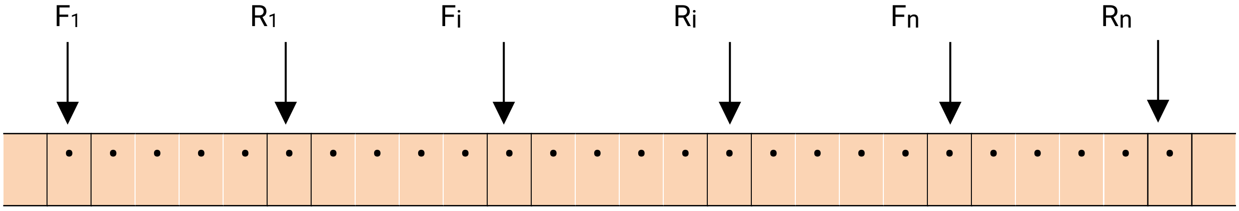
This implementation is very inefficient as it involves searching the queue for the highest priority element and shifting the trailing elements after the deletion. A better implementation is as follows:

b) Add the elements at the *rear* end as earlier. Using a stable sorting algorithm (stable sorting algorithm is a sorting algorithm in which the relative positions of two identical items remain the same in the unsorted and sorted list), sort the elements of the queue so that the highest priority elements is at the *front* end. When a deletion is required, delete it from the *front* end only.



ii. Multi-queue representation

This implementation assumes N different priority values. For each priority P_i, there are two pointers F_i and R_i corresponding to front and rear pointers respectively. The elements between F_i and R_i are all of equal priority value P_i. The following figure represents a view of such structure.



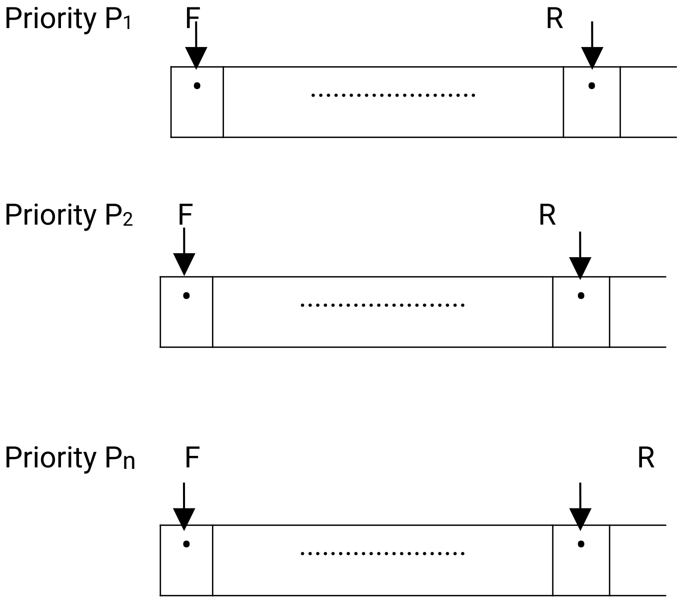
With this representation, an element with priority value P_i will consult F_i for deletion and R_i for insertion. But this implementation is associated with number of difficulties:

1. It may lead to huge shifting in order to make a room for an item to be inserted.
2. Large number of pointers are involved when the range of priority values are large.

There are two other techniques to represent multi-queue which are shown below:

a) Multiple queues with simple queues

Here, for each priority value, a simple queue is to be maintained. An element will be added into a particular queue depending on its priority value.

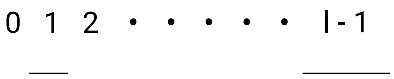


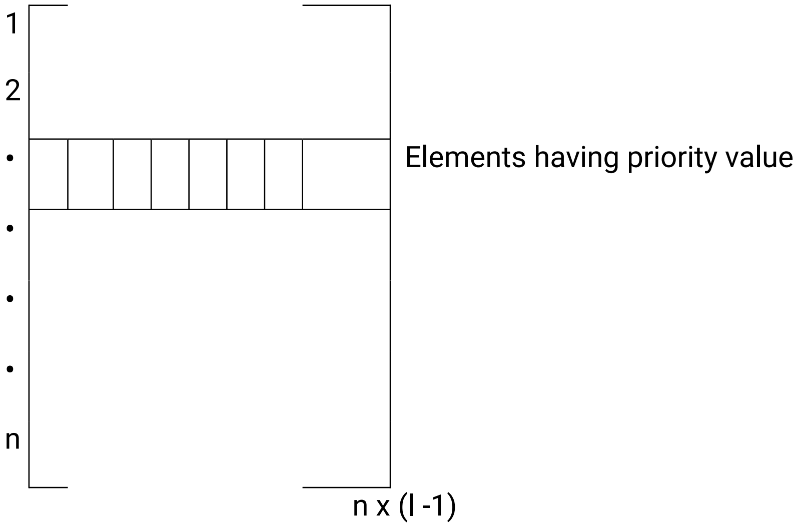
Advantage:

Multiqueue with multiple queues can have different queues of arbitrary length.

b) Matrix representation

This is better than the multiple queue representation. Here, we can get rid off maintaining several pointers for front and rear in several queues.

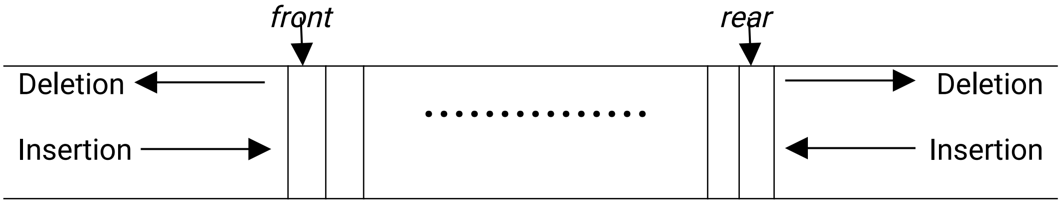




Both these representations are not economic from the memory utilization point of view; majority of the memory space remains vacant.

1.3.4.2 Deque

Another variation of the queue is the deque. Unlike queue, in deque, both insertion and deletion operations can be made at either end of the structure. Actually, the term deque is originated from double ended queue. Such a structure is shown below.



From the deque structure, it is clear that it is a general representation of both stack and queue, or in other words, a deque can be used as a stack as well as a queue.

A simple method to represent a deque is using a doubly linked list. Another popular representation is using a circular array.

The following four operations are possible on a deque which consists a list of items:

1. *PUSHDQ*(*item*) : To insert *item* at the *front* end of the deque.
2. *POPDQ*( ) : To remove the *front item* from the deque.
3. *INJECT*(*item*) : To insert *item* at the *rear* end of the deque.
4. *EJECT*( ) : To remove the *rear item* from the deque.

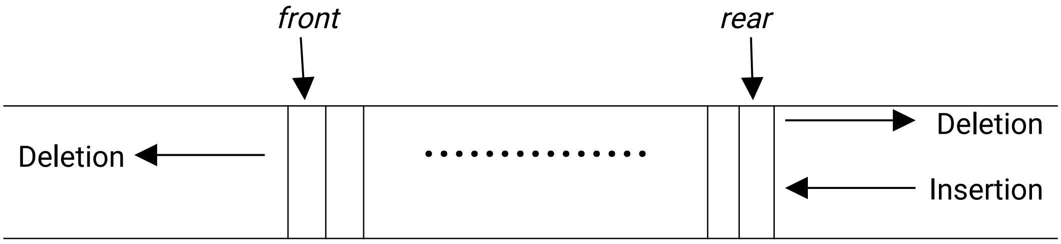
Types of Deque

There are two variations if deques:

1. Input restricted deque
2. Output restricted deque

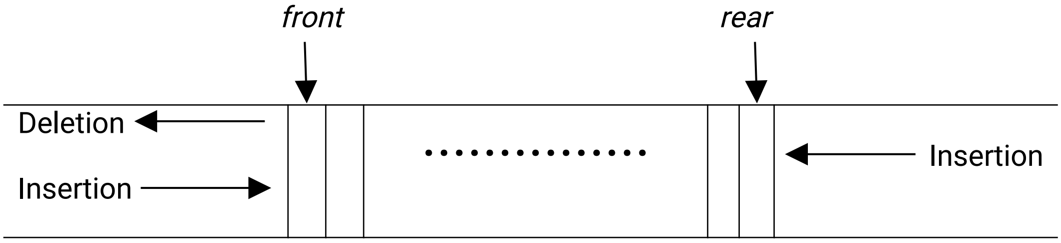
1. **Input restricted deque**:

It is a deque which allows insertions at one end (say rear end) only, but allows deletions at both ends.



2. **Output restricted deque**:

It is a deque where deletions take place at one end (say front end) only, but allows insertions at both ends.


front
rear
Deletion
Insertion
Insertion