1. Fundamental Notations

1.1 Problem solving concept top down and bottom up design

Top Down Program Design

Top down program design is an approach to program design that starts with the general concept and repeatedly breaks it down into its component parts. In other words, it starts with the abstract and continually subdivides it until it reaches the specific. Consider creating the prime factorization of a number like 1540. The steps involved might look like:

- 1540
- 2 x 770
- 2 x 2 x 385
- 2 x 2 x 5 x 77
- 2 x 2 x 5 x 7 x 11

Top down program design works the same way. We start with the overall objective and wind up with a series of steps needed to accomplish it.

Bottom Up Program Design

Bottom up program design works in the exact opposite way. It starts with the component parts and repeatedly combines them to achieve the general concept. In other words, it starts with the specific and continually combines it until it reaches the abstract. For example, consider the factorization from the previous section. For bottom up design the steps involved might look like:

- 2 x 2 x 5 x 7 x 11
- 2 x 2 x 5 x 77
- 2 x 2 x 385
- 2 x 770
- 1540

DIFFERENCE BETWEEN TOP DOWN APPROACH AND BOTTOM UP APPROACH

- Structure/procedure oriented programming languages like C programming language follows top down approach. Whereas object oriented programming languages like C++ and Java programming language follows bottom up approach.

- Top down approach begins with high level design and ends with low level design or development. Whereas, bottom up approach begins with low level design or development and ends with high level design.
- In top down approach, main() function is written first and all sub functions are called from main function. Then, sub functions are written based on the requirement. Whereas, in bottom up

approach, code is developed for modules and then these modules are integrated with main() function.

- Now-a-days, both approaches are combined together and followed in modern software design

1.1.1 Structured Programming

Structured programming is a logical programming method that is considered a precursor to object-oriented programming (OOP). Structured programming facilitates program understanding and modification and has a top-down design approach, where a system is divided into compositional subsystems.

Structured programming is a procedural programming subset that reduces the need for goto statements. Modular programming is another example of structural programming, where a program is divided into interactive modules.

1.2 Concept of datatypes, variables & constants

1.2.1 Data type

A data type, in programming, is a classification that specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error. A string, for example, is a data type that is used to classify text and an integer is a data type used to classify whole numbers.

Data Type	Used for	Example
String	Alphanumeric characters	hello world, Alice, Bob123
Integer	Whole numbers	7, 12, 999
Float (floating point)	Number with a decimal point	3.15, 9.06, 00.13
Character	Encoding text numerically	97 (in ASCII, 97 is a lower case 'a')
Boolean	Representing logical values	TRUE, FALSE

1.2.2 Variable

In programming, a variable is a value that can change, depending on conditions or on information passed to the program. Typically, a program consists of instructions that tell the computer what to do and data that the program uses when it is running. The data consists of *constants* or fixed values that never change and variable values (which are usually initialized to "0" or some default value because the actual values will be supplied by a program's user). Usually, both constants and variables are defined as certain datatypes. Each data type prescribes and limits the form of the data. Examples of data types include: an integer expressed as a decimal number, or a string of text characters, usually limited in length.

1.2.3 Constants

In programming, a constant is a value that never changes. The other type of values that programs use is variables, symbols that can represent different values throughout the course of a program.
A constant can be a number, like 25 or 3.6.

A character, like a or $, a character string, like "this is a string".

Constants are also used in spreadsheet applications to place non-changing values in cells. In contrast, a spreadsheet formula can produce a different value each time the spreadsheet is opened or changed.

1.3 Concepts of Pointer variables

A Pointer in C language is a variable which holds the address of another variable of same data type.

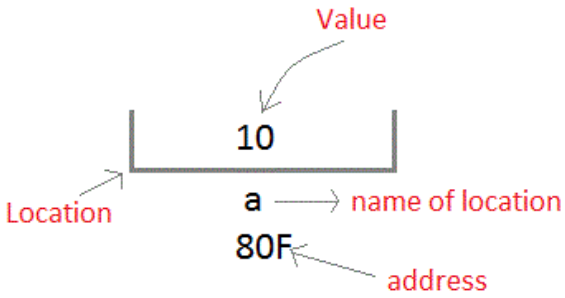
Pointers are used to access memory and manipulate the address.

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language.

Whenever a **variable** is declared in a program, system allocates a location i.e. an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

Let us assume that system has allocated memory location 80F for a variable a.

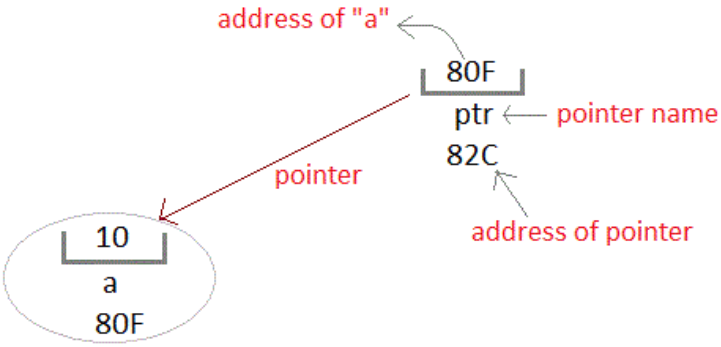
```
int a = 10;
```



We can access the value 10 either by using the variable name a or by using its address 80F.

The question is how we can access a variable using it's address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**.

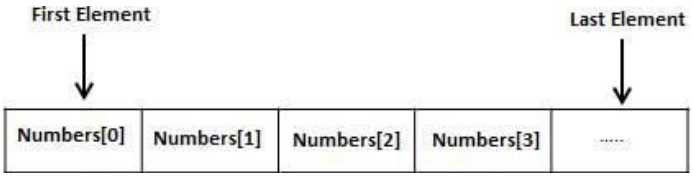
A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.



2. ARRAY

2.1 Concept of Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.Array makes it easier to calculate the position of each element by simply adding an offset to a base value.All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



2.1.1 Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

2.1.2 Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

2.1.3 Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
double salary = balance[9];
```

The above statement will take the 10^{th} element from the array and assign the value to salary variable.

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration −

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional integer array −

```
int threedim[5][10][4];
```

2.2 Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows −

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

2.2.1 Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
{0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

2.2.2 Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array

2.3 Operations on Arrays

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	Default Value
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

2.3.1 Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array –

Algorithm

Let **Array** be a linear unordered array of **MAX** elements.

Example

Result

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the K^{th} position of LA –

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

main() {

   int LA[] = {1,3,5,7,8};

   int item = 10, k = 3, n = 5;

   int i = 0, j = n;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {

      printf("LA[%d] = %d \n", i, LA[i]);

   }
```

```
   n = n + 1;

   while( j >= k) {
      LA[j+1] = LA[j];
      j = j - 1;
   }

   LA[k] = item;

   printf("The array elements after insertion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

For other variations of array insertion operation click here

2.3.2 Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the K^{th} position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>


void main() {

   int LA[] = {1,3,5,7,8};

   int k = 3, n = 5;

   int i, j;


   printf("The original array elements are :\n");


   for(i = 0; i<n; i++) {

      printf("LA[%d] = %d \n", i, LA[i]);

   }


   j = k;


   while( j < n) {

      LA[j-1] = LA[j];
```

```
      j = j + 1;
   }

   n = n -1;

   printf("The array elements after deletion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

2.3.3 Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
```

```
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>


void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   while( j < n){
      if( LA[j] == item ) {
         break;
      }

      j = j + 1;
   }

   printf("Found element %d at position %d\n", item, j+1);
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

2.3.4 Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the K^{th} position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>


void main() {

   int LA[] = {1,3,5,7,8};

   int k = 3, n = 5, item = 10;

   int i, j;


   printf("The original array elements are :\n");


   for(i = 0; i<n; i++) {

      printf("LA[%d] = %d \n", i, LA[i]);

   }
```

```
   LA[k-1] = item;

   printf("The array elements after updation :\n");

         for(i = 0; i<n; i++) {

      printf("LA[%d] = %d \n", i, LA[i]);

   }

}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

3. Linked Lists

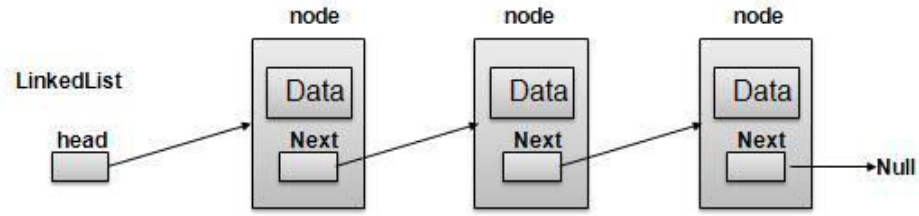
3.1Introduction to Linked Lists

Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.

Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.

Linked Lists are used to create trees and graphs.

3.2 Linked List Representation



As per above shown illustration, following are the important points to be considered.

- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

3.3 operations on Linked List

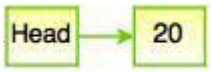
1. Create
2. Insert
3. Delete
4. Traverse
5. Search
6. Concatenation
7. Display

3.3.1 Create

- Create operation is used to create constituent node when required.
- In create operation, memory must be allocated for one node and assigned to head as follows.

Creating first node

```
head = (node*) malloc (sizeof(node));
head -> data = 20;
head -> next = NULL;
```



3.3.2 Insert

- Insert operation is used to insert a new node in the linked list.
- Suppose, we insert a node B(New Node), between A(Left Node) and C(Right Node), it is

represented as:
point B.next to B

```
NewNode.next -> RightNode;
```

We can insert an element using three cases:

i. At the beginning of the list
ii. At a certain position (Middle)
iii. At the end

Inserting an element

node* nextnode = malloc(sizeof(node));

nextnode -> data = 22;

nextnode -> next = NULL;

head -> next = nextnode;

The above figure represents the example of create operation, where the next element (i.e 22) is added to the next node by using insert operation.

i. At the beginning of the list

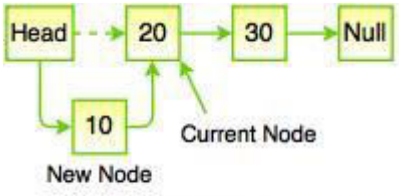


Fig. Inserting a node at the beginning of the list

New node becomes the new head of the linked list because it is always added before the head of the given linked list.

ii. At certain position (Middle)

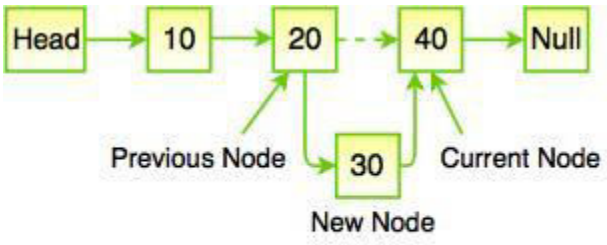


Fig. Inserting a node at middle of the list

While inserting a node in middle of a linked list, it requires to find the current node. The dashed line represents the old node which points to new node.

iii. At the end

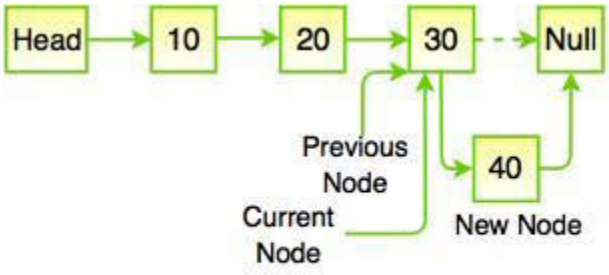


Fig. Inserting a node at the end of the list

While inserting a node at the end of the list, it is achieved by comparing the element values.

3.3.3 Delete

- Delete operation is used to delete node from the list.
- This operation is more than one step process.

We can delete an element using three cases:

i. From the beginning of the list
ii. From the middle
iii. From the end

Deleting the element

```
int delete (node** head, node* n);    // Delete the node n if exists.
```

i. From the beginning of the list

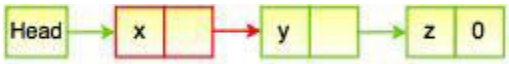
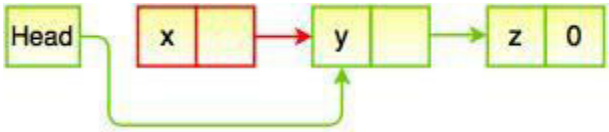


Fig. Deleting a node from the beginning

When deleting the node from the beginning of the list then there is no relinking of nodes to be performed; it means that the first node has no preceding node. The above figure shows the removing node with x. However, it requires to fix the pointer to the beginning of the list which is shown in the figure below:



ii. From the middle

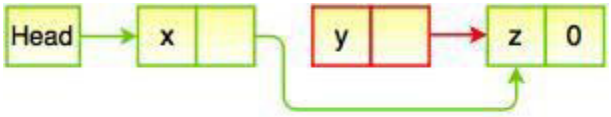


Fig. Deleting a node from the middle

Deleting a node from the middle requires the preceding node to skip over the node being removed.

The above figure shows the removal of node with x. It means that there is a need refer to the node before we can remove it.

iii. From the end

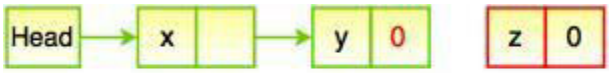


Fig. Deleting a node from the end

Deleting a node from the end requires that the preceding node becomes the new end of the list that points to nothing after it. The above figure shows removing the node with z.

3.3.4 Traverse

- Traverse operations is a process of examining all the nodes of linked list from the end to the other end.
- In traverse operation, recursive function is used to traverse a linked list in a reverse order.

The following code snippet represents traversing a node in a linked list:

```
void traverse(node *head)
{
  if(head != NULL)
  {
    traverse (head -> next);
    printf("%d", head -> data);
  }
}
```

3.4 Applications of Linked List

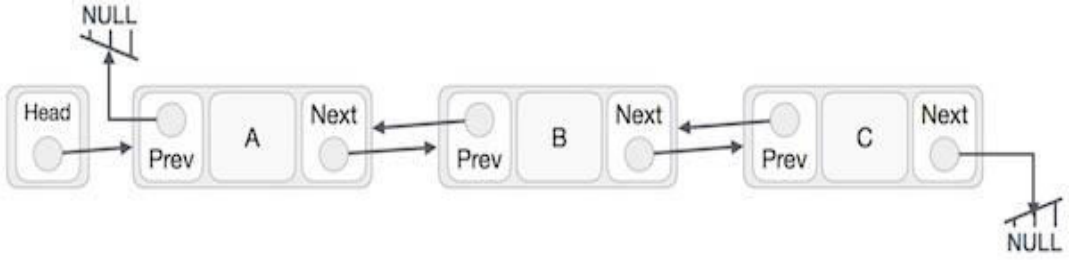
- Linked Lists can be used to implement Stacks , Queues.
- Linked Lists can also be used to implement Graphs. (Adjacency list representation of Graph).
- Implementing Hash Tables :- Each Bucket of the hash table can itself be a linked list. (Open chain hashing).
- Undo functionality in Photoshop or Word . Linked list of states.
- A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term.
- However, for any polynomial operation , such as addition or multiplication of polynomials , linked list representation is more easier to deal with.
- Linked lists are useful for dynamic memory allocation.
- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running.
- All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed

3.5 Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

3.5.1 Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

3.6 Operations on Doubly Linked List

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner.

3.6.1 Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

Example

```
//insert link at the first location

void insertFirst(int key, int data) {


   //create a link

   struct node *link = (struct node*) malloc(sizeof(struct node));

   link->key = key;

   link->data = data;
```

```
   if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //update first prev link
      head->prev = link;
   }

   //point it to old first link
   link->next = head;

   //point first to new first link
   head = link;
}
```

3.6.2 Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

Example

```
//delete first item
struct node* deleteFirst() {

   //save reference to first link
   struct node *tempLink = head;

   //if only one link
   if(head->next == NULL) {
      last = NULL;
```

```
   } else {
      head->next->prev = NULL;
   }

   head = head->next;

   //return the deleted link
   return tempLink;
}
```

4. STACKS, QUEUES AND RECURSION

4.1 Introduction to stacks

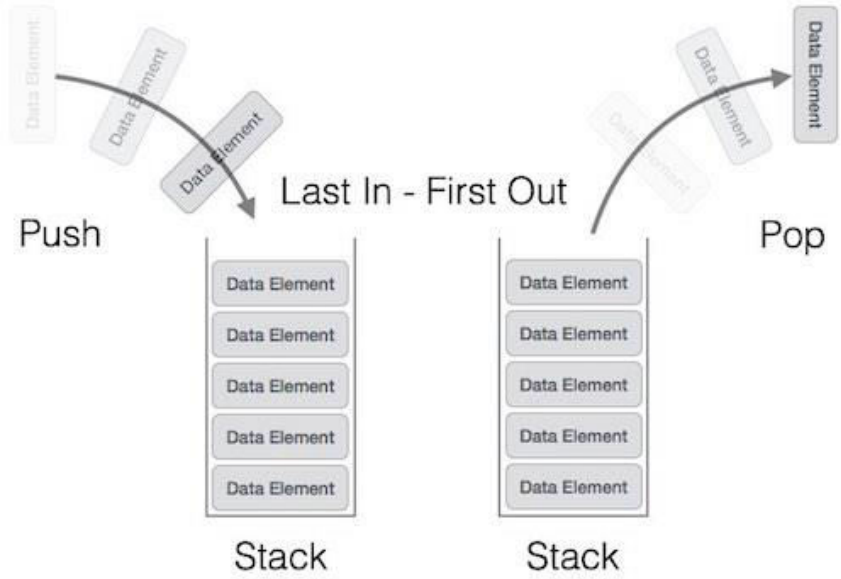
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

4.2 Representation of stacks

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

4.3 Implementation of stacks

A stack can be easily implemented either through an array or a linked list. What identifies the data structure as a stack in either case is not the implementation but the interface: the user is only allowed to pop or push items onto the array or linked list, with few other helper operations. The following will demonstrate both implementations, using pseudocode.

4.3.1 Array

An array can be used to implement a (bounded) stack, as follows. The first element (usually at the zero offset) is the bottom, resulting in `array[0]` being the first element pushed onto the stack and the last element popped off. The program must keep track of the size (length) of the stack, using a variable *top* that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted (assuming a zero-based index convention). Thus, the stack itself can be effectively implemented as a three-element structure:

```
structure stack:
    maxsize : integer
    top : integer
    items : array of item
procedure initialize(stk : stack, size : integer):
    stk.items ← new array of size items, initially empty
    stk.maxsize ← size
    stk.top ← 0
```

The *push* operation adds an element and increments the *top* index, after checking for overflow:

```
procedure push(stk : stack, x : item):
    if stk.top = stk.maxsize:
        report overflow error
    else:
        stk.items[stk.top] ← x
        stk.top ← stk.top + 1
```

Similarly, *pop* decrements the *top* index after checking for underflow, and returns the item that was previously the top one:

```
procedure pop(stk : stack):
    if stk.top = 0:
        report underflow error
    else:
        stk.top ← stk.top - 1
        r ← stk.items[stk.top]
        return r
```

Using a dynamic array, it is possible to implement a stack that can grow or shrink as much as needed. The size of the stack is simply the size of the dynamic array, which is a very efficient implementation of a stack since adding items to or removing items from the end of a dynamic array requires amortized O(1) time.

4.3.2 Linked list

Another option for implementing stacks is to use a singly linked list. A stack is then a pointer to the "head" of the list, with perhaps a counter to keep track of the size of the list:

```
structure frame:
    data : item
    next : frame or nil
structure stack:
    head : frame or nil
    size : integer
procedure initialize(stk : stack):
    stk.head ← nil
    stk.size ← 0
```

Pushing and popping items happens at the head of the list; overflow is not possible in this implementation (unless memory is exhausted):

```
procedure push(stk : stack, x : item):
    newhead ← new frame
    newhead.data ← x
    newhead.next ← stk.head
    stk.head ← newhead
    stk.size ← stk.size + 1
procedure pop(stk : stack):
    if stk.head = nil:
        report underflow error
```

```
    r ← stk.head.data
    stk.head ← stk.head.next
    stk.size ← stk.size - 1
    return r
```

4.4 Applications of Stack

In a stack, only limited operations are performed because it is restricted data structure. The elements are deleted from the stack in the reverse order.

Following are the applications of stack:

1. Expression Evaluation
2. Expression Conversion
   i. Infix to Postfix
   ii. Infix to Prefix
   iii. Postfix to Infix
   iv. Prefix to Infix
3. Backtracking
4. Memory Management

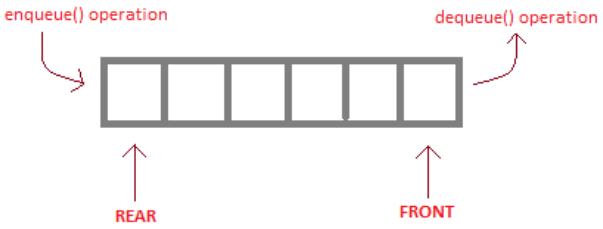
4.4 Introduction to Queues

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

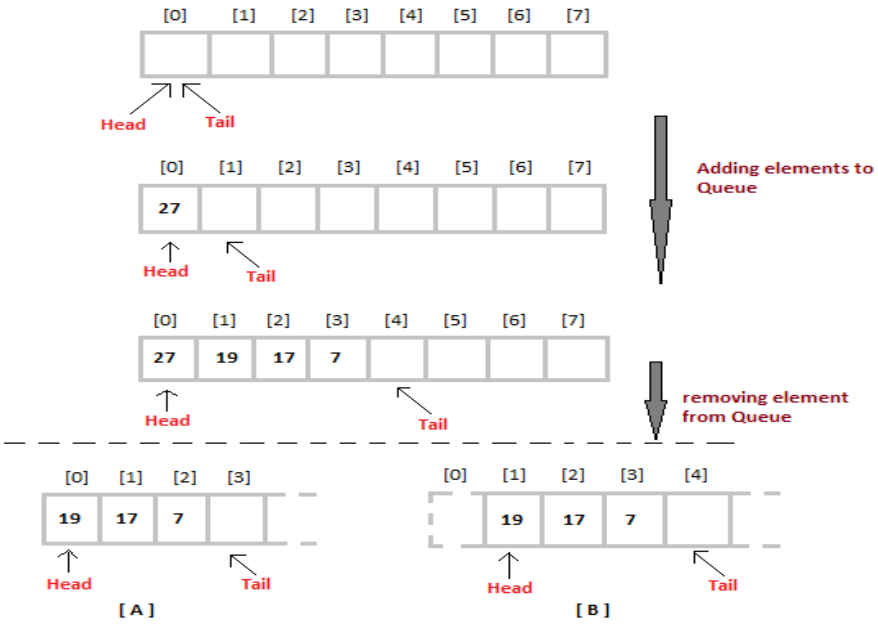
QUEUE DATA STRUCTURE

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

4.6 Implementation of Queues

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

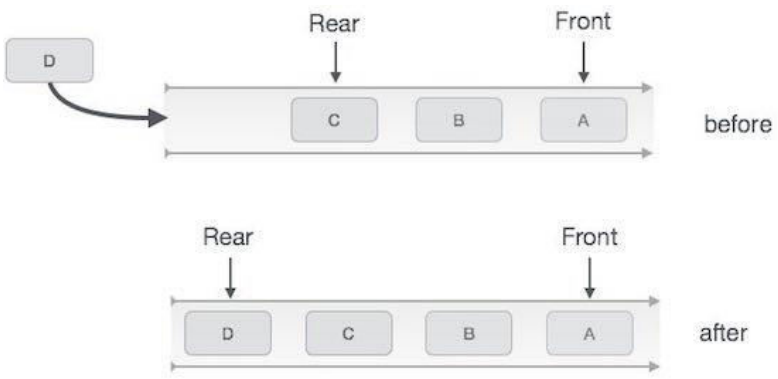
In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

4.6.1 Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)


   if queue is full
      return overflow
   endif


   rear ← rear + 1
   queue[rear] ← data
   return true


end procedure
```

Implementation of enqueue() in C programming language –

Example

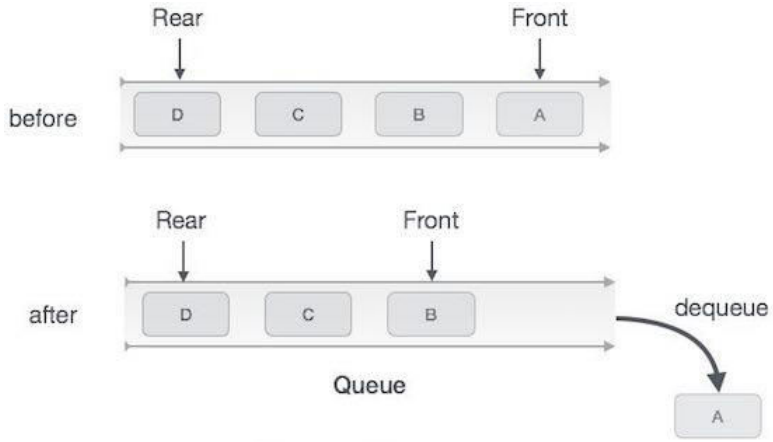
```
int enqueue(int data)
   if(isfull())
      return 0;
```

```
   rear = rear + 1;

   queue[rear] = data;


   return 1;

end procedure
```

4.6.2 Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue


   if queue is empty

      return underflow
```

```
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

4.7 Circular Queue

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as **Ring Buffer.**
- It is an abstract data type.

- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

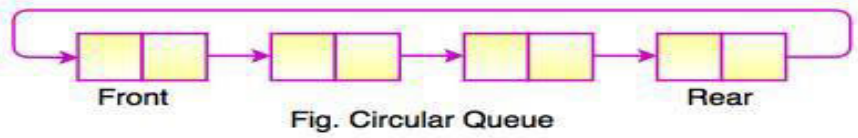


Fig. Circular Queue

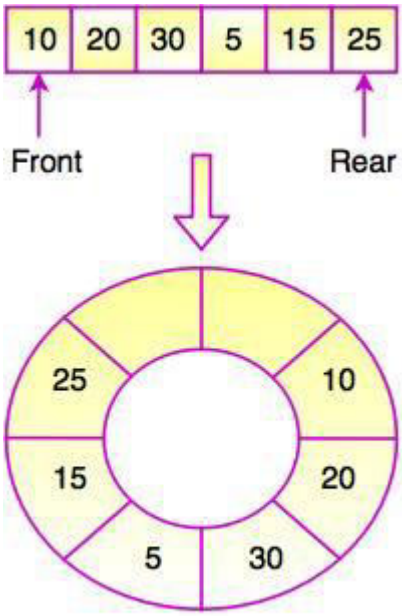


Fig. Circular Queue

The above figure shows the structure of circular queue. It stores an element in a circular way and performs the operations according to its FIFO structure.

Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
  Steps:
  1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

Steps:

1. Check whether queue is Empty means check (front==-1).
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

- ## Example: Program for Circular Queue

```
#include<stdio.h>
#include<cstdlib>
#define max 6
int q[10],front=0,rear=-1;
int main()
{
   int ch;
   void insert();
   void delet();
   void display();

   printf("\nCircular Queue Operations\n");
   printf("1.Insert\n2.Delete\n3.Display\n4.Exit\n");
   while(1)
   {
      printf("Enter Your Choice:");
      scanf("%d",&ch);
      switch(ch)
      {
      case 1: insert();
         break;
      case 2: delet();
         break;
      case 3:display();
         break;
      case 4: exit(0);
      default:printf("Invalid option\n");
```

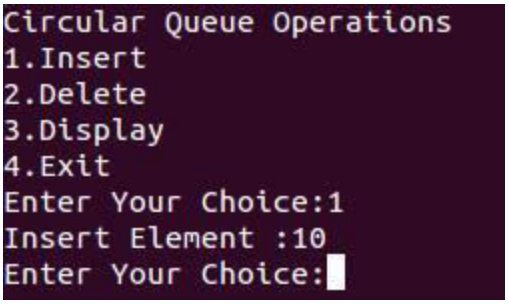
```
        }
    }
}
void insert()
{
    int x;
    if((front==0&&rear==max-1)||(front>0&&rear==front-1))
        printf("Queue is Overflow\n");
    else
    {
        printf("Insert Element :");
        scanf("%d",&x);
        if(rear==max-1&&front>0)
        {
            rear=0;
            q[rear]=x;
        }
        else
        {
            if((front==0&&rear==-1)||(rear!=front-1))
             q[++rear]=x;
        }
    }
}
void  delet()
{
    int a;
    if((front==0)&&(rear==-1))
    {
    printf("Queue is Underflow\n");
    exit(0);
    }
    if(front==rear)
```

```
    {
        a=q[front];
        rear=-1;
        front=0;
    }
    else
        if(front==max-1)
        {
            a=q[front];
            front=0;
        }
        else a=q[front++];
        printf("Deleted Element is : %d\n",a);
}
void display()
{
    int i,j;
    if(front==0&&rear==-1)
    {
        printf("Queue is Underflow\n");
        exit(0);
    }
    if(front>rear)
    {
        for(i=0;i<=rear;i++)
            printf("\t%d",q[i]);
        for(j=front;j<=max-1;j++)
            printf("\n \t%d",q[j]);
        printf("\nRear is at %d\n",q[rear]);
        printf("\nFront is at %d\n",q[front]);
    }
    else
    {
```

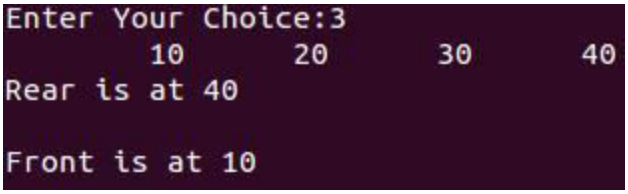
```
        for(i=front;i<=rear;i++)
        {
            printf("\t%d",q[i]);
        }
        printf("\nRear is at %d\n",q[rear]);
        printf("\nFront is at %d\n",q[front]);
    }
    printf("\n");
}
```

- **Output:**

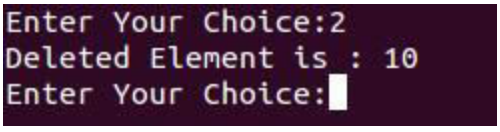
1.Insert



2. Display



3. Delete



4.8 De-queues

In Double Ended Queue, insert and delete operation can be occur at both ends that is front and rear of the queue.

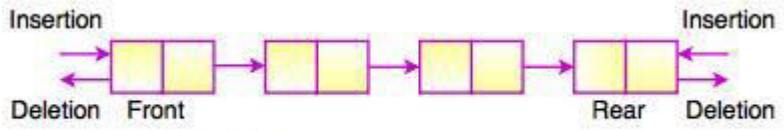


Fig. Double Ended Queue (Dequeue)

Example: Program for Double Ended Queue (Dequeue)

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 30

typedef struct dequeue
{
   int data[MAX];
   int rear,front;
}dequeue;

void initialize(dequeue *p);
int empty(dequeue *p);
int full(dequeue *p);
void enqueueR(dequeue *p,int x);
void enqueueF(dequeue *p,int x);
int dequeueF(dequeue *p);
int dequeueR(dequeue *p);
void print(dequeue *p);

void main()
{
   int i,x,op,n;
```

```
    dequeue q;
    initialize(&q);
    do
    {
     printf("\n1.Create\n2.Insert(Rear)\n3.Insert(Front)\n4.Delete(Rear)\n5.Delet(Front)
");
        printf("\n6.Print\n7.Exit\n\nEnter your choice:");
        scanf("%d",&op);

        switch(op)
        {
           case 1: printf("\nEnter number of elements:");
                   scanf("%d",&n);
                   initialize(&q);
                   printf("\nEnter the data:");
                   for(i=0;i<n;i++)
                   {
                      scanf("%d",&x);
                      if(full(&q))
                      {
                         printf("\nQueue is Full!!");
                         exit(0);
                      }
                      enqueueR(&q,x);
                   }
                   break;

           case 2: printf("\nInsert Element : ");
                   scanf("%d",&x);
                   if(full(&q))
                   {
                      printf("\nQueue is Full!!");
                      exit(0);
```

```
        }
        enqueueR(&q,x);
        break;

case 3: printf("\nInsert Element :");
        scanf("%d",&x);
        if(full(&q))
        {
           printf("\nQueue is Full!!");
           exit(0);
        }
        enqueueF(&q,x);
        break;

case 4: if(empty(&q))
        {
           printf("\nQueue is Empty!!");
           exit(0);
        }
        x=dequeueR(&q);
        printf("\n Deleted Element is %d\n",x);
        break;

case 5: if(empty(&q))
        {
           printf("\nQueue is Empty!!");
           exit(0);
        }
        x=dequeueF(&q);
        printf("\nDeleted Element is %d\n",x);
        break;

case 6: print(&q);
```

```
                break;

          default: break;
       }
   }while(op!=7);
}
void initialize(dequeue *P)
{
   P->rear=-1;
   P->front=-1;
}
int empty(dequeue *P)
{
   if(P->rear==-1)
      return(1);

   return(0);
}
int full(dequeue *P)
{
   if((P->rear+1)%MAX==P->front)
      return(1);

   return(0);
}
void enqueueR(dequeue *P,int x)
{
   if(empty(P))
   {
      P->rear=0;
      P->front=0;
      P->data[0]=x;
   }
```

```
    else
    {
        P->rear=(P->rear+1)%MAX;
        P->data[P->rear]=x;
    }
}
void enqueueF(dequeue *P,int x)
{
    if(empty(P))
    {
        P->rear=0;
        P->front=0;
        P->data[0]=x;
    }
    else
    {
        P->front=(P->front-1+MAX)%MAX;
        P->data[P->front]=x;
    }
}
int dequeueF(dequeue *P)
{
    int x;
    x=P->data[P->front];
    if(P->rear==P->front)    //delete the last element
        initialize(P);
    else
        P->front=(P->front+1)%MAX;
    return(x);
}
int dequeueR(dequeue *P)
{
    int x;
```

```
    x=P->data[P->rear];
    if(P->rear==P->front)
        initialize(P);
    else
        P->rear=(P->rear-1+MAX)%MAX;
    return(x);
}
void print(dequeue *P)
{
    if(empty(P))
    {
        printf("\nQueue is empty!!");
        exit(0);
    }
    int i;
    i=P->front;
    while(i!=P->rear)
    {
        printf("\n%d",P->data[i]);
        i=(i+1)%MAX;
    }
    printf("\n%d\n",P->data[P->rear]);
}
```

Output:

1. Create

```
1.Create
2.Insert(Rear)
3.Insert(Front)
4.Delete(Rear)
5.Delete(Front)
6.Print
7.Exit

Enter your choice:1

Enter number of elements:5

Enter the data:40
20
66
30
10
```

2. Insert (Front)

```
1.Create
2.Insert(Rear)
3.Insert(Front)
4.Delete(Rear)
5.Delete(Front)
6.Print
7.Exit

Enter your choice:3

Insert Element :40
```

3. Insert (Rear)

```
1.Create
2.Insert(Rear)
3.Insert(Front)
4.Delete(Rear)
5.Delete(Front)
6.Print
7.Exit

Enter your choice:2

Insert Element : 50
```

4. Display (Insert)

```
1.Create
2.Insert(Rear)
3.Insert(Front)
4.Delete(Rear)
5.Delete(Front)
6.Print
7.Exit

Enter your choice:6

40
40
20
66
30
10
50
```

5. Delete (Rear)

```
1.Create
2.Insert(Rear)
3.Insert(Front)
4.Delete(Rear)
5.Delete(Front)
6.Print
7.Exit

Enter your choice:4

 Deleted Element is 50
```

6. Delete (Front)

```
1.Create
2.Insert(Rear)
3.Insert(Front)
4.Delete(Rear)
5.Delete(Front)
6.Print
7.Exit

Enter your choice:5

Deleted Element is 40
```

7. Display (Delete)

```
1.Create
2.Insert(Rear)
3.Insert(Front)
4.Delete(Rear)
5.Delete(Front)
6.Print
7.Exit

Enter your choice:6

40
20
66
30
10
```

4.9 Applications of Queues

- Queue is useful in CPU scheduling, Disk Scheduling. When multiple processes require CPU at the same time, various CPU scheduling algorithms are used which are implemented using Queue data structure.
- When data is transferred asynchronously between two processes.Queue is used for synchronization. Examples : IO Buffers, pipes, file IO, etc.
- In print spooling, documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate. Spooling also lets you place a number of print jobs on a queue instead of waiting for each one to finish before specifying the next one.
- Breadth First search in a Graph .It is an algorithm for traversing or searching graph data structures. It starts at some arbitrary node of a graph and explores the neighbor nodes first, before moving to the next level neighbors.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

4.10 Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc

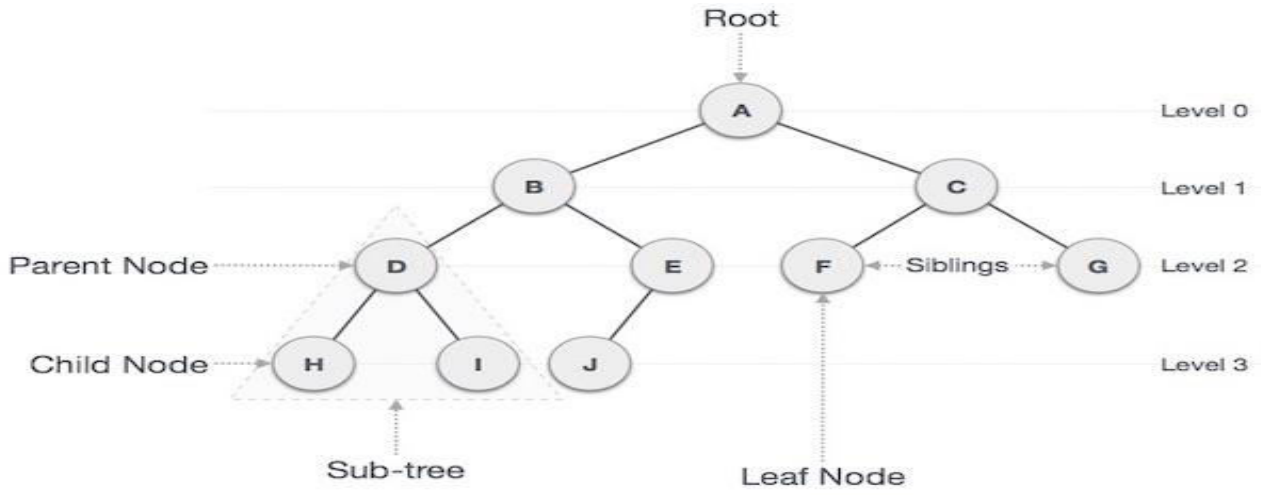
```
int fact(int n)
{
    if (n < = 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```

5. TREES

5.1 Concept of trees

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



5.1.1 Important Terms

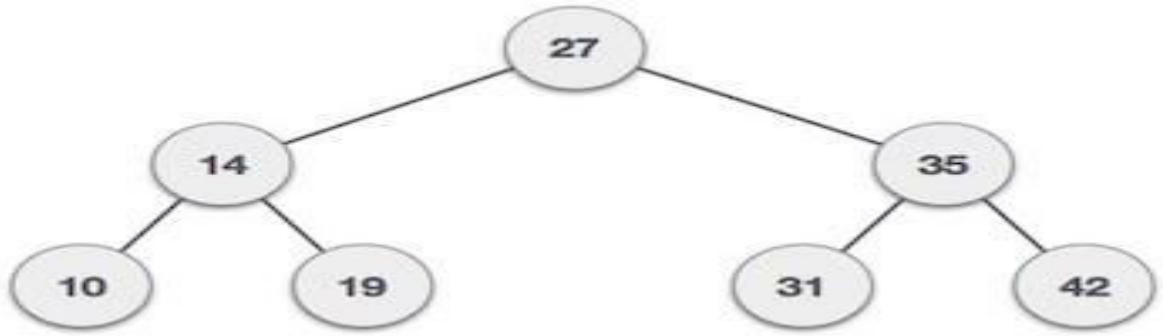
Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.

- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

5.2 Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

5.3 Traversing BST

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

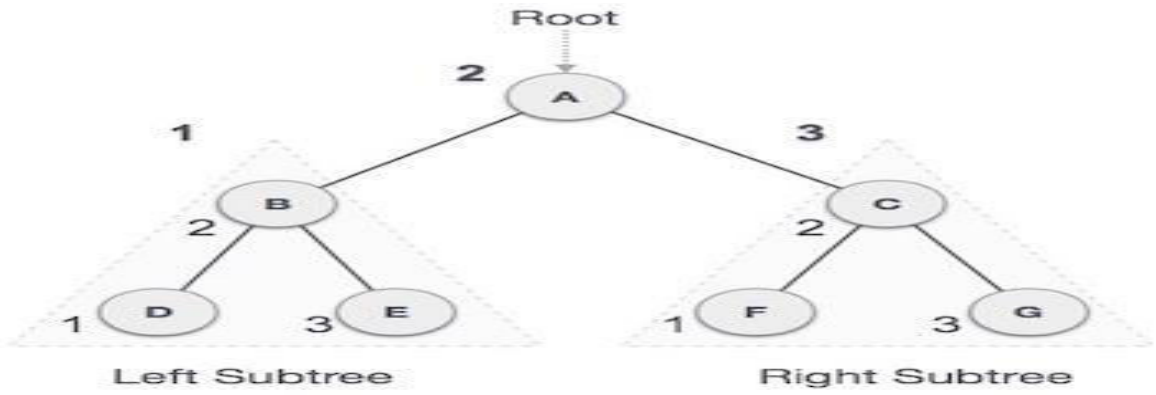
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

5.3.1 In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

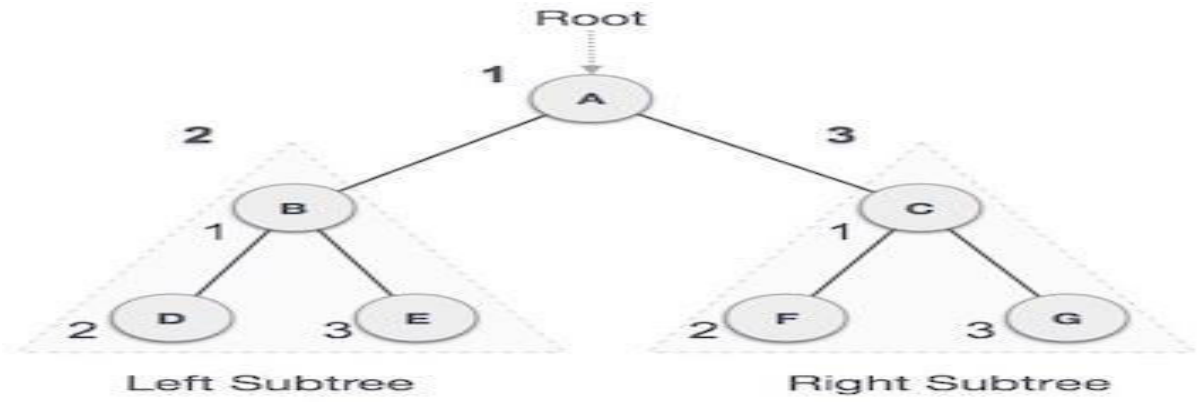
Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
```

```
Step 3 – Recursively traverse right subtree.
```

5.3.2 Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

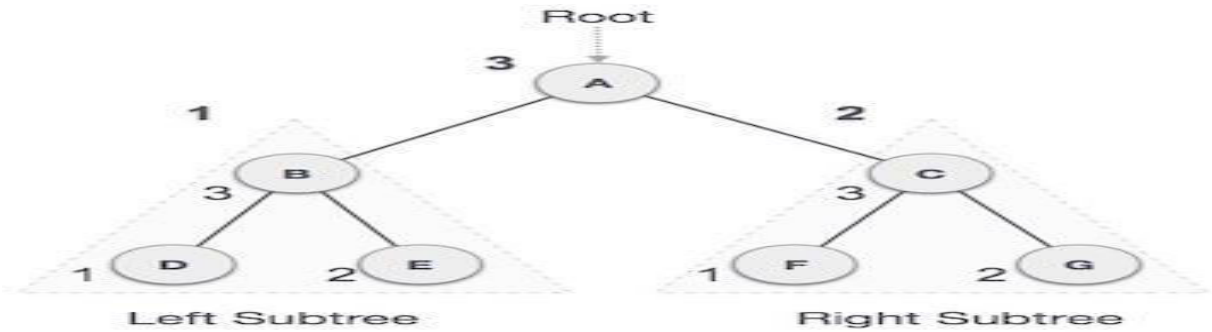
$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

```
Until all nodes are traversed –
Step 1 – Visit root node.
Step 2 – Recursively traverse left subtree.
Step 3 – Recursively traverse right subtree.
```

5.3.3 Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```

5.4 BST Operations

5.4.1 Insert Operation in BST

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
  then create root node
return


If root exists then
  compare the data with node.data


  while until insertion position is located


    If data is greater than node.data
      goto right subtree
    else
```

```
      goto left subtree

   endwhile

   insert data

end If
```

Implementation

The implementation of insert function should look like this −

```
void insert(int data) {

   struct node *tempNode = (struct node*) malloc(sizeof(struct node));

   struct node *current;

   struct node *parent;


   tempNode->data = data;

   tempNode->leftChild = NULL;

   tempNode->rightChild = NULL;


   //if tree is empty, create root node

   if(root == NULL) {

      root = tempNode;

   } else {

      current = root;

      parent  = NULL;


      while(1) {

         parent = current;
```

```
         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }

         //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

5.4.2 Search Operation in BST

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node
   endwhile

   return data not found

end if
```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");
```

```
   while(current->data != data) {

      if(current != NULL)
      printf("%d ",current->data);

      //go to left tree

      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }

      return current;
   }
}
```

5.5 Introduction to Heap

The heap is a binary tree, meaning at the most, each parent has two children. There are two types of heaps: the **max** and **min** heap. The root node of a max heap is the highest value in the heap, whereas a min heap has the minimum value allocated to the root node.

The heap data structure is a very useful data structure that every programmer should know well. The heap data structure is used behind the scenes to perform the heap sort. Understanding how the heap works will allow programmers to make wiser decisions when programming in an environment where memory management is crucial.

6. SORTING AND SEARCHING

6.1 Introduction to Sorting and Searching

6.1.1 Sorting

Sorting is nothing but arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

6.1.2 Searching

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- It can be done on internal data structure or on external data structure.

6.2 Search Algorithm

To search an element in a given array, it can be done in following ways:

1. Sequential Search
2. Binary Search

6.2.1 Linear Search

- Sequential search is also called as **Linear Search.**
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

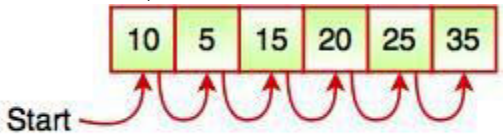


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

The following code snippet shows the sequential search operation:

```
function searchValue(value, target)
{
    for (var i = 0; i < value.length; i++)
    {
        if (value[i] == target)
        {
            return i;
        }
    }
    return -1;
}
searchValue([10, 5, 15, 20, 25, 35] , 25);   // Call the function with array and number to be searched
```

Example: Program for Linear Search

```
#include <stdio.h>
int main()
{
```

```
  int arr[50], search, cnt, num;

  printf("Enter the number of elements in array\n");
  scanf("%d",&num);

  printf("Enter %d integer(s)\n", num);

  for (cnt = 0; cnt < num; cnt++)
  scanf("%d", &arr[cnt]);

  printf("Enter the number to search\n");
  scanf("%d", &search);

  for (cnt = 0; cnt < num; cnt++)
  {
    if (arr[cnt] == search)     /* if required element found */
    {
      printf("%d is present at location %d.\n", search, cnt+1);
      break;
    }
  }
  if (cnt == num)
    printf("%d is not present in array.\n", search);

  return 0;
}
```

Output:

```
Enter the number of elements in array
5
Enter 5 integer(s)
20
30
6
46
78
Enter the number to search
46
46 is present at location 4.
```

6.2.2 Binary Search

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of O(log n).
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.

- The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

For example, if searching an element 25 in the 7-element array, following figure shows how binary search works:

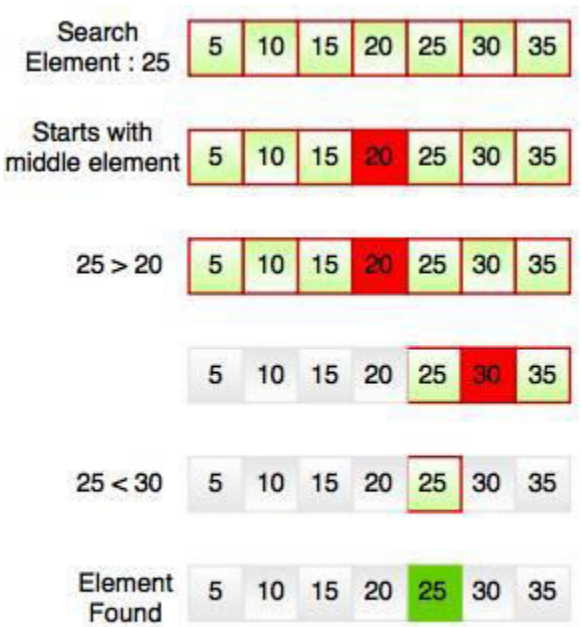


Fig. Working Structure of Binary Search

Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

Example: Program for Binary Search

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

void main()
{
  int f, l, m, size, i, sElement, list[50]; //int f, l ,m : First, Last, Middle
  clrscr();

  printf("Enter the size of the list: ");
  scanf("%d",&size);

  printf("Enter %d integer values : \n", size);

  for (i = 0; i < size; i++)
```

```
    scanf("%d",&list[i]);

  printf("Enter value to be search: ");
  scanf("%d", &sElement);

  f = 0;
  l = size - 1;
  m = (f+l)/2;

  while (f <= l) {
    if (list[m] < sElement)
      f = m + 1;
    else if (list[m] == sElement) {
      printf("Element found at index %d.\n",m);
      break;
    }
    else
      l = m - 1;
    m = (f + l)/2;
  }
  if (f > l)
    printf("Element Not found in the list.");
  getch();
}
```

Output:

```
Enter the size of the list: 5
Enter 5 integer values :
10
30
20
50
40
Enter value to be search: 20
Element found at index 2.
```

6.3 Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

6.3.1 Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

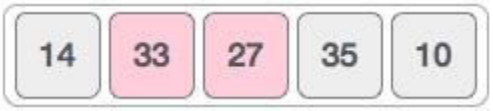
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

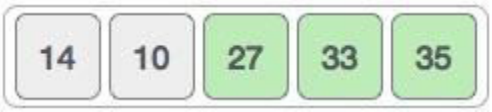
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

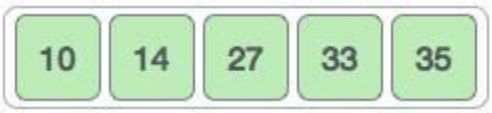
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap**function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

6.3.2 Insertion Sort

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value

Step 6 − Repeat until list is sorted

6.3.3 Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −

Now, let us learn some programming aspects of selection sort.

Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted