Introduction to Data Structures & Algorithms

Session Objectives

- To understand the concepts of
  - Data structures
  - Types of Data Structures
  - Applications
  - Algorithms
  - ADTs

Session Topics

- Algorithms
- ADTs
- Properties of an Algorithm
- Data structures
- Types of Data structures
- Problem Solving Phase
- Stacks and Queues

Good Computer Program

- A computer program is a series of instructions to carry out a particular task written in a language that a computer can understand.
- The process of preparing and feeding the instructions into the computer for execution is referred as programming.
- There are a number of features for a good program
  - Run efficiently and correctly
  - Have a user friendly interface
  - Be easy to read and understand
  - Be easy to debug
  - Be easy to modify
  - Be easy to maintain

Good Computer Program

- Programs consists of two things: Algorithms and data structures
- A Good Program is a combination of both algorithm and a data structure
- An algorithm is a step by step recipe for solving an instance of a problem
- A data structure represents the logical relationship that exists between individual elements of data to carry out certain tasks
- A data structure defines a way of organizing all data items that consider not only the elements stored but also stores the relationship between the elements

Algorithms

- An algorithm is a step by step recipe for solving an instance of a problem.
- Every single procedure that a computer performs is an algorithm.
- An algorithm is a precise procedure for solving a problem in finite number of steps.
- An algorithm states the actions to be executed and the order in which these actions are to be executed.
- An algorithm is a well ordered collection of clear and simple instructions of definite and effectively computable operations that when executed produces a result and stops executing at some point in a finite amount of time rather than just going on and on infinitely.

Algorithm Properties

An algorithm possesses the following properties:

- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.
- It takes zero or more inputs
- It should be efficient and flexible
- It should use less memory space as much as possible
- It results in one or more outputs

Various steps in developing Algorithms

- Devising the Algorithm:

  It’s a method for solving a problem. Each step of an algorithm must be precisely defined and no vague statements should be used. Pseudo code is used to describe the algorithm , in less formal language than a programming language.

- Validating the Algorithm:

  The proof of correctness of the algorithm. A human must be able to perform each step using paper and pencil by giving the required input , use the algorithm and get the required output in a finite amount of time.

Various steps in developing Algorithms

- Expressing the algorithm:

  To implement the algorithm in a programming language.

  The algorithm used should terminate after a finite number of steps.

Efficiency of an algorithm

- Writing efficient programs is what every programmer hopes to be able to do. But what kinds of programs are efficient? The question leads to the concept of generalization of programs.
- Algorithms are programs in a general form. An algorithm is an idea upon which a program is built. An algorithm should meet three things:

  It should be independent of the programming language in which the idea is realized

  Every programmer having enough knowledge and experience should understand it

  It should be applicable to inputs of all sizes

Efficiency of an algorithm

- Efficiency of an algorithm denotes the rate at which an algorithm solves a problem of size n.
- It is measured by the amount of resources it uses, the time and the space.
- The time refers to the number of steps the algorithm executes while the space refers to the number of unit memory storage it requires.
- An algorithm's complexity is measured by calculating the time taken and space required for performing the algorithm.
- The input size, denoted by n, is one parameter , used to characterize the instance of the problem.
- The input size n is the number of registers needed to hold the input (data segment size).

Time Complexity of an Algorithm

- Time Complexity of an algorithm is the amount of time(or the number of steps) needed by a program to complete its task (to execute a particular algorithm)
- The way in which the number of steps required by an algorithm varies with the size of the problem it is solving. The time taken for an algorithm is comprised of two times

    Compilation Time

    Run Time

- Compilation time is the time taken to compile an algorithm. While compiling it checks for the syntax and semantic errors in the program and links it with the standard libraries , your program has asked to.

Time Complexity of an Algorithm

- Run Time: It is the time to execute the compiled program.

  The run time of an algorithm depend upon the number of instructions present in the algorithm. Usually we consider, one unit for executing one instruction.
- The run time is in the control of the programmer , as the compiler is going to compile only the same number of statements , irrespective of the types of the compiler used.
- Note that run time is calculated only for executable statements and not for declaration statements
- Time complexity is normally expressed as an order of magnitude, eg O(n^2) means that if the size of the problem n doubles then the algorithm will take four times as many steps to complete.

Time Complexity of an Algorithm

- Time complexity of a given algorithm can be defined for computation of function f() as a total number of statements that are executed for computing the value of f(n).
- Time complexity is a function dependent from the value of n. In practice it is often more convenient to consider it as a function from |n|
- Time complexity of an algorithm is generally classified as three types.

  (i) Worst case

  (ii) Average Case

  (iii) Best Case

Time Complexity

- Worst Case: It is the longest time that an algorithm will use over all instances of size n for a given problem to produce a desired result.

- Average Case: It is the average time(or average space) that the algorithm will use over all instances of size n for a given problem to produce a desired result. It depends on the probability distribution of instances of the problem.

- Best Case: It is the shortest time (or least space) that the algorithm will use over all instances of size n for a given problem to produce a desired result.

Space Complexity

- Space Complexity of a program is the amount of memory consumed by the algorithm (apart from input and output, if required by specification) until it completes its execution.
- The way in which the amount of storage space required by an algorithm varies with the size of the problem to be solved.
- The space occupied by the program is generally by the following:

  A fixed amount of memory occupied by the space for the program code and space occupied by the variables used in the program.

  A variable amount of memory occupied by the component variable whose size is dependent on the problem being solved. This space increases or decreases depending upon whether the program uses iterative or recursive procedures.

Space Complexity

- The memory taken by the instructions is not in the control of the programmer as its totally dependent upon the compiler to assign this memory.
- But the memory space taken by the variables is in the control of a programmer. More the number of variables used, more will be the space taken by them in the memory.
- Space complexity is normally expressed as an order of magnitude, eg $O(n^2)$means that if the size of the problem n doubles then four times as much working storage will be needed.
- There are three different spaces considered for determining the amount of memory used by the algorithm.

Space Complexity

- Instruction Space is the space in memory occupied by the compiled version of the program. We consider this space as a constant space for any value of n. We normally ignore this value , but remember that is there. The instruction space is independent of the size of the problem
- Data Space is the space in memory , which used to hold the variables , data structures, allocated memory and other data elements. The data space is related to the size of the problem.
- Environment Space is the space in memory used on the run time stack for each function call. This is related to the run time stack and holds the returning address of the previous function. The memory each function utilises on the stack is a constant as each item on the stack has a return value and pointer on it.

Iterative Factorial Example

```
fact ( long n)
{
  for (i=1; i<=n; i++)
x=i*x;
return x;
}
```

- Space occupied is
- Data Space: i, n and x
- Environment Space: Almost nothing because the function is called only once.
- The algorithm has a complexity of O(1) because it does not depend on n. No matter how big the problem becomes ,the space complexity remains the same since the same variables are used , and the function is called only once.

Recursive Factorial Example

```
long fact (long x)
{
if (x<=1)
return(1);
else
return (x * fact(x-1));
}
```

- Space occupied is
- Data space : x
- Environment Space: fact() is called recursively , and so the amount of space this program used is based on the size of the problem

Recursive Factorial Example

- The space complexity is

O(n)= (x + function call) * n

= x + function call + memory needed for fact(x-1)

x+function call + x+ function call + ……+

x+ function call n(n-1)+…….+1

- Note that in measuring space complexity, memory space is always allocated for variables whether they are used in the program or not.
- Space Complexity is not as big of an issue as time complexity because space can be reused, whereas time cannot.

Problem Solving Phase

- A problem/Project needs programs to create ,append, update the database, print data, permit online enquiry and so on.
- A Programmer should identify all requirements to solve the problem. Each problem should have the following specifications

  Type of Programming language

  Narration of the program describing the tasks to be performed

  Frequency of Processing (hourly, daily, weekly etc)

  Output and input of the program

  Limitations and restrictions for the program

  Detailed Specifications

Method 1: Algorithm Analysis

Code each algorithm and run them to see how long they take.

Problem: How will you know if there is a better program or whether there is no better program?

What will happen when the number of inputs is twice as many? Three? A hundred?

Method 2: Algorithm Analysis

Develop a model of the way computers work and compare how the algorithms behave in the model.

Goal: To be able to predict performance at a coarse level. That is, to be able to distinguish between good and bad algorithms.

Another benefit: when assumptions change, we can predict the effects of those changes.

Why algorithm analysis?

As computers get faster and problem sizes get bigger, analysis will become *more* important.

Why? The difference between good and bad algorithms will get bigger.

How to Measure Algorithm Performance

- What metric should be used to judge algorithms?
  - Length of the program (lines of code)
  - Ease of programming (bugs, maintenance)
  - Memory required
  - ❑ Running time

- **Running time is the dominant standard.**
  - Quantifiable and easy to compare
  - Often the critical bottleneck

The Need for Data Structures

Data structures organize data

$\Rightarrow$ more efficient programs.

More powerful computers $\Rightarrow$ more complex applications.

More complex applications demand more calculations.

Complex computing tasks are unlike our everyday experience.

- More typically, a *data structure* is meant to be an *organization for a collection of data items*.
- Any organization for a collection of records can be searched, processed in any order, or modified.
- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days. A data structure requires a certain amount of:
- space for each data item it stores
- time to perform a single basic operation
- programming effort.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Data Structures

DS includes

- Logical or mathematical description of the structure and Implementation of the structure on a computer
- Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure.

Classification of Data Structures

"***Data Structures*** "deals with the study of how the data is organized in the memory, how efficiently the data can be retrieved and manipulated, and the possible ways in which different data items are logically related.

Types:

Primitive Data Structure: Ex. int,float,char

Non-primitive Data Structures:
Ex.Arrays,Structures,stacks

Linear Data Structures: Ex.Stacks,queues,linked list

Non-Linear Data Structures: Ex.Trees,Graphs.

Classification of Data Structures

1. Primary Data structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers.
2. All the basic constants (integers, floating point numbers, character constants, string constants)and pointers are considered as primary data structures
3. Secondary Data Structures are more complicated data structures derived from primary data structures
4. They emphasize on grouping same or different data items with relationship between each data item
5. Secondary data structures can be broadly classified as static data structures and dynamic data structures

Classification of Data Structures

- If a data structure is created using static memory allocation (ie. a data structure formed when the number of data items are known in advance), it is known as static data structure or fixed size data structure
- If a data structure is created , using dynamic memory allocation(ie. a data structure formed when the number of data items are not known in advance) it is known as dynamic data structure or variable size data structure
- Dynamic data structures can be broadly classified as linear data structures and non linear data structures
- Linear data structures have a linear relationship between its adjacent elements. Linked lists are examples of linear data structures.

Classification of Data Structures

- A linked list is a linear dynamic data structure that can grow and shrink during its execution time
- A circular linked list is similar to a linked list except that the first and last nodes are interconnected
- Non linear data structures don’t have a linear relationship between its adjacent elements
- In a linear data structure , each node has a link which points to another node, whereas in a non linear data structure, each node may point to several other nodes
- A tree is a nonlinear dynamic data structure that may point to one or more nodes at a time
- A graph is similar to tree except that it has no hierarchical relationship between its adjacent elements

Abstract data type (ADTs)

- A data type that is defined entirely by a set of operations is referred to as Abstract data type or simply ADT
- Abstract data types are a way of separating the specification and representation of data types
- An ADT is a black box, where users can only see the syntax and semantics of its operations
- An ADT is a combination of interface and implementation The interface defines the logical properties of the ADT and especially the signatures of its operations
- The implementation defines the representation of data structure and the algorithms that implement the operations
- An abstract data type encapsulates data and functions into a named data type

Abstract data type (ADTs)

- It is similar to a structure in C, but can include functions in it
- The basic difference between ADTs and primitive data types is that the latter allow us to look at the representation, whereas former hide the representation from us
- An ADT consists of a collection of values and operations with the values derive their meaning solely through the operations that can be performed upon them
- Benefits of using ADTs:

  Code is easier to understand

  Implementations of ADTs can be changed without requiring changes to the program that uses the ADTs

Data Structures: *Data Collections*

- Linear structures
  - Array: Fixed-size
  - Linked-list: Variable-size
  - Stack: Add to top and remove from top
  - Queue: Add to back and remove from front
  - Priority queue: Add anywhere, remove the highest priority
- Tree: A branching structure with no loops
- Hash tables: Unordered lists which use a ‘hash function’ to insert and search
- Graph: A more general branching structure, with less stringent connection conditions than for a tree

ADTs Collection

- *ADT is a data structure and a set of operations which can be performed on it.*
  - *A class in object-oriented design is an ADT*
- *The pre-conditions* define a state of the program which the client guarantees will be true before calling any method,
- *post-conditions* define the state of the program that the object's method will guarantee to create for you when it returns.
- create Create a new collection
- add Add an item to a collection
- delete Delete an item from a collection find Find an item matching some criterion in the collection
- destroy Destroy the collection

Data Structures and ADTs

- A container in which data is being stored
  - Example: structure, file, or array
- An ADT is a data structure which does not exist within the host language, but rather must be created out of existing tools
- It is both a collection of data and a set of rules that govern how the data will be manipulated
- Examples: list, stack, queue, tree, table, and graph
- An ADT sits on top of the data structure, and the rules that govern the use of the data define the interface between the data structure and the ADT

Lists

- Lists are ordered data sets
  - The type of ordering is entirely dependent upon the application
- Lists come in two basic forms: sequential and linked
- The primary difference is how the items are laid out in memory
  - Can be described as the difference between physical order and logical order:
    - Sequential implementation – the physical arrangement of the data elements defines the logical order of entries on the list. Typically an array-based structure, possibly of fixed maximum size
    - Linked implementation – The physical order of data elements is unrelated to the logical order of the entries on the list. Typically a linked set of nodes, each allocated dynamically as needed

Lists

- This key difference is reflected in the types of applications for which 2- list types are suited:
  - Sequential–best when the order of the data does not matter, when ordering can be done after it has all been loaded into the array, or when direct indexing of elements is useful
  - Linked– the lack of relationship between physical and logical order means that a new value can be placed anywhere in memory, lending this list type to applications where flexibility is desired

Sequential List Implementation

- Inefficiency: If the list is to be maintained in alphabetical order, inserting a new value may mean having to shift existing ones out of the way
- Likewise, if an element were to be deleted, we might have to shift remaining values 'forward' toward the front of the array

Features of Sequential Lists

- directly indexable
- easy to traverse – can move backwards and forwards easily
- fast access
- easy to implement
- contiguous locations
- physical and logical order the same
- not particularly flexible

Data Structures

- *Dynamic data structures* - grow and shrink during execution
- *Linked lists* - insertions and removals made anywhere
- *Stacks* - insertions and removals made only at top of stack
- *Queues* - insertions made at the back and removals made from the front
- *Binary trees* - high-speed searching and sorting of data and efficient elimination of duplicate data items

Stacks

A ***Stack*** is defined as a special type of data structure where items are inserted from one end called ***top*** of stack and items are deleted from the same end.

Stack is organized as a ***Last In First Out(LIFO)*** data structure.

Operations performed on Stacks are:

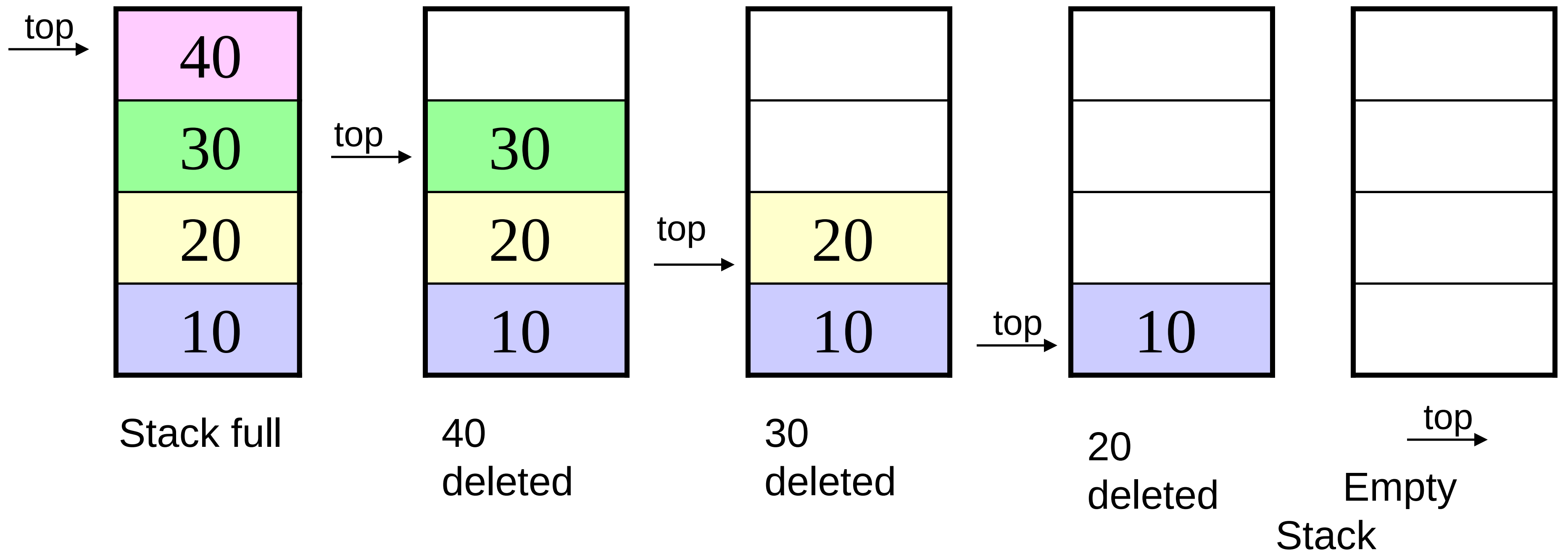
- ❖ Insert an item into the stack (Store)
- ❖ Delete an item from the stack (Retrieve)
- ❖ Display the contents of the stack

Stacks

- A stack is an ordered collection of items, generally implemented with only two principle operations, called Push and Pop.
- stack – new nodes can be added and removed only at the top
- Push adds an item to a stack
- Pop extracts the most recently

  pushed item from the stack
  - similar to a pile of dishes
  - last-in, first-out (LIFO)
  - Bottom of stack indicated by a link member to **null**
  - stores the popped value
  - constrained version of a linked list

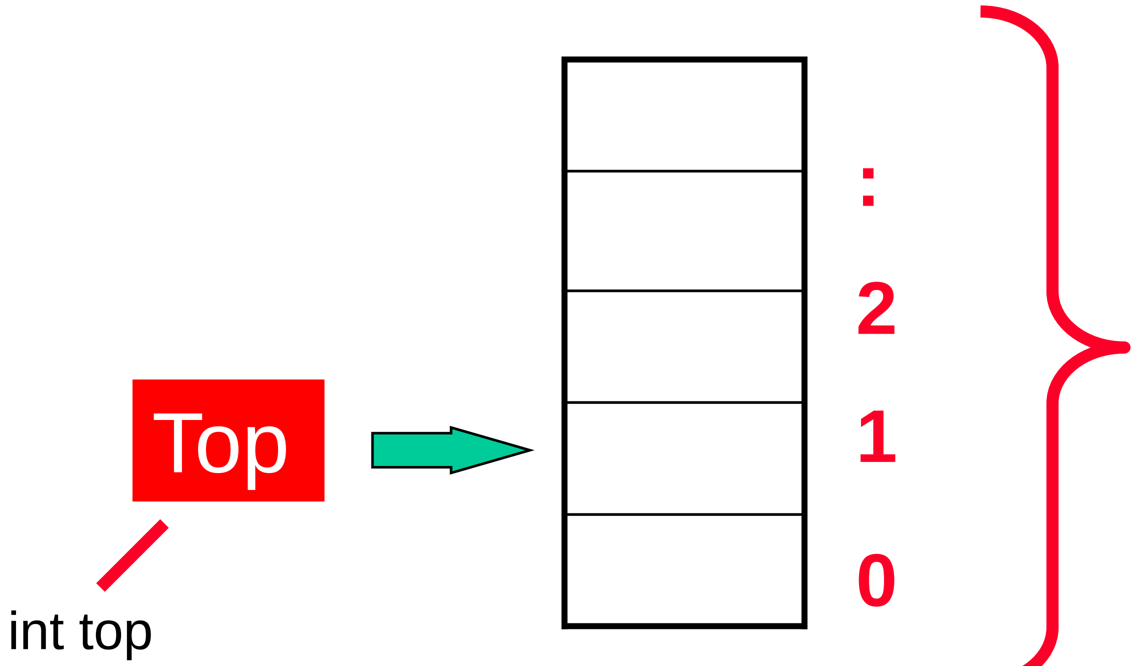
Deletion Operations



Since, items are inserted from one end, in stack deletions should be done from the same end..

When stack is empty it is not possible to delete any item and this situation is called ***Stack Underflow.***

Implementation



Insert/ Push Operation

Inserting an element into the stack is called ***push*** operation.

Function to insert an item: (Push)

```
void push(int item, int *top, int s[])
{
    if(*top == STACKSIZE - 1)   /*Is stack empty?*/
    {
            printf(“Stack Overflow\n”);
            return;
    }
    /* Increment top and then insert an item*/
    s[++(*top)] = item;
}
```

Delete/Pop Operation

Deleting an element from the stack is called ***pop*** operation.

Function to delete an item: (Pop)

```
int pop(int *top, int s[])
{
        int item_deleted /*Holds the top most item */
        if(*top == -1)
        {
                return 0;          /*Indicates empty stack*/
        }
        /*Obtain the top most element and change the position
of top item */
        item_deleted=s[(*top)--];
         /*Send to the calling function*/
        return item_deleted;
}
```

Display Procedure

If the stack already has some elements, all those items are displayed one after the other.

```
void display(int top, int s[]){
        int i;
        if(top == -1)    /* Is stack empty?*/
        {
                printf(“Stack is empty\n”);
                return;
        }
        /*Display contents of a stack*/
        printf(“Contents of the stack\n”);
        for(i = 0;i <= top; i++)
        {
                printf(“%d\n”,s[i]);
        }
}
```

Applications of Stack

- ❖ Conversion of expressions.

  Infix to postfix, postfix to prefix, prefix to infix, vice-versa.

- ❖ Evaluation of expressions.

  Arithmetic expression in the form of either postfix or prefix.

  can be easily evaluated.

- ❖ Recursion.

  Ex. Tower of Hanoi etc.

- ❖ Other applications.

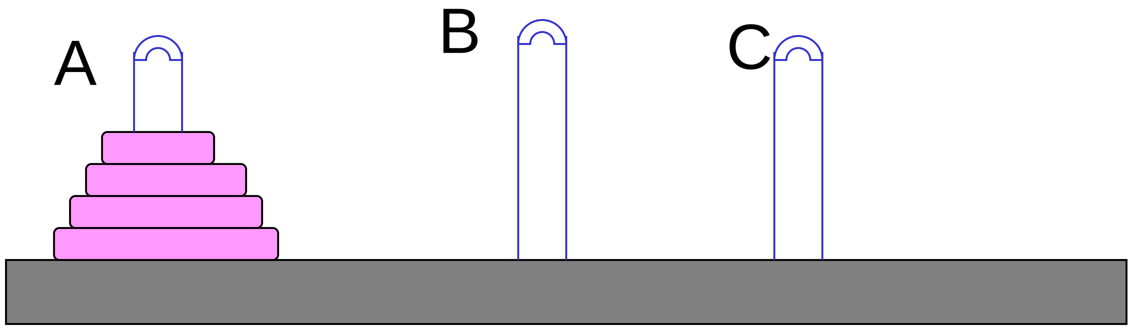
  Ex:Checking if a string is a palindrome or not.

  Topological Sort.

- ❖ System Software and Compiler Design

Towers of Hanoi

- Three pegs labeled A, B,C
- In A, there are finite number of disks with decreasing size
- Objective- move the disks from A to C using B as an auxiliary.
- The rules of the game are as follows
- Only one disk may be moved at a time.
- At no time can a larger disk be placed on a smaller disk

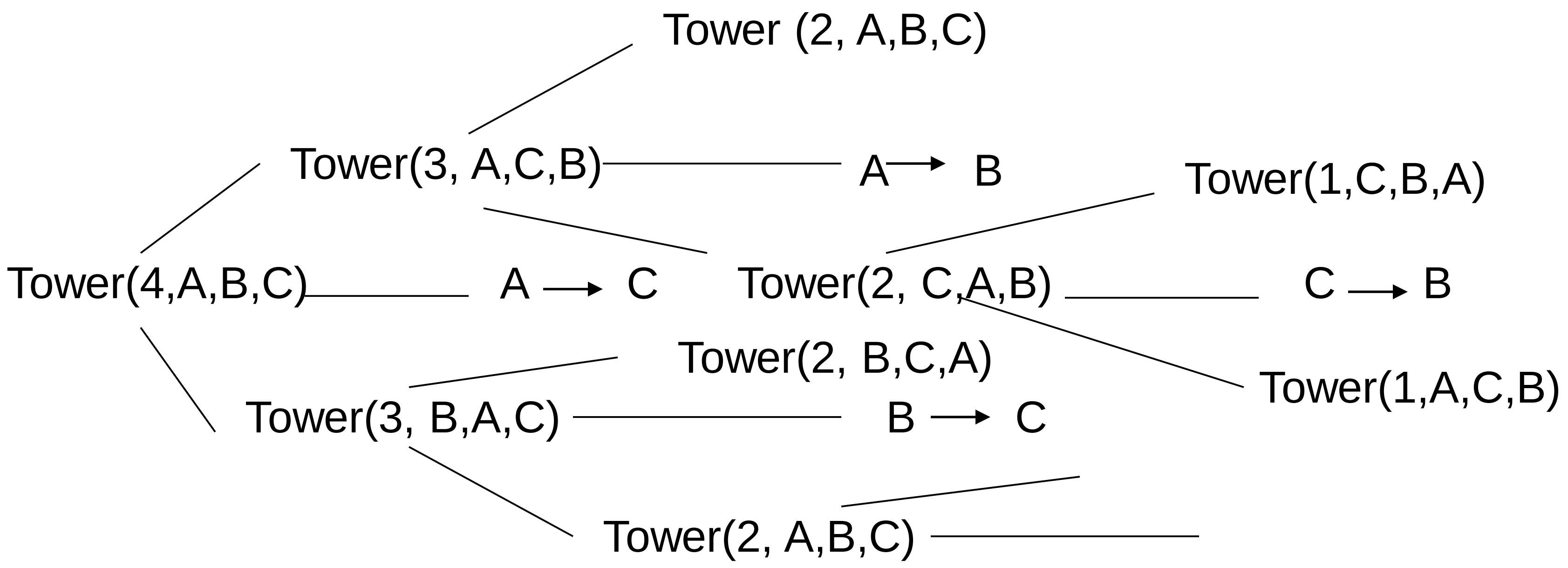


Procedure

- For n disks

  1. Move the top n-1 disks from peg A to peg B
  2. Move the top disk from peg A to C
  3. Move the top n-1 disks from peg B to peg C
- Tower(N-1, BEG, END, AUX)
- Tower(1,BEG, AUX, END)
- Tower(N-1,AUX,BEG,END)

Procedure


Tower (2, A,B,C)
Tower(3, A,C,B)
A → B
Tower(1,C,B,A)
Tower(4,A,B,C)
A → C
Tower(2, C,A,B)
C → B
Tower(2, B,C,A)
Tower(1,A,C,B)
Tower(3, B,A,C)
B → C
Tower(2, A,B,C)

Algorithm

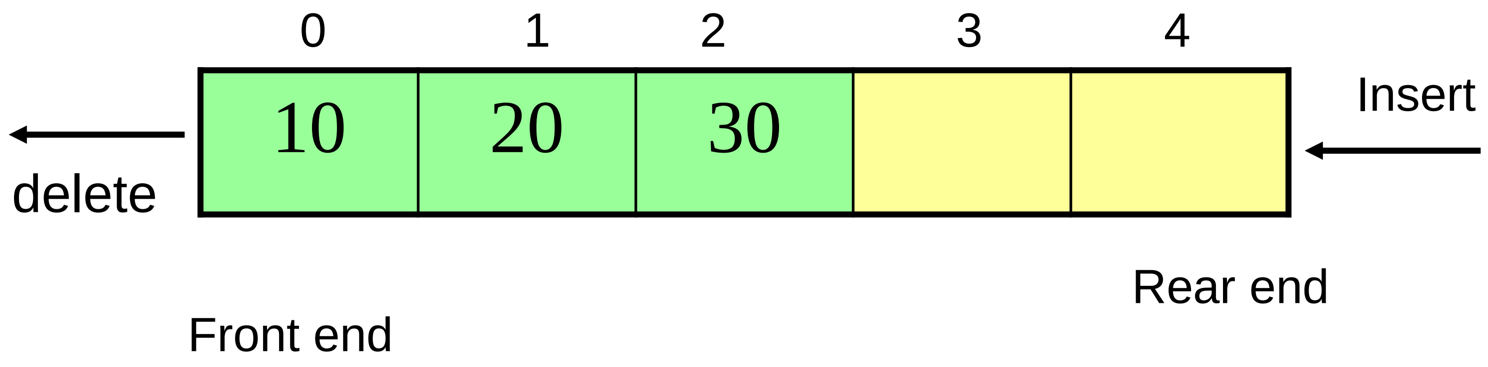
- Tower(N,BEG,AUX,END)
- If N=1, then BEG $\rightarrow$ END
-   return
- Tower(N-1, BEG,END,AUX) // Move N-1 disks from BEG to AUX
- BEG $\rightarrow$ END
- Tower(N-1, AUX, BEG,END)
- return

Queues

- A ***queue*** is defined as a special type of data structure where the elements are inserted from one end and elements are deleted from the other end.
- The end from where the elements are inserted is called ***REAR end.***
- The end from where the elements are deleted is called ***FRONT end*****.**
- Queue is organized as ***First In First Out*** (FIFO)Data Structure.

Queues

In a queue, the elements are always inserted at the rear end and deleted from the front end.



Pictorial representation of a Queue

Operations Performed on Queues

- ✓ Insert an item into a queue
- ✓ Delete an item from queue
- ✓ Display the contents of queue

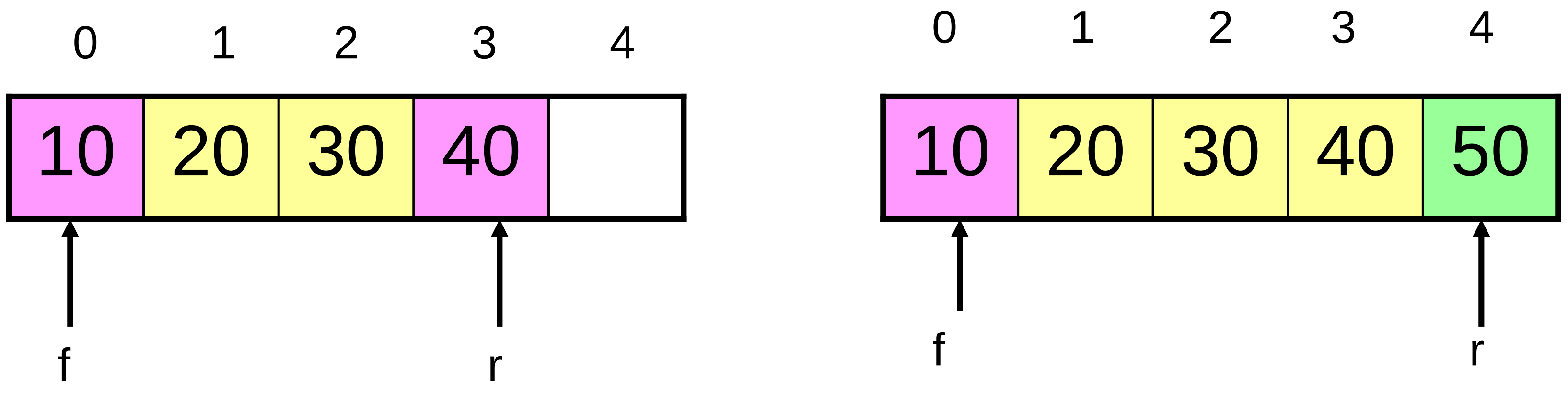
Different types of Queues

- ✓ Queue(Ordinary queue)
- ✓ Circular Queue
- ✓ Double ended Queue
- ✓ Priority Queue

Insertion of Elements

- The ordinary queue operates on first come first serve basis. Items will be inserted from one end and deleted at the other end in the same order in which they are inserted.

Insert at the rear end



To insert an item 50

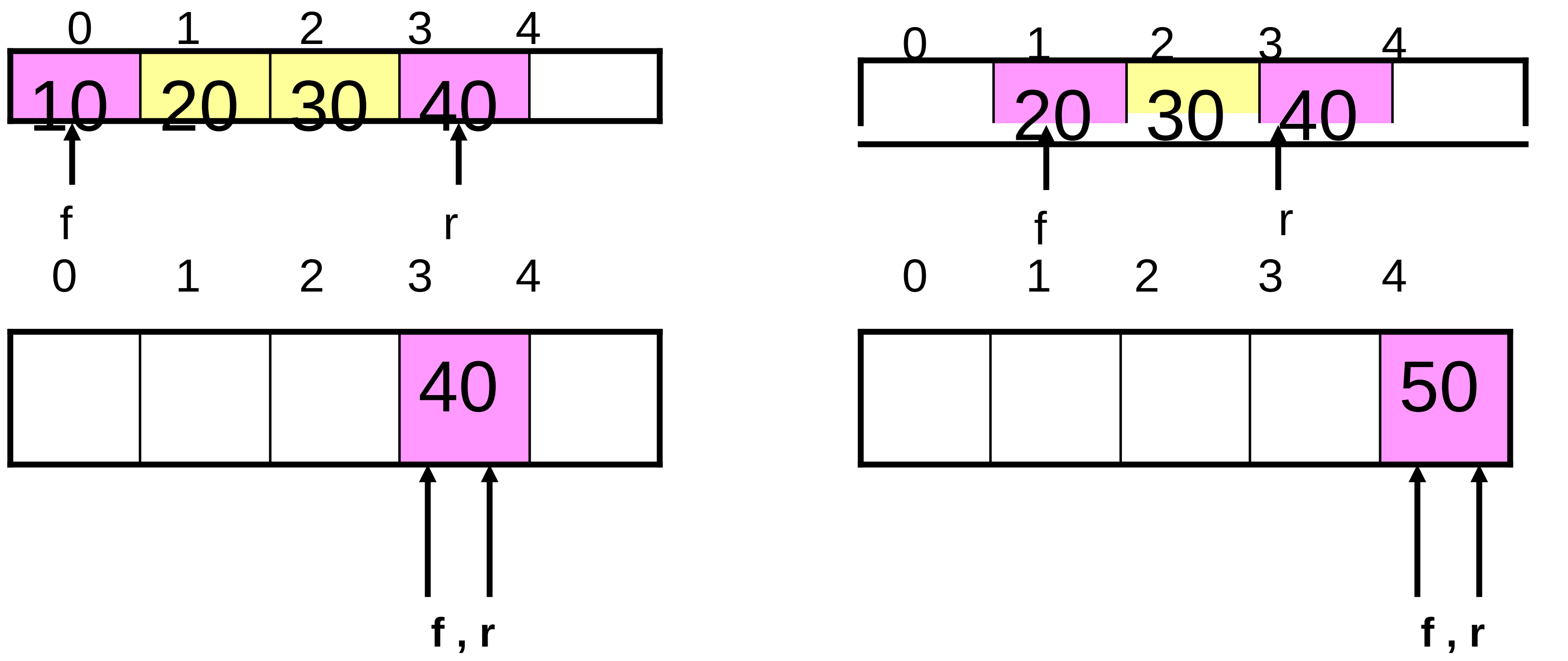
Whenever queue is full, it is not possible to insert any element into queue and this condition is called ***OVERFLOW.***

Function to insert an item at the rear end of the queue

```
void insert_rear(int item,int q[],int *r)
{
    if(q_full(*r)) /* Is queue full */
    {
            printf(“Queue overflow\n”);
            return;
    }
    /* Queue is not full */
    q[++(*r)] = item;
}
int q_full(int r)
{
    return (r == QUEUE_SIZE -1)? 1 : 0 ;
}
```

Delete from the front end

- Whenever the value of ***f*** is greater than ***r*** **,**then the queue is said to be empty and is not possible to delete an element from queue. This condition is called ***Underflow.***



Function to delete an item from the front end

```
void delete_front(int q[], int *f, int *r) {
    if( q_empty(f , r) ) /* Is queue empty*/
    {
        printf("Queue underflow\n");
        return;
    }
    printf("The element deleted is %d\n",q[(*f)++]);
    if( f > r )
    {
        f = 0;
        r = -1;
    }
}
int q_empty(int f,int r)
{
    return (f>r) ? 1 : 0;
}
```

Function to display the contents of queue

The contents of queue can be displayed only if queue is not empty. If queue is empty an appropriate message is displayed.

```
void display(int q[], int f, int r)
{
      int i;
      if( q_empty (f , r )) /* Is queue empty*/
      {
              printf("Queue is empty\n");
              return;
      }
      printf("Contents of queue is \n");
      for(i = f;i <=r;i++)
              printf("%d\n",q[i]);
}
```

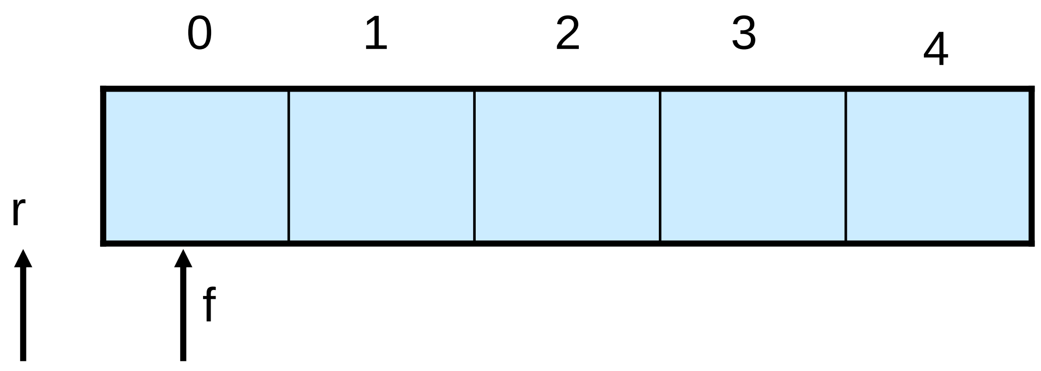
Double Ended Queue

A ***Deque*** is a special type of data structure in which insertions and deletions will be done either at the front end or at the rear end of the queue. Here, insertion and deletion are done from both the ends.

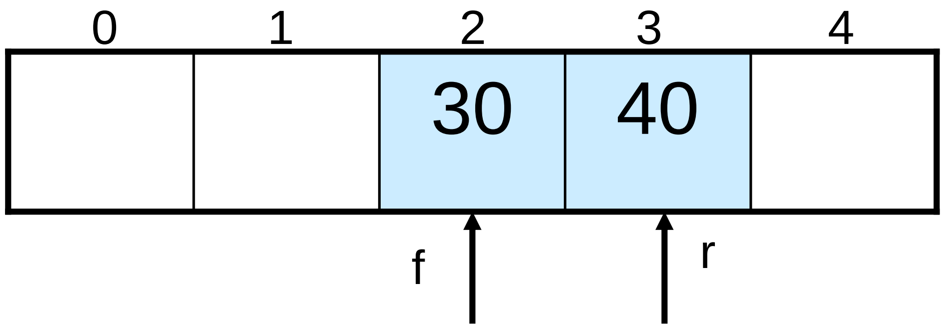
Operations performed on Deques are:

- ✓ Insert an item from front end
- ✓ Insert an item from rear end
- ✓ Delete an item from front end
- ✓ Delete an item from rear end
- ✓ Display the contents of queue

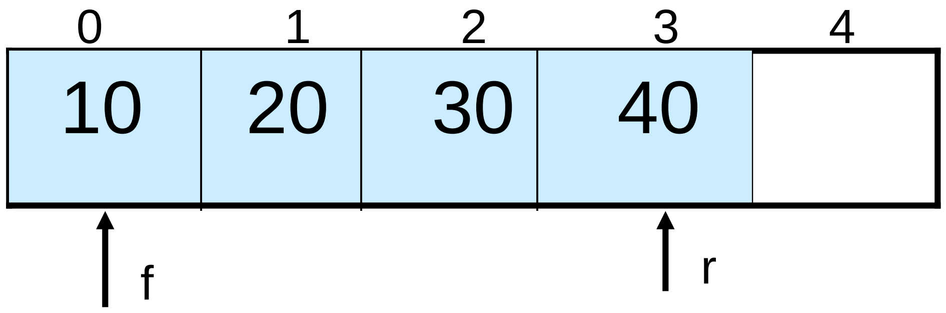
Insert at the front end



```
if(f==0 && r == -1)
    q[++r] = item;
```



```
if( f != 0)
    q[--f] = item;
```



Not possible to insert
an item at the front end

Function to insert an item at front end

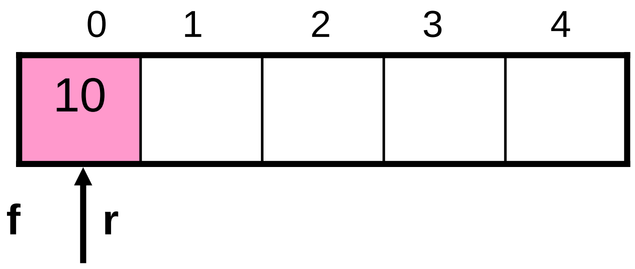
```
void insert_front(int item,int q[],int *f,int *r)
{
        if(*f == 0 && *r == -1)
        {
                q[++(*r)] = item;
                return;
        }
        if(*f != 0)
        {
                q[--(*f)] = item;
                return;
        }
        printf(“Front insertion not possible”);
}
```

Function to delete an item from the rear end

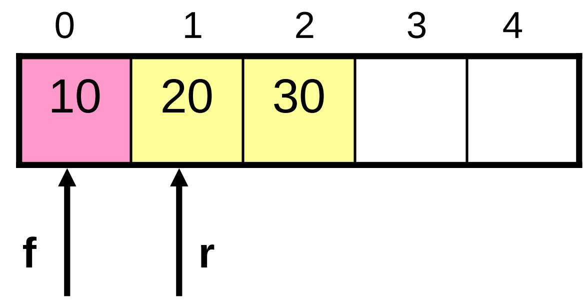
```
void delete_rear(int q[], int *f,int *r)
{
    if(q_empty(f, r))
    {
            printf(“Queue underflow\n”);
            return;
    }
    printf(“The element deleted is %d”,q[(*r)--]);
    if(f > r)
    {
            f = 0;
            r = -1;
    }
}
```

Circular Queue

- In ***circular queue***, the elements of a given queue can be stored efficiently in an array so as to **"wrap around"** so that end of the queue is followed by the front of the queue.

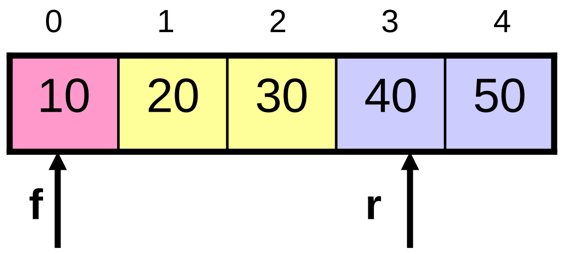


Initial queue

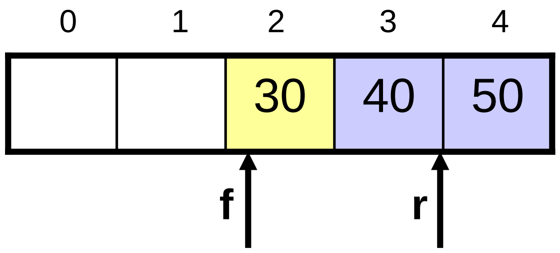


After inserting 20 & 30

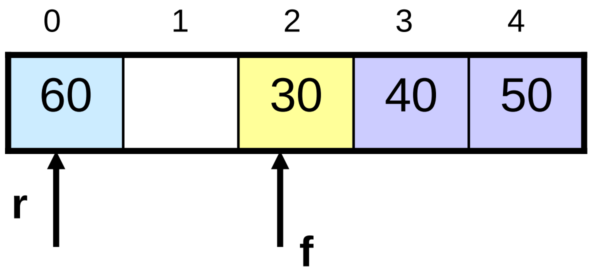
Circular Queue



After inserting 40 and 50



After deleting 10 and 20



After inserting 60

Operations Performed on Circular Queues

- Insert rear
- Delete front
- Display the contents of Queue

Function to insert an item at the rear end

```
void insert_rear(int item,int q[],int *r,int *count)
{
    if(q_full(*count))
    {
            printf("Overflow of Queue\n");
            return;
    }
    r = (*r + 1) % QUEUE_SIZE;     /* Increment rear pointer */
    q[*r]=item;                        /* Insert the item */
    *count +=1;                        /* Update the counter */
}
int(q_full(int count))
{
    /* Return true if Q is full,else false */
    return (count == QUEUE_SIZE - 1) ? 1 : 0;
}
```

Function to delete an item from the front end

```
void delete_front(int q[], int *f, int *count)
{
      if(q_empty(*count))
      {
                printf("Underflow of queue\n");
                return;
      }
      /* Access the item */
      printf("The deleted element is %d\n",q[*f]);
      /* Point to next first item */
      f = (*f + 1) % QUEUE_SIZE;
      /* Update counter */
      *count -= 1;
}
int(q_empty(int count))
{
       /* Return true if Q is empty ,else false */
      return(count == 0) ? 1 : 0;
}
```

Priority Queue

- The priority queue is a special type of data structure in which items can be inserted or deleted based on the priority.
- If the elements in the queue are of same priority, then the element, which is inserted first into the queue, is processed.

Types of Priority queues:

- Ascending priority queue –

  Elements can be inserted in any order.

  While deleting, smallest item is deleted first.
- Descending priority queue –

  Elements can be inserted in any order.

  While deleting, largest item is deleted first.

Operations performed on Priority Queue

- Insert an element.
- Delete an element.
- Display the contents of Queue.

Function to insert an item at the correct place in priority queue.

```
void insert_item(int item,int q[], int *r)
{
      int j;
      if(q_full(*r))
      {
            printf("Queue is full\n");
            return;
      }
      j = *r; /* Compare from this initial point */
      /* Find the appropriate position */
      while(j >= 0 && item < q[j])
      {
            q[j+1] = q[j]; /*Move the item at q[j] to its next position */
            j--;
      }
       /* Insert an item at the appropriate position */
      q[j+1]=item;
      /* Update the rear item */
      *r = *r + 1;

}
```

Summary

- "***Data Structures*** "deals with the study of how the data is organized in the memory, how efficiently the data can be retrieved and manipulated, and the possible ways in which different data items are logically related.
- A ***Stack*** is defined as a special type of data structure where items are inserted from one end called ***top*** of stack and items are deleted from the same end.
- A ***queue*** is defined as a special type of data structure where the elements are inserted from one end and elements are deleted from the other end.
- A ***Deque*** is a special type of data structure in which insertions and deletions will be done either at the front end or at the rear end of the queue.
- In ***circular queue***, the elements of a given queue can be stored efficiently in an array so as to **"wrap around"** so that end of the queue is followed by the front of the queue.
- The priority queue is a special type of data structure in which items can be inserted or deleted based on the priority.