Basic terms related to data structure.

Data Definition

Data Definition defines a particular data with the following characteristics.

- Atomic − Definition should define a single concept.
- Traceable − Definition should be able to be mapped to some data element.
- Accurate − Definition should be unambiguous.
- Clear and Concise − Definition should be understandable.

Data Object

Data Object represents an object having a data.

Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

- List
- Array

- Stack
- Queue

Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

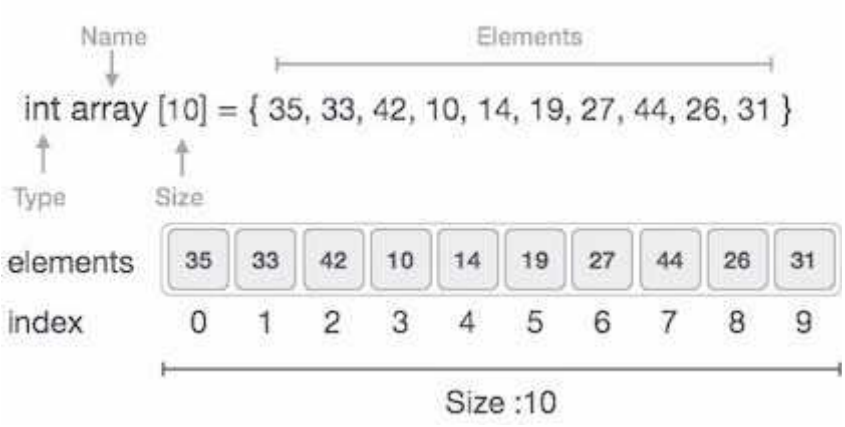
Data Structures and Algorithms - Arrays

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- Element − Each item stored in an array is called an element.
- Index − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let’s take C array declaration.



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

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

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

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

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>

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
```

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>

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
```

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>

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

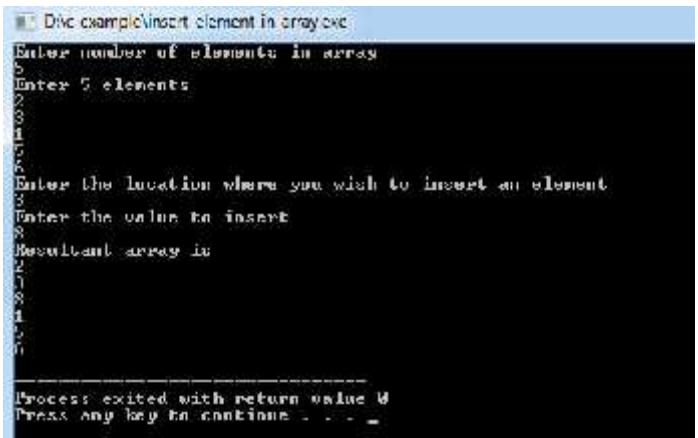
```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Program logic to insert an element at desired position.

For example consider an array n[10] having four elements:
n[0] = 1, n[1] = 2, n[2] = 3 and n[3] = 4
And suppose you want to insert a new value 60 at first position of array. i.e. n[0] = 60, so we have to move elements one step below so after insertion
n[1] = 1 which was n[0] initially, n[2] = 2, n[3] = 3 and n[4] = 4.

```c
#include <stdio.h>

int main()
{
    int array[50], position, c, n, value;

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

    printf("Enter %d elements\n", n);

    for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

    printf("Please enter the location where you want to insert an new element\n");
    scanf("%d", &position);

    printf("Please enter the value\n");
    scanf("%d", &value);

    for (c = n - 1; c >= position - 1; c--)
           array[c+1] = array[c];

    array[position-1] = value;

    printf("Resultant array is\n");

    for (c = 0; c <= n; c++)
           printf("%d\n", array[c]);

    return 0;
}
```

Program Output:

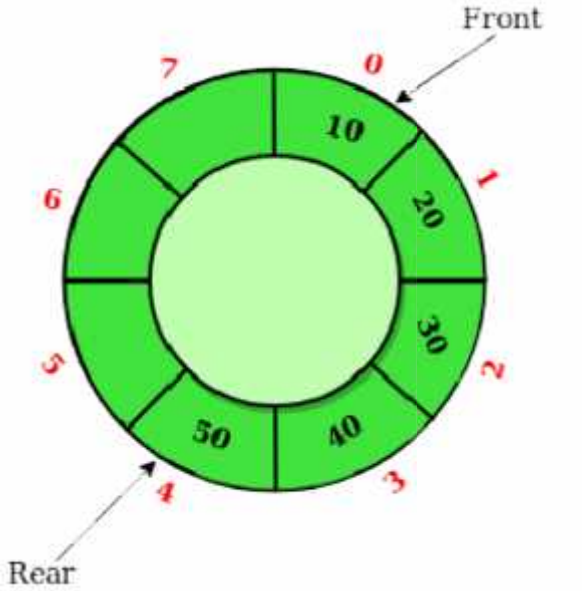


```c
#include <stdio.h>

int main()
{
   int array[100], position, c, n;

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

   printf("Enter %d elements\n", n);

   for ( c = 0 ; c < n ; c++ )
      scanf("%d", &array[c]);

   printf("Enter the location where you wish to delete
element\n");
   scanf("%d", &position);

   if ( position >= n+1 )
      printf("Deletion not possible.\n");
   else
   {
      for ( c = position - 1 ; c < n - 1 ; c++ )
         array[c] = array[c+1];

      printf("Resultant array is\n");

      for( c = 0 ; c < n - 1 ; c++ )
         printf("%d\n", array[c]);
   }
```

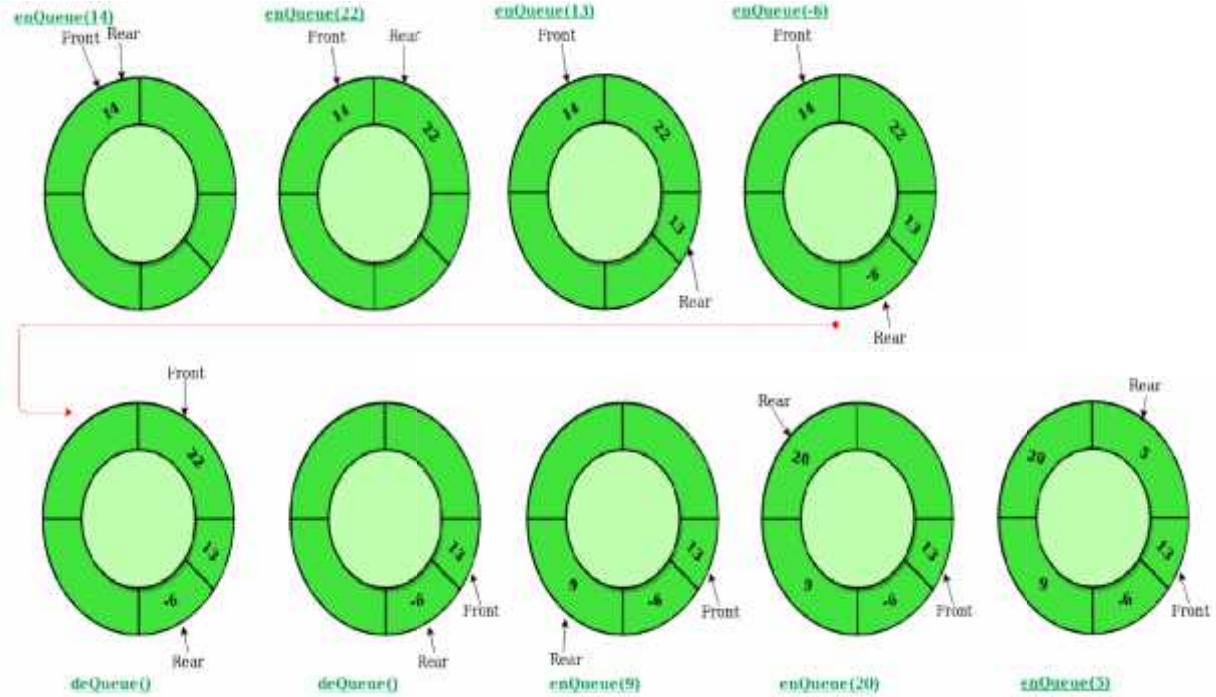
```
    return 0;
}
```

Circular Queue | Set 1 (Introduction and Array Implementation)

Prerequisite – Queues
Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



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.

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

```
// C or C++ program for insertion and
// deletion in Circular Queue
#include<bits/stdc++.h>
using namespace std;

struct Queue
{
    // Initialize front and rear
    int rear, front;

    // Circular Queue
    int size;
    int *arr;

    Queue(int s)
    {
       front = rear = -1;
       size = s;
       arr = new int[s];
    }

    void enQueue(int value);
    int deQueue();
    void displayQueue();
};


/* Function to create Circular queue */
void Queue::enQueue(int value)
{
    if ((front == 0 && rear == size-1) ||
            (rear == front-1))
    {
        printf("\nQueue is Full");
        return;
    }

    else if (front == -1) /* Insert First Element */
    {
        front = rear = 0;
        arr[rear] = value;
    }

    else if (rear == size-1 && front != 0)
    {
        rear = 0;
```

```
        arr[rear] = value;
    }

    else
    {
        rear++;
        arr[rear] = value;
    }
}

// Function to delete element from Circular Queue
int Queue::deQueue()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return INT_MIN;
    }

    int data = arr[front];
    arr[front] = -1;
    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else if (front == size-1)
        front = 0;
    else
        front++;

    return data;
}

// Function displaying the elements
// of Circular Queue
void Queue::displayQueue()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return;
    }
    printf("\nElements in Circular Queue are: ");
    if (rear >= front)
    {
        for (int i = front; i <= rear; i++)
            printf("%d ",arr[i]);
    }
    else
    {
        for (int i = front; i < size; i++)
            printf("%d ", arr[i]);

        for (int i = 0; i <= rear; i++)
            printf("%d ", arr[i]);
    }
}

/* Driver of the program */
int main()
{
    Queue q(5);
```

```
    // Inserting elements in Circular Queue
    q.enQueue(14);
    q.enQueue(22);
    q.enQueue(13);
    q.enQueue(-6);

    // Display elements present in Circular Queue
    q.displayQueue();

    // Deleting elements from Circular Queue
    printf("\nDeleted value = %d", q.deQueue());
    printf("\nDeleted value = %d", q.deQueue());

    q.displayQueue();

    q.enQueue(9);
    q.enQueue(20);
    q.enQueue(5);

    q.displayQueue();

    q.enQueue(20);
    return 0;
}
```

Run on IDE
Output:

```
Elements in Circular Queue are: 14 22 13 -6

Deleted value = 14

Deleted value = 22

Elements in Circular Queue are: 13 -6

Elements in Circular Queue are: 13 -6 9 20 5

Queue is Full
```

Data Structure and Algorithms - Linked List

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of 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.

- LinkedList − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

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

- Linked List contains a 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.

Types of Linked List

Following are the various types of linked list.

- Simple Linked List − Item navigation is forward only.
- Doubly Linked List − Items can be navigated forward and backward.
- Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

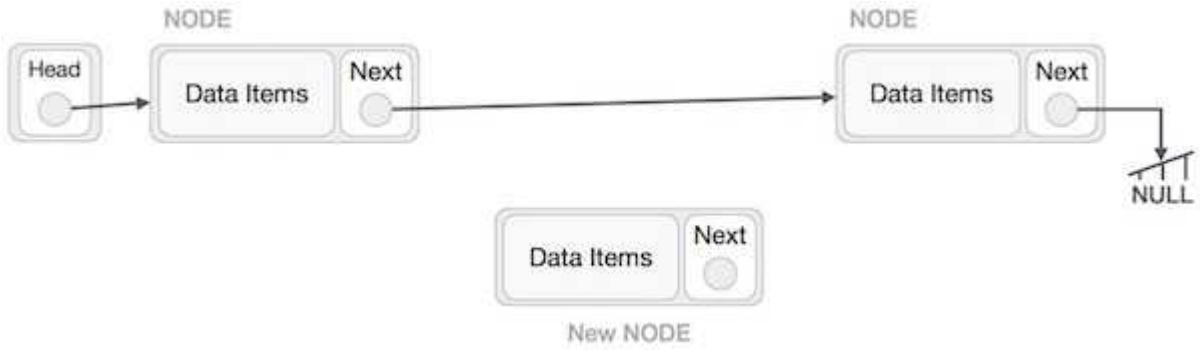
Basic Operations

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.
- Display − Displays the complete list.
- Search − Searches an element using the given key.
- Delete − Deletes an element using the given key.

Insertion Operation

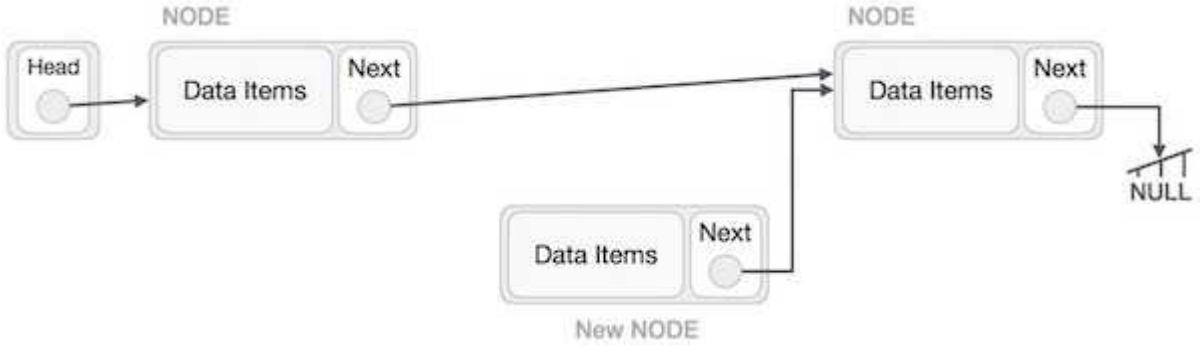
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C −

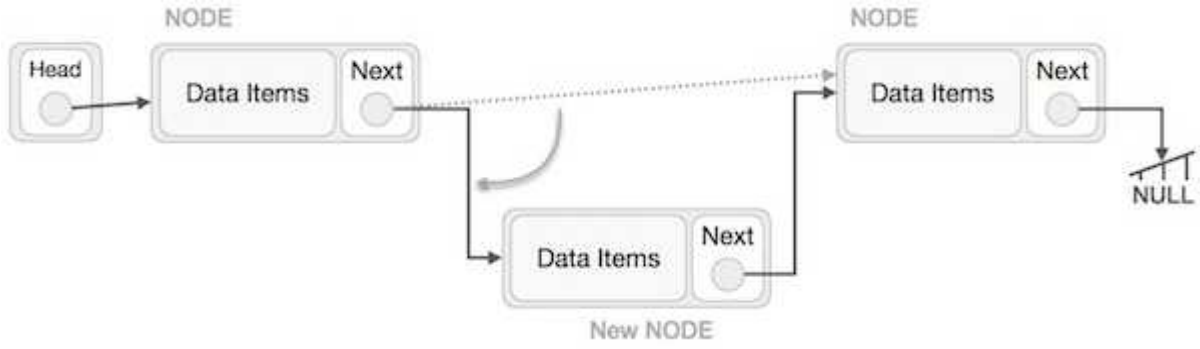
```
NewNode.next -> RightNode;
```

It should look like this −



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```



This will put the new node in the middle of the two. The new list should look like this −

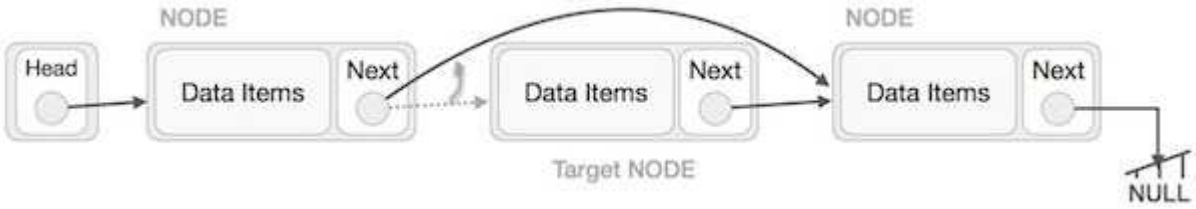
Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

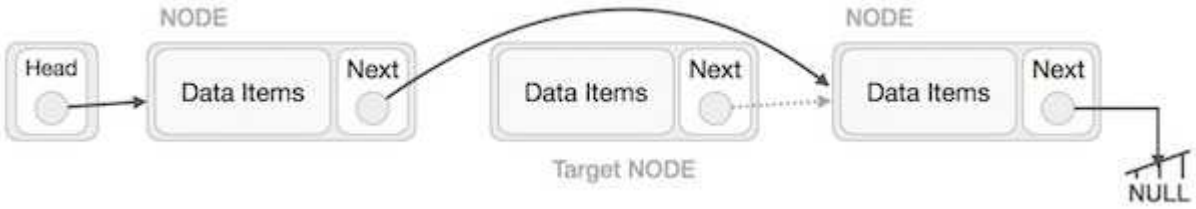
The left (previous) node of the target node now should point to the next node of the target node −

```
LeftNode.next -> TargetNode.next;
```

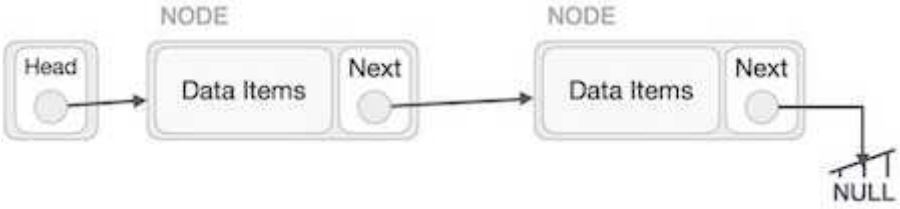


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

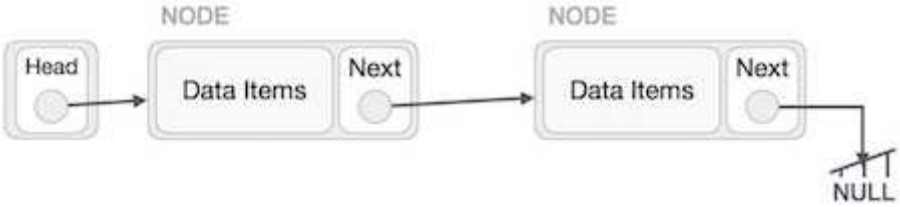


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

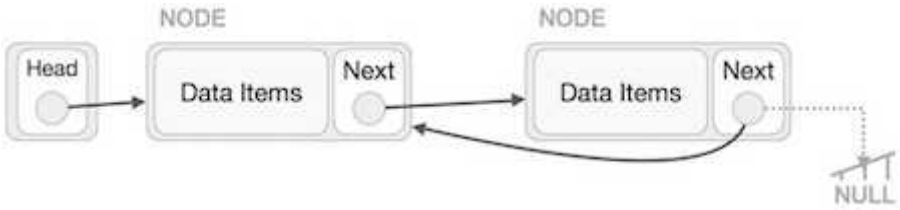


Reverse Operation

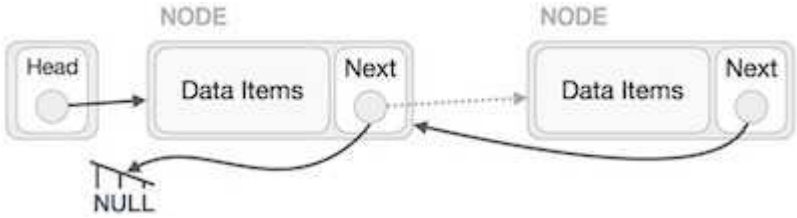
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



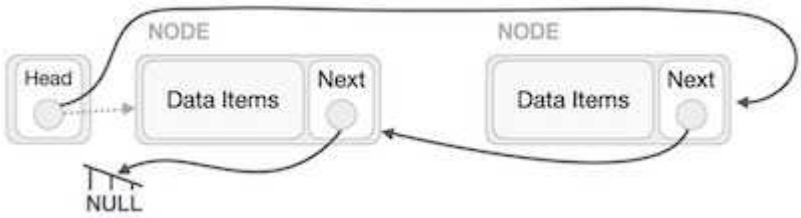
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



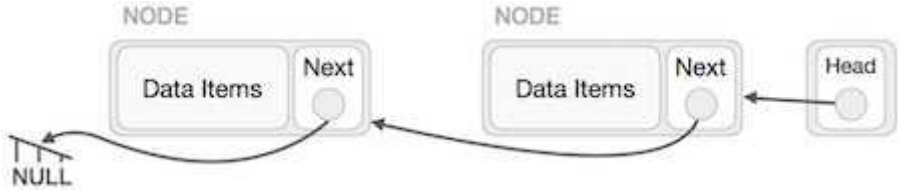
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



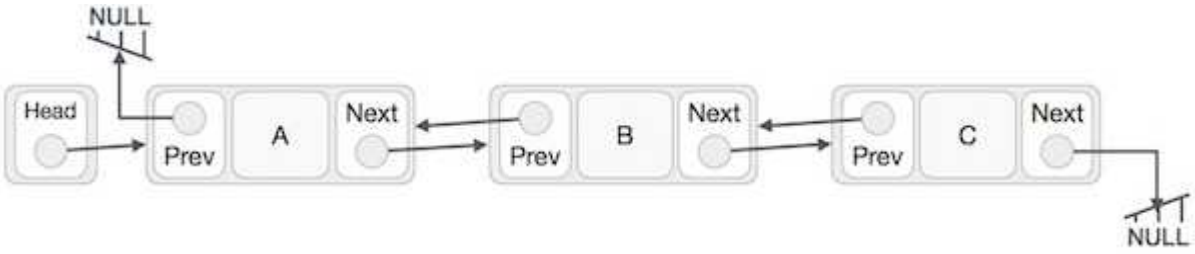
The linked list is now reversed.

Data Structure - 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.

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.

Basic Operations

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.

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;
}
```

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;

}
```

Insertion at the End of an Operation

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

Example

```
//insert link at the last location

void insertLast(int key, int data) {


   //create a link

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

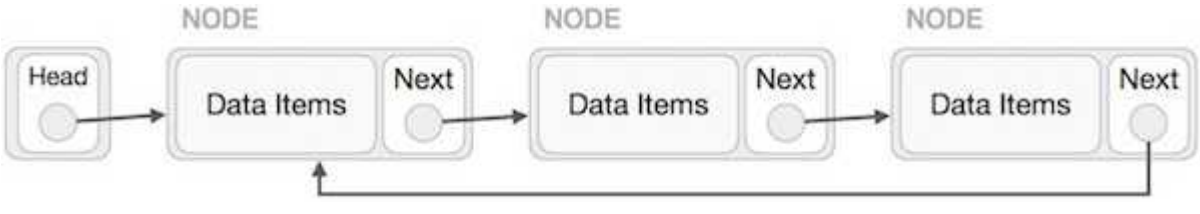
```
   link->key = key;

   link->data = data;


   if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //make link a new last link
      last->next = link;


      //mark old last node as prev of new link
      link->prev = last;
   }


   //point last to new last node
   last = link;
}
```

Data Structure - Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

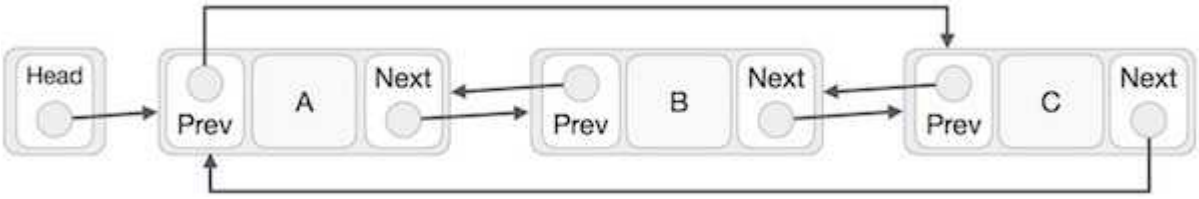
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



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

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- `insert` – Inserts an element at the start of the list.
- `delete` – Deletes an element from the start of the list.
- `display` – Displays the list.

Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single 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()) {

      head = link;

      head->next = head;
```

```
   } else {
      //point it to old first node
      link->next = head;

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

Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
//delete first item
struct node * deleteFirst() {
   //save reference to first link
   struct node *tempLink = head;

   if(head->next == head) {
      head = NULL;
      return tempLink;
   }

   //mark next to first link as first
   head = head->next;

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

Display List Operation

Following code demonstrates the display list operation in a circular linked list.

```
//display the list
```

```
void printList() {
   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   if(head != NULL) {
      while(ptr->next != ptr) {
         printf("(%d,%d) ",ptr->key,ptr->data);
         ptr = ptr->next;
      }
   }

   printf(" ]");
}
```

Data Structure and Algorithms - Stack

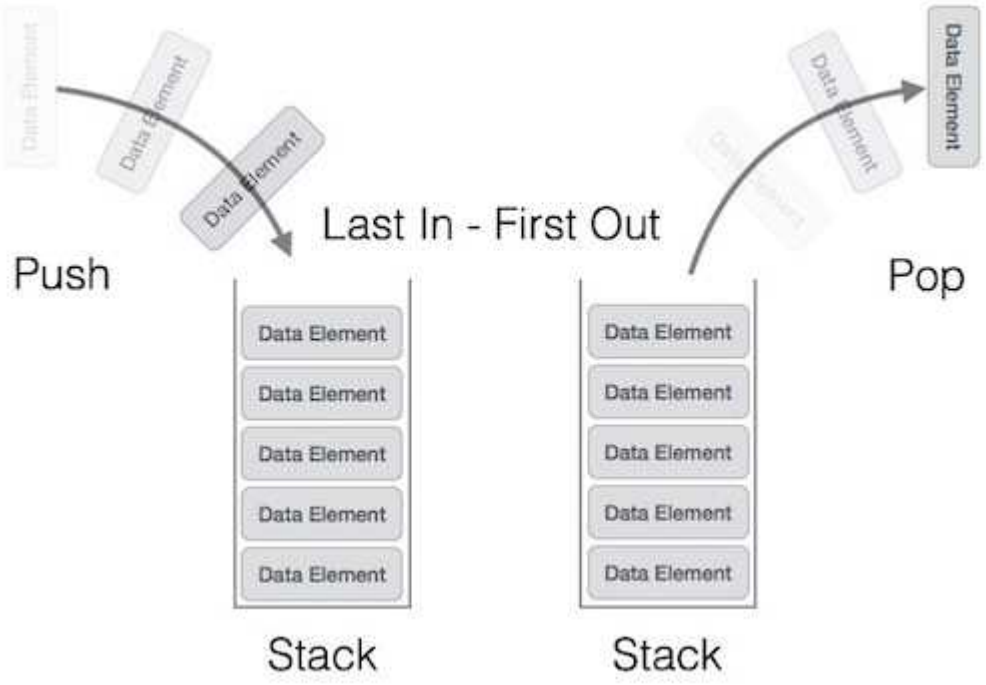
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

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.

Stack Representation

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.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- `push()` − Pushing (storing) an element on the stack.
- `pop()` − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- `peek()` − get the top data element of the stack, without removing it.
- `isFull()` − check if stack is full.
- `isEmpty()` − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named `top`. The `top`pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek

   return stack[top]

end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {

   return stack[top];

}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull


   if top equals to MAXSIZE

      return true

   else

      return false

   endif


end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {

   if(top == MAXSIZE)

      return true;

   else

      return false;

}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty


   if top less than 1
      return true
   else
      return false
   endif


end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

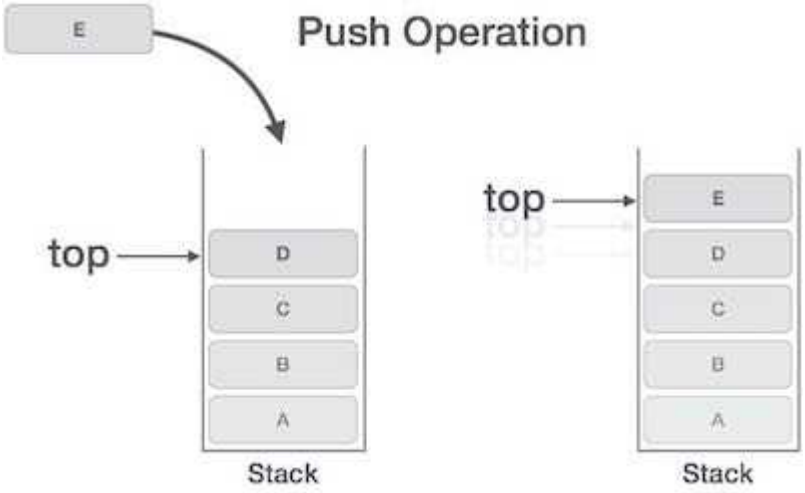
Example

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- Step 1 − Checks if the stack is full.
- Step 2 − If the stack is full, produces an error and exit.
- Step 3 − If the stack is not full, increments top to point next empty space.
- Step 4 − Adds data element to the stack location, where top is pointing.
- Step 5 − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data


   if stack is full
      return null
   endif


   top ← top + 1


   stack[top] ← data


end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

Example

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
```

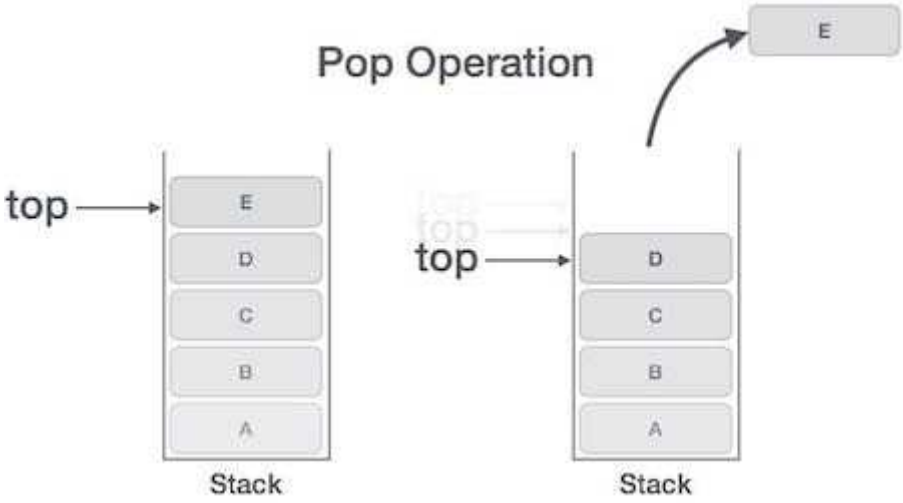
```
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- Step 1 − Checks if the stack is empty.
- Step 2 − If the stack is empty, produces an error and exit.
- Step 3 − If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 − Decreases the value of top by 1.
- Step 5 − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif
```

```
   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

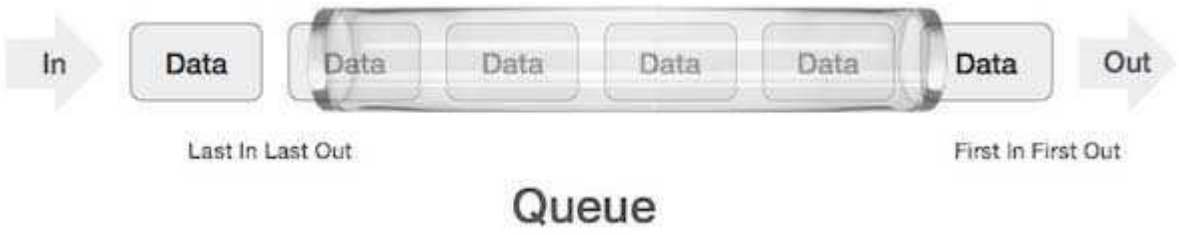
Data Structure and Algorithms - Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

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.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Let’s first learn about supportive functions of a queue −

peek()

This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek

   return queue[front]

end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {

   return queue[front];

}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull


   if rear equals to MAXSIZE

      return true

   else

      return false

   endif


end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {

   if(rear == MAXSIZE - 1)

      return true;

   else

      return false;

}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty


   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif


end procedure
```

If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

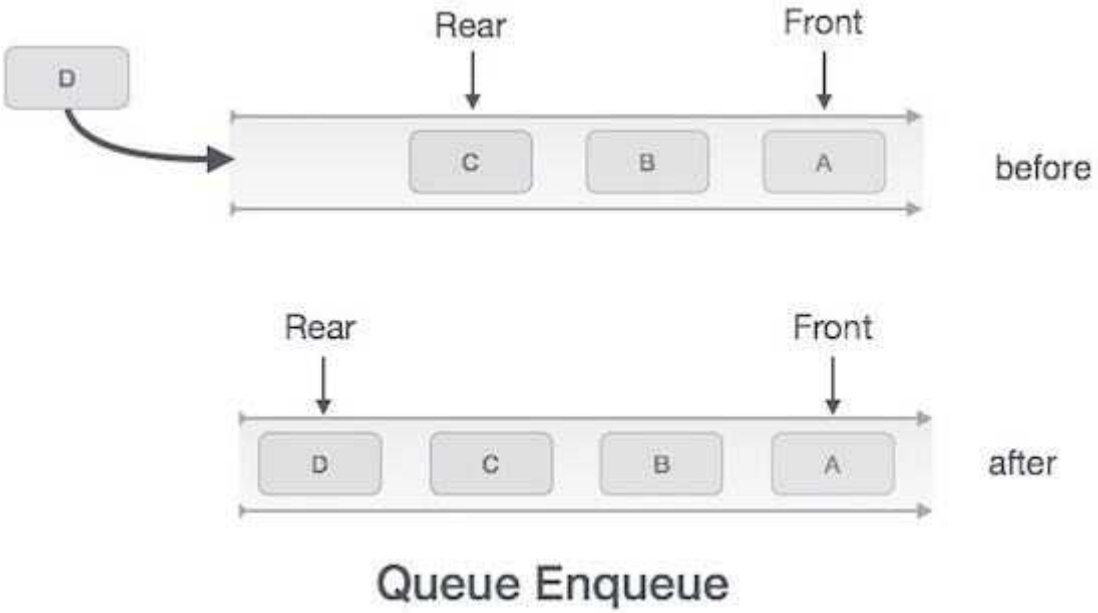
Enqueue Operation

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

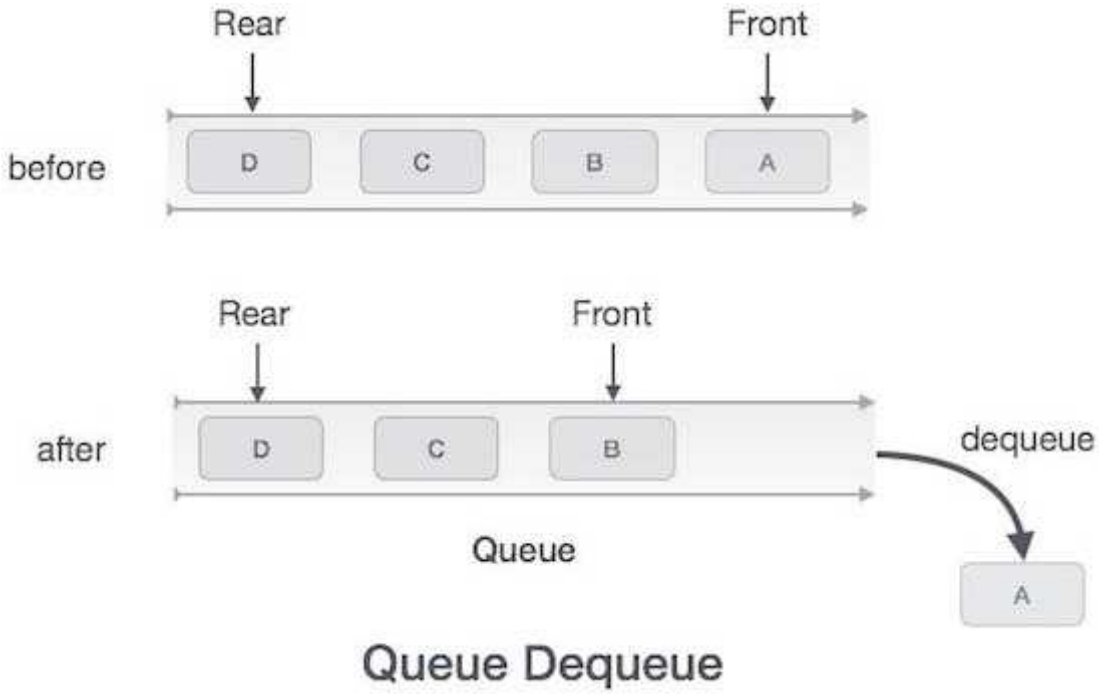
```
int enqueue(int data)

   if(isfull())

      return 0;
```

```
   rear = rear + 1;

   queue[rear] = data;


   return 1;

end procedure
```

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
```

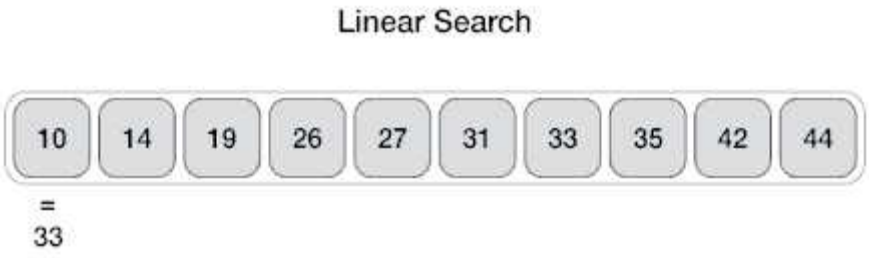
Implementation of dequeue() in C programming language −

Example

```
int dequeue() {

   if(isempty())
      return 0;

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

   return data;
}
```

Data Structure and Algorithms Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Pseudocode

```
procedure linear_search (list, value)


   for each item in the list


      if match item == value


         return the item's location


      end if


   end for


end procedure
```

Data Structure and Algorithms Binary Search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

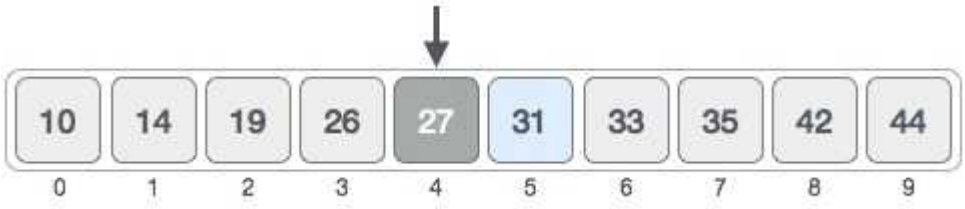
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

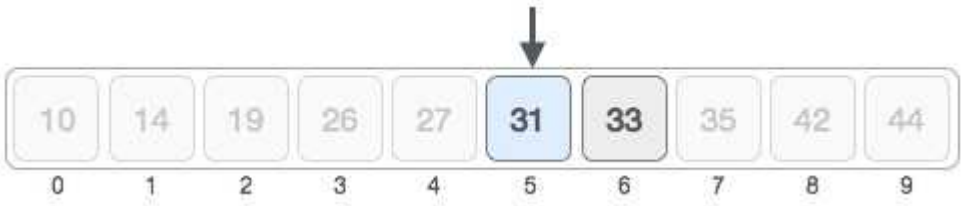
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search

   A ← sorted array

   n ← size of array

   x ← value to be searched


   Set lowerBound = 1
```

```
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint

   end while

end procedure
```

Data Structure - Sorting Techniques

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

- Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

- Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

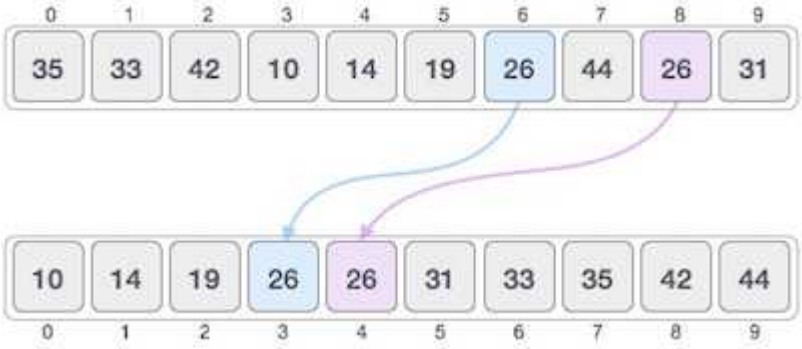
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.

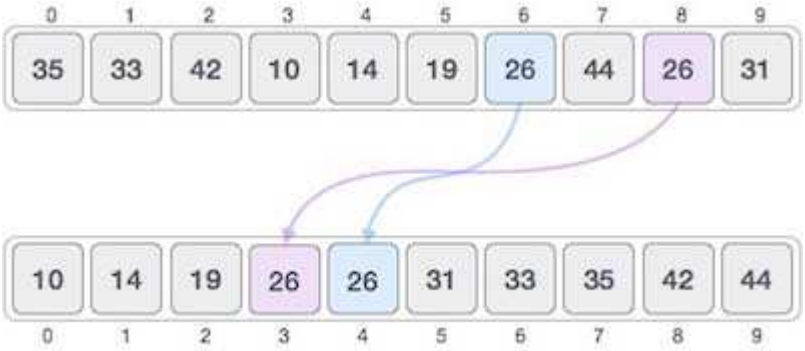
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Adaptive and Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Important Terms

Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them −

Increasing Order

A sequence of values is said to be in increasing order, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

Decreasing Order

A sequence of values is said to be in decreasing order, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

Non-Increasing Order

A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.

Non-Decreasing Order

A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

Data Structure - Bubble Sort Algorithm

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.

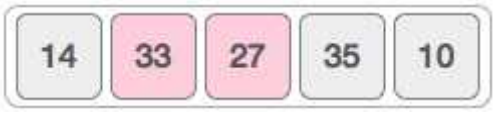
How Bubble Sort Works?

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.

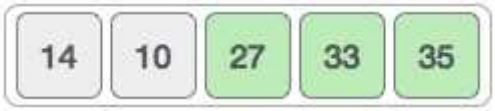
Then we move to the next two values, 35 and 10.

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 swapfunction 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
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order.

This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable `swapped` which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows −

```
procedure bubbleSort( list : array of items )


   loop = list.count;


   for i = 0 to loop-1 do:

      swapped = false


      for j = 0 to loop-1 do:


         /* compare the adjacent elements */

         if list[j] > list[j+1] then

            /* swap them */

            swap( list[j], list[j+1] )

            swapped = true

         end if


      end for


      /*if no number was swapped that means

      array is sorted now, break the loop.*/


      if(not swapped) then

         break

      end if


   end for


end procedure return list
```

Data Structure and Algorithms 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.

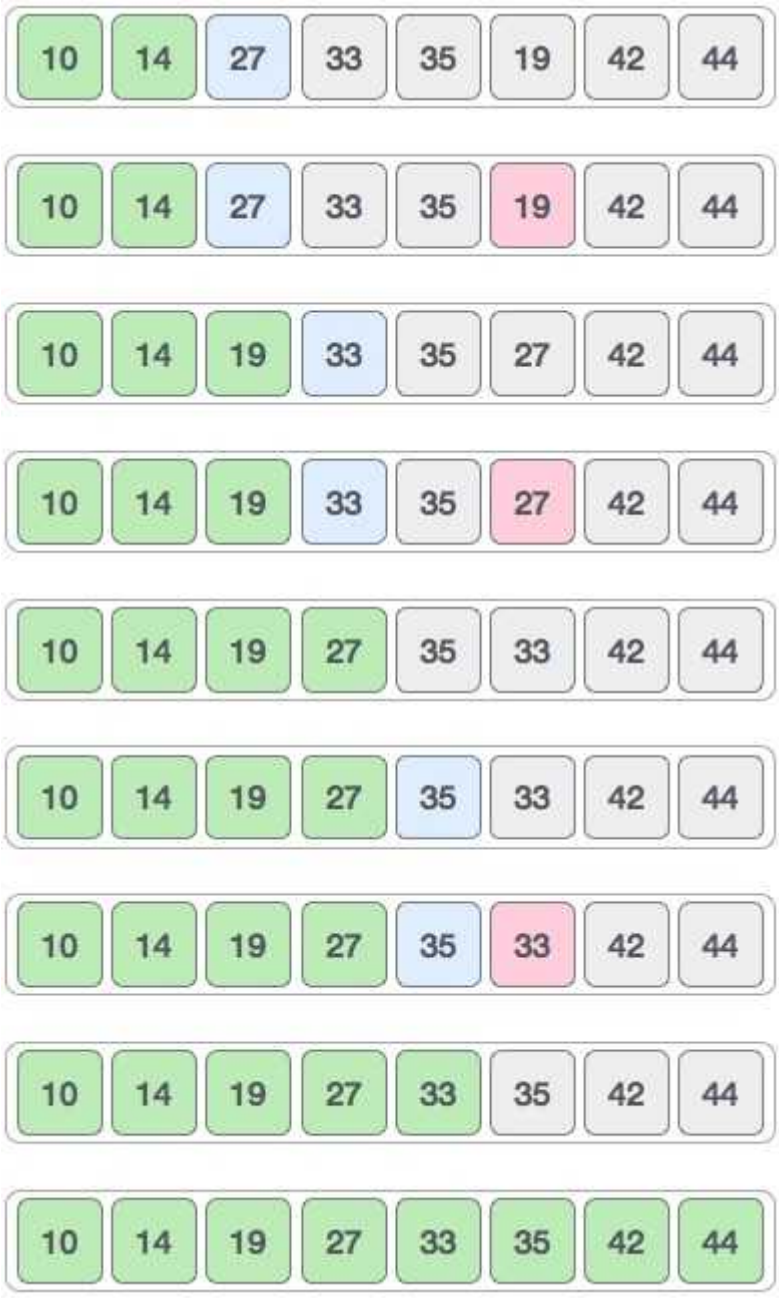
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

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

Pseudocode

```
procedure selection sort

   list  : array of items

   n     : size of list


   for i = 1 to n - 1

   /* set current element as minimum*/

      min = i


      /* check the element to be minimum */
```

```
      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if

   end for

end procedure
```

Data Structures - Merge Sort Algorithm

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

```
Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.
```

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array

   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
```

```
      add b[0] to the end of c

      remove b[0] from b

   end while


   return c

end procedure
```

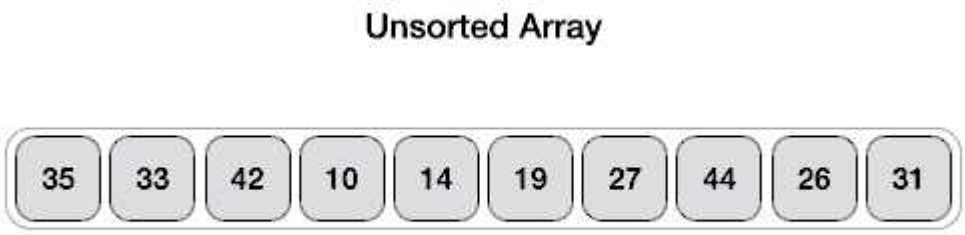
Data Structure and Algorithms - Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

```
Step 1 – Choose the highest index value has pivot
Step 2 – Take two variables to point left and right of the list excluding pivot
Step 3 – left points to the low index
Step 4 – right points to the high
Step 5 – while value at left is less than pivot move right
Step 6 – while value at right is greater than pivot move left
Step 7 – if both step 5 and step 6 does not match swap left and right
Step 8 – if left ≥ right, the point where they met is new pivot
```

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

```
function partitionFunc(left, right, pivot)

   leftPointer = left - 1

   rightPointer = right


   while True do

      while A[++leftPointer] < pivot do

         //do-nothing

      end while


      while rightPointer > 0 && A[--rightPointer] > pivot do

         //do-nothing

      end while


      if leftPointer >= rightPointer

         break

      else

         swap leftPointer,rightPointer

      end if


   end while


   swap leftPointer,right

   return leftPointer
```

```
end function
```

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

```
Step 1 – Make the right-most index value pivot
Step 2 – partition the array using pivot value
Step 3 – quicksort left partition recursively
Step 4 – quicksort right partition recursively
```

Quick Sort Pseudocode

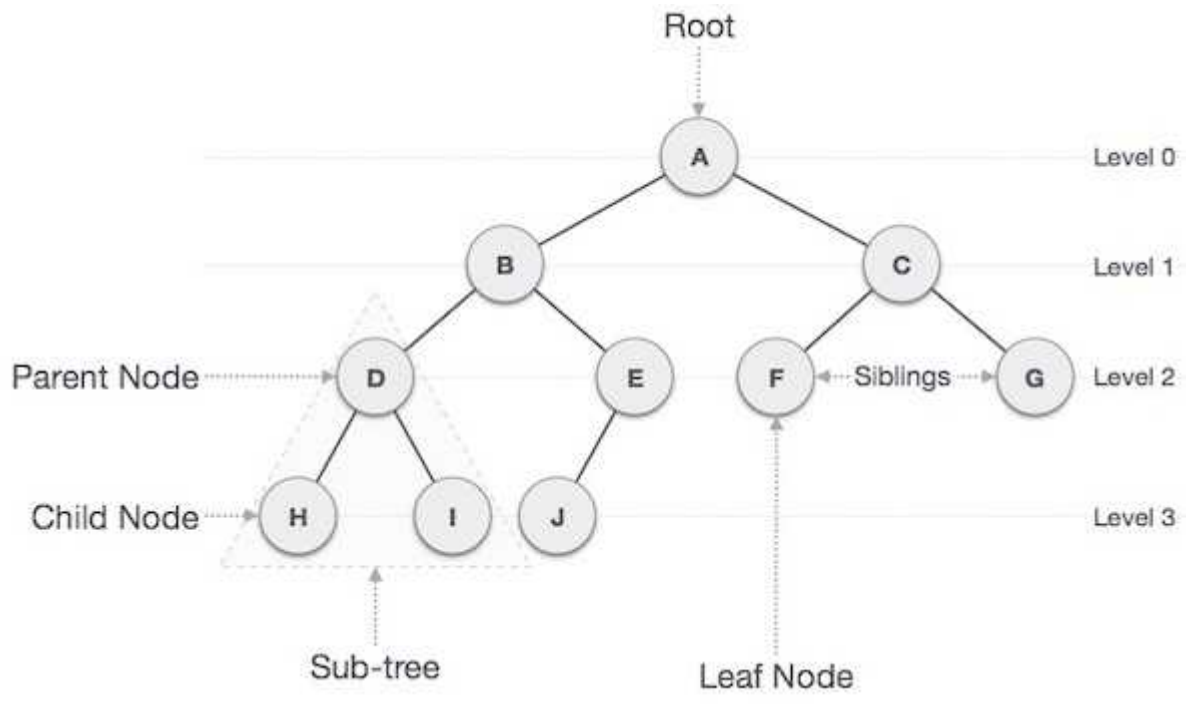
To get more into it, let see the pseudocode for quick sort algorithm −

```
procedure quickSort(left, right)


   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if


end procedure
```

Data Structure and Algorithms - Tree

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.



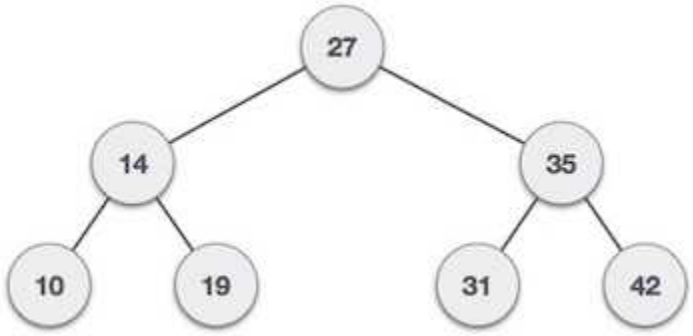
Important Terms

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.

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.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- Insert − Inserts an element in a tree/create a tree.
- Search − Searches an element in a tree.
- Preorder Traversal − Traverses a tree in a pre-order manner.
- Inorder Traversal − Traverses a tree in an in-order manner.
- Postorder Traversal − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation

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;
            }
        }
    }
}
```

Search Operation

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;
   }
}
```

Data Structure & Algorithms - Tree Traversal

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 −

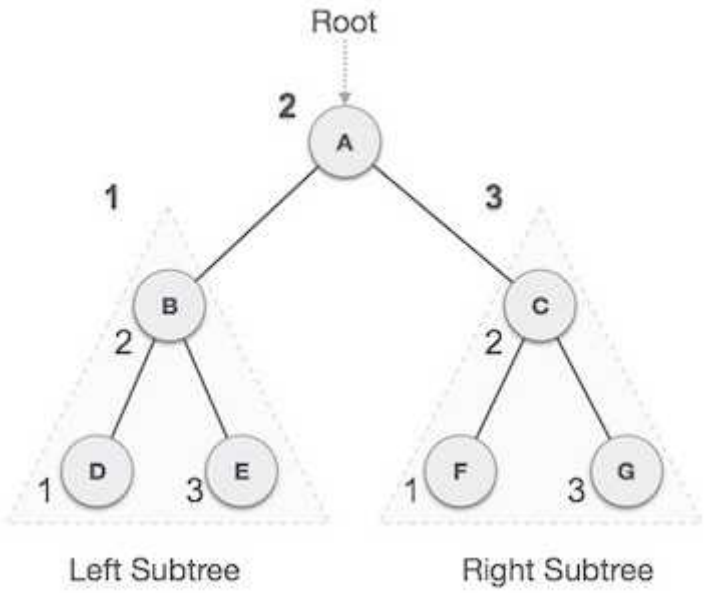
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

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.

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 −

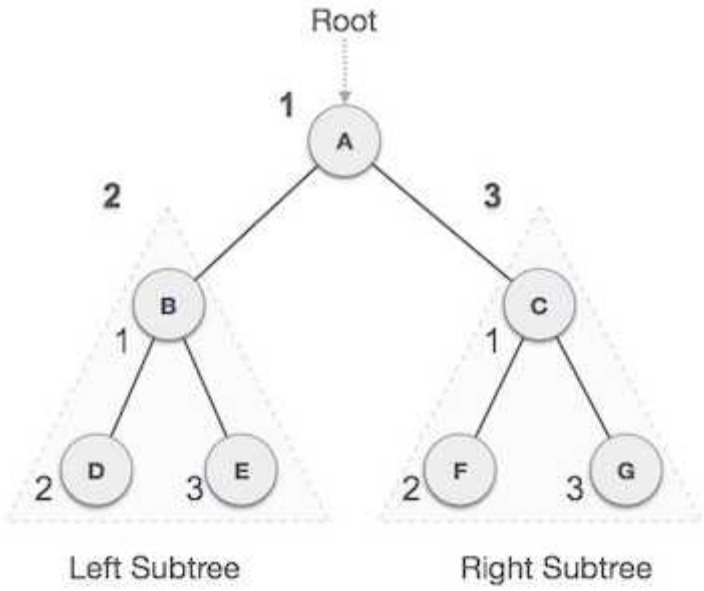
D → B → E → A → F → C → G

Algorithm

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

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 −

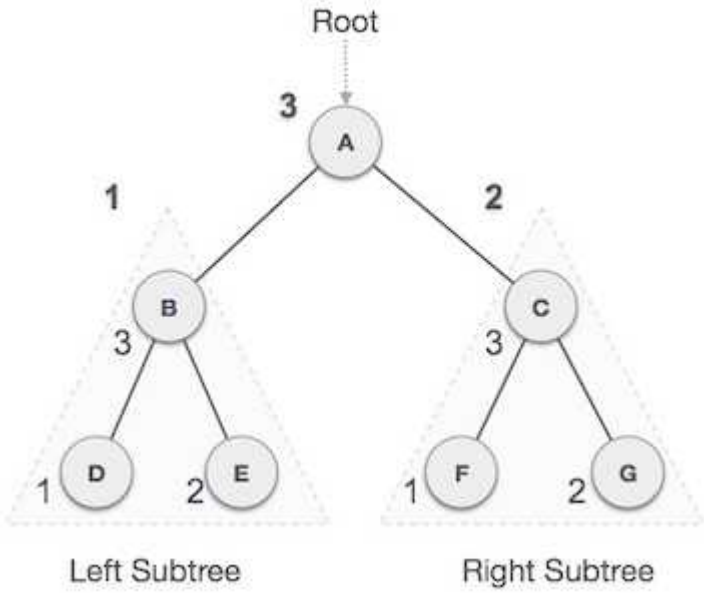
A → B → D → E → C → F → G

Algorithm

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

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 → E → B → F → G → C → A

Data Structure - Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

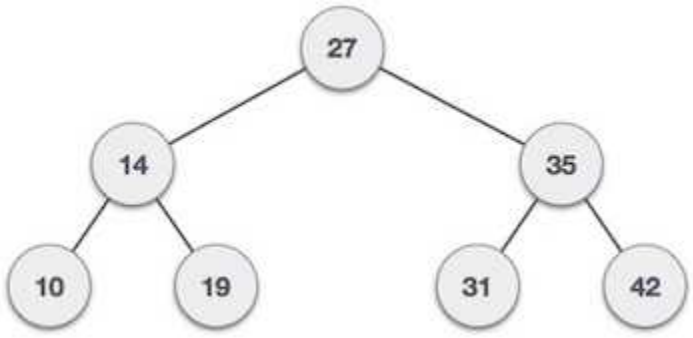
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

```
left_subtree (keys)  ≤  node (key)  ≤  right_subtree (keys)
```

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree −

- Search − Searches an element in a tree.
- Insert − Inserts an element in a tree.
- Pre-order Traversal − Traverses a tree in a pre-order manner.
- In-order Traversal − Traverses a tree in an in-order manner.
- Post-order Traversal − Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

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

Search Operation

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

```
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;
}
```

Insert Operation

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.

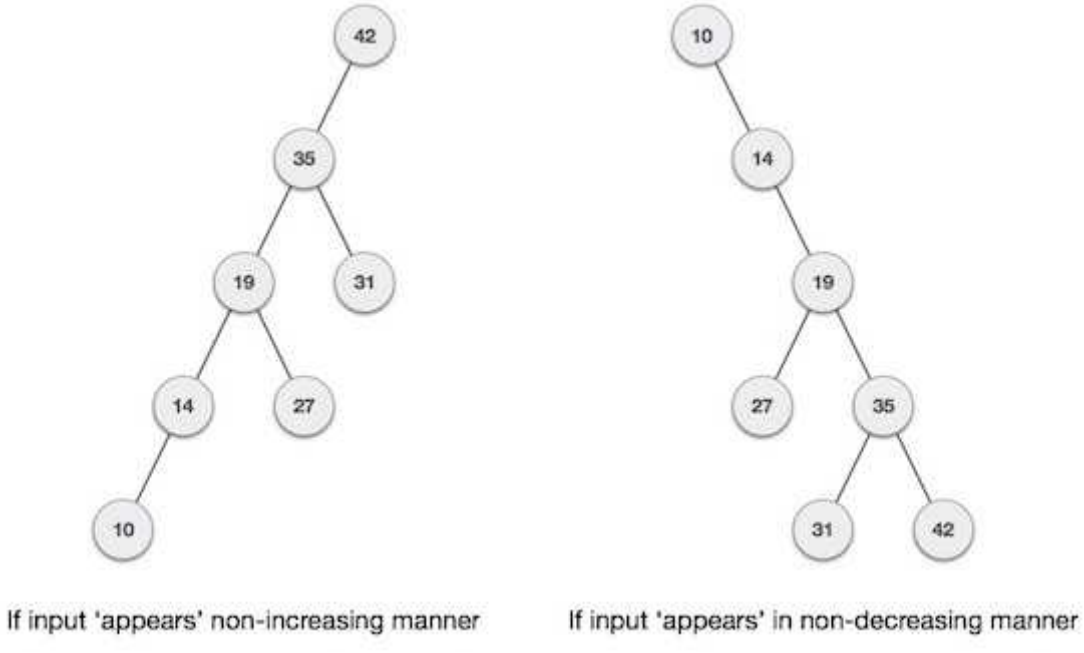
Algorithm

```
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
   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;
         }
      }
   }
}
```

Data Structure and Algorithms - AVL Trees

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



If input 'appears' non-increasing manner

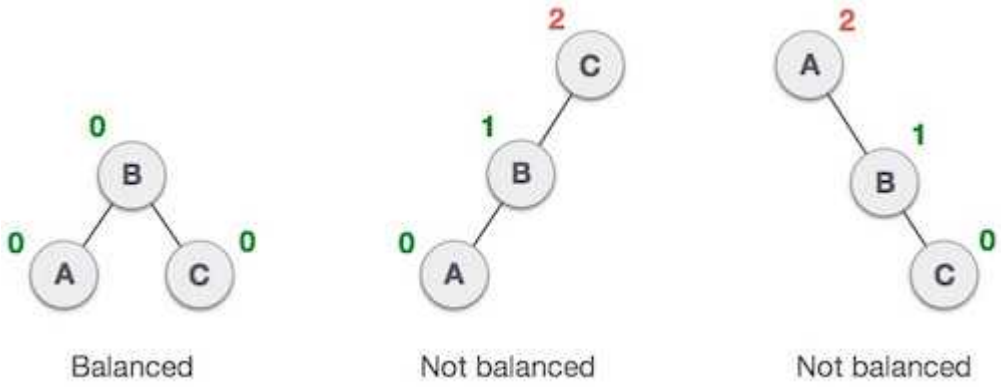
If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left

and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

Here we see that the first tree is balanced and the next two trees are not balanced −



In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

```
BalanceFactor = height(left-sutree) − height(right-sutree)
```

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

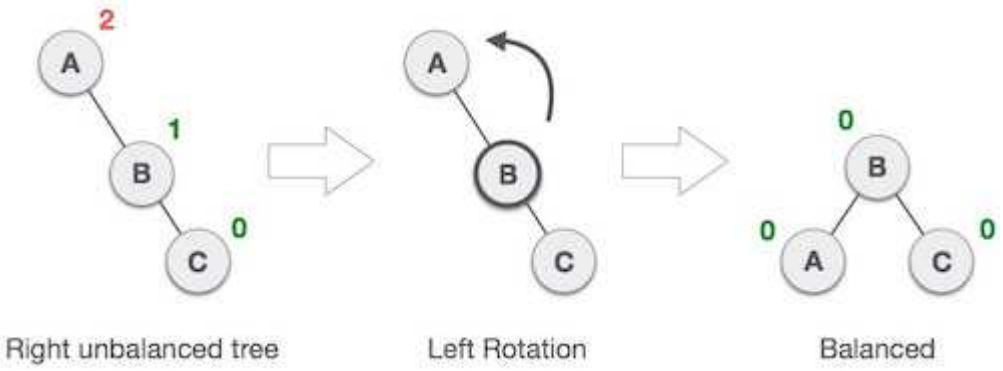
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

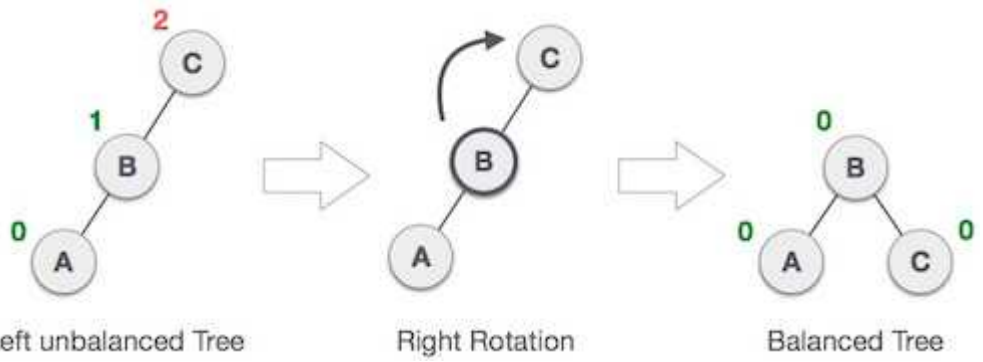
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making Athe left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

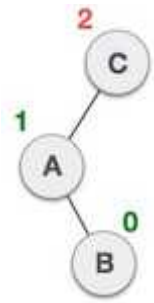
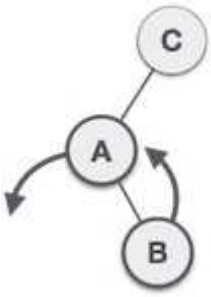
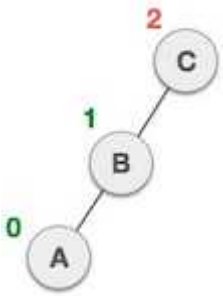
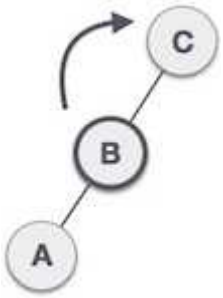
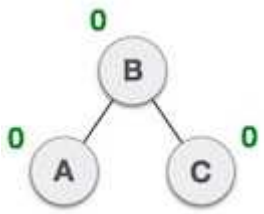


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

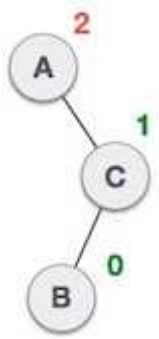
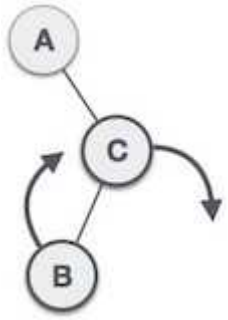
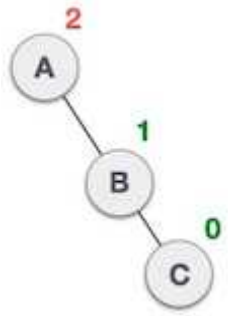
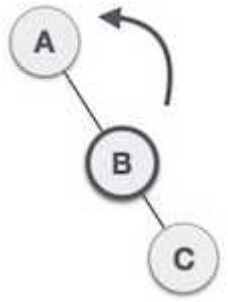
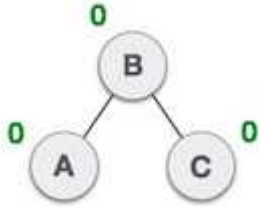
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

State	Action

	A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
	Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

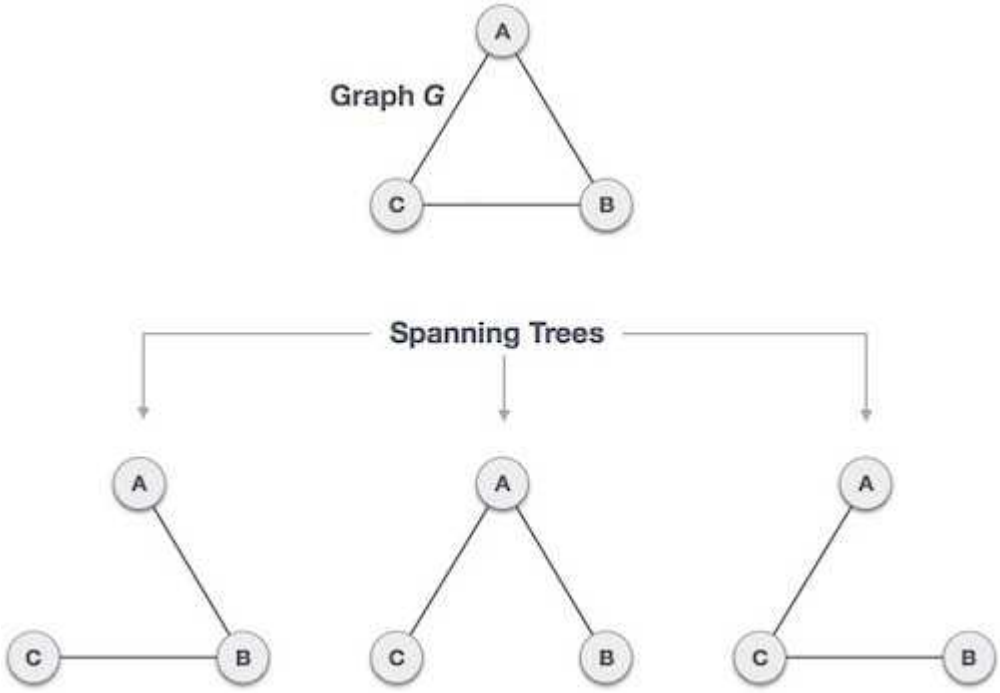
The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.
	First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.
	Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.
	The tree is now balanced.

Data Structure & Algorithms - Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, 3^{3-2} = 3 spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.

- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Mathematical Properties of Spanning Tree

- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here −

- Kruskal's Algorithm
- Prim's Algorithm

Heap Data Structures

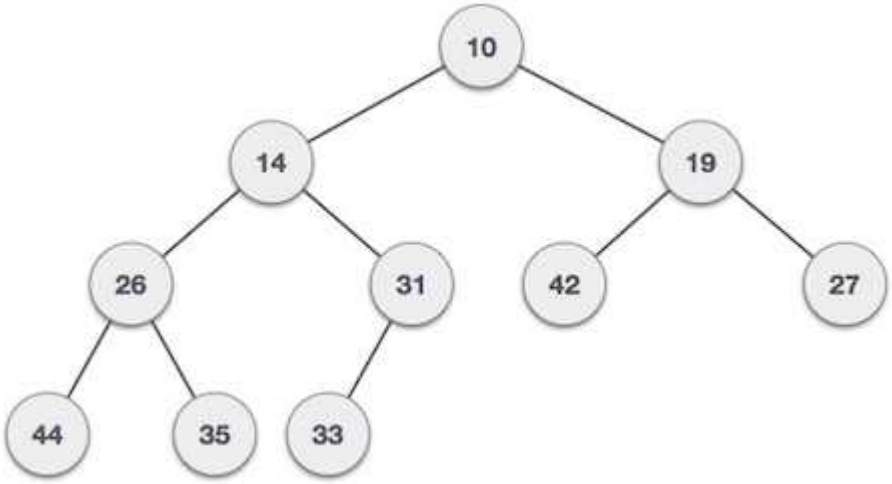
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then −

$$key(\alpha) \geq key(\beta)$$

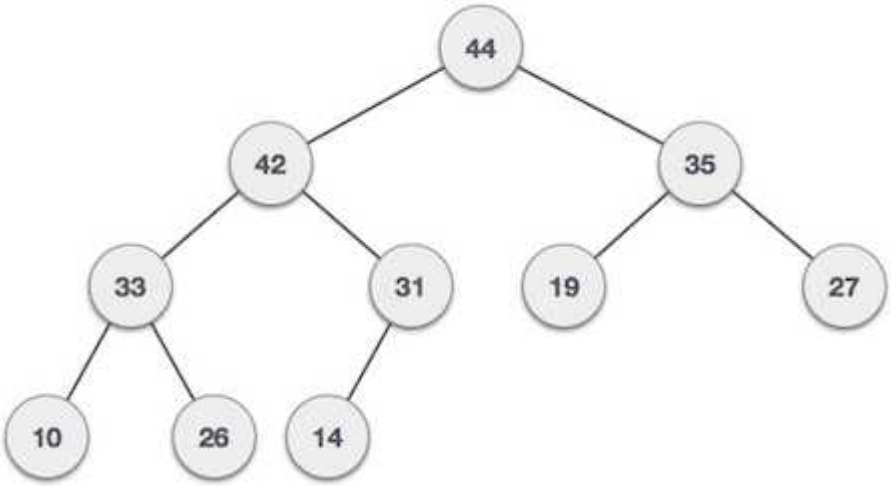
As the value of parent is greater than that of child, this property generates Max Heap. Based on this criteria, a heap can be of two types −

```
For Input → 35 33 42 10 14 19 27 44 26 31
```

Min-Heap − Where the value of the root node is less than or equal to either of its children.



Max-Heap − Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

```
Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.
```

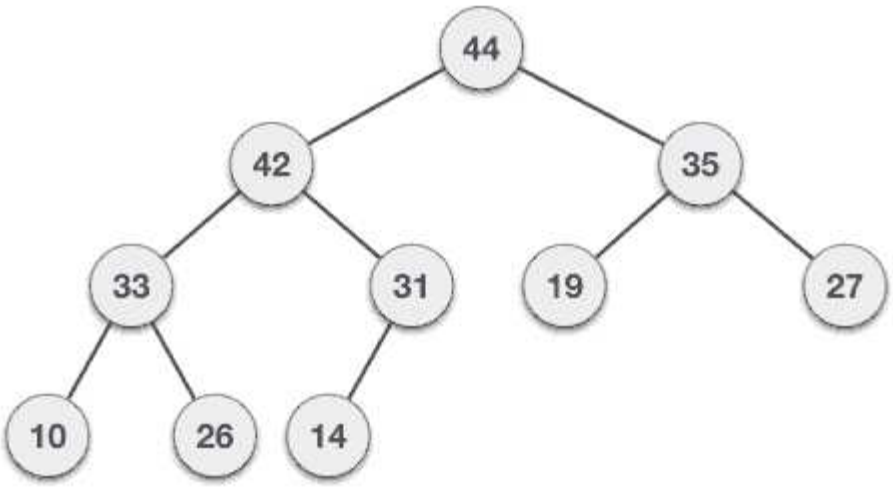
Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

```
Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.
```



Data Structure - Recursion Basics

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function α either calls itself directly or calls a function β that in turn calls the original function α. The function α is called recursive function.

Example − a function calling itself.

```
int function(int value) {

   if(value < 1)

      return;

   function(value - 1);


   printf("%d ",value);

}
```

Example − a function that calls another function which in turn calls it again.

```
int function(int value) {
   if(value < 1)
      return;
   function(value - 1);

   printf("%d ",value);
}
```

Properties

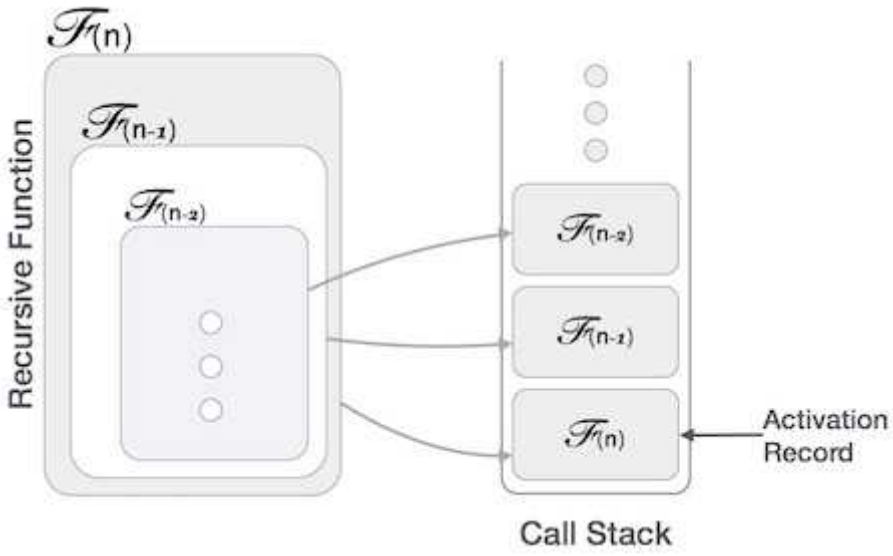
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- Base criteria − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- Progressive approach − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

Many programming languages implement recursion by means of stacks. Generally, whenever a function (caller) calls another function (callee) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity

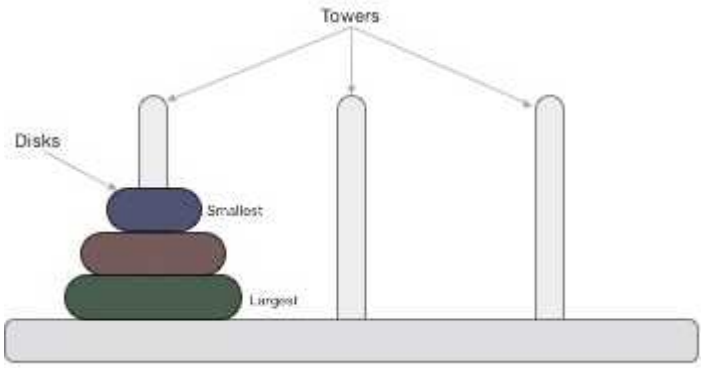
In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

Data Structure & Algorithms - Tower of Hanoi

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



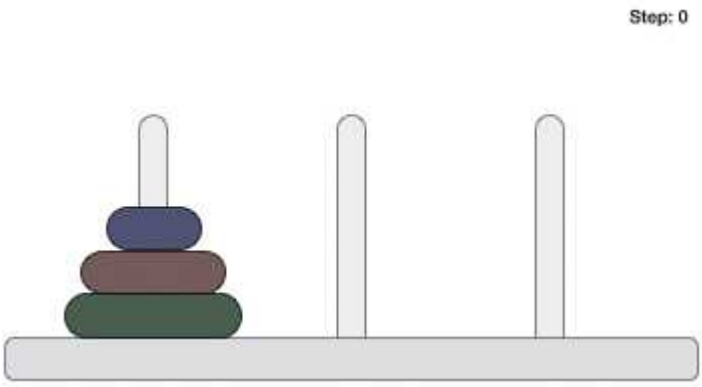
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



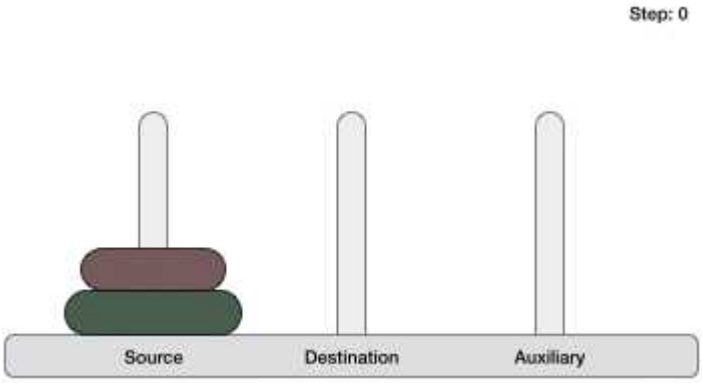
Tower of Hanoi puzzle with n disks can be solved in minimum $2^n - 1$ steps. This presentation shows that a puzzle with 3 disks has taken 2^3 - 1 = 7 steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

```
Step 1 – Move n-1 disks from source to aux
Step 2 – Move nth disk from source to dest
Step 3 – Move n-1 disks from aux to dest
```

A recursive algorithm for Tower of Hanoi can be driven as follows −

```
START

Procedure Hanoi(disk, source, dest, aux)


   IF disk == 1, THEN

      move disk from source to dest

   ELSE

      Hanoi(disk - 1, source, aux, dest)     // Step 1

      move disk from source to dest          // Step 2

      Hanoi(disk - 1, aux, dest, source)     // Step 3

   END IF


END Procedure

STOP
```

Data Structure & Algorithms Fibonacci Series

Fibonacci series generates the subsequent number by adding two previous numbers. Fibonacci series starts from two numbers − F_0 & F_1. The initial values of F_0 & F_1 can be taken 0, 1 or 1, 1 respectively.

Fibonacci series satisfies the following conditions −

```
Fn = Fn-1 + Fn-2
```

Hence, a Fibonacci series can look like this −

F_8 = 0 1 1 2 3 5 8 13

or, this −

F_8 = 1 1 2 3 5 8 13 21

For illustration purpose, Fibonacci of F_8 is displayed as −

Fibonacci Iterative Algorithm

First we try to draft the iterative algorithm for Fibonacci series.

```
Procedure Fibonacci(n)

   declare f0, f1, fib, loop


   set f0 to 0

   set f1 to 1


   display f0, f1


   for loop ← 1 to n


      fib ← f0 + f1

      f0 ← f1
```

```
      f1 ← fib


      display fib
   end for


end procedure
```

To know about the implementation of the above algorithm in C programming language, click here.

Fibonacci Recursive Algorithm

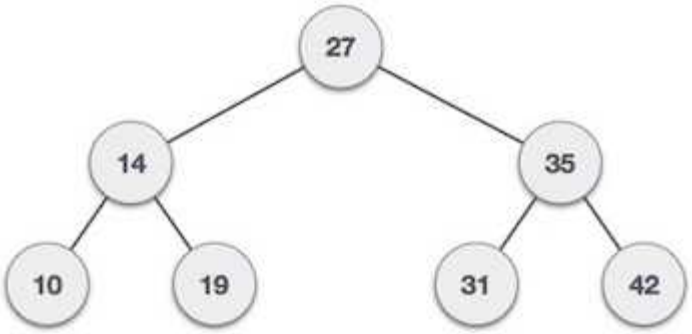
Let us learn how to create a recursive algorithm Fibonacci series. The base criteria of recursion.

```
START

Procedure Fibonacci(n)

   declare f0, f1, fib, loop


   set f0 to 0

   set f1 to 1


   display f0, f1


   for loop ← 1 to n


      fib ← f0 + f1

      f0 ← f1

      f1 ← fib


      display fib
```

Students you shuld know the answer of the following questions after reading the above described material.

What is data-structure?What are various data-structures available?What is algorithm?Why we need to do algorithm analysis?What are the criteria of algorithm analysis?What is asymptotic analysis of an algorithm?What are asymptotic notations?What is linear data structure?What are common operations that can be performed on a data-structure?Briefly explain the approaches to develop algorithms.Give some examples greedy algorithms.What are some examples of divide

and conquer algorithms?What are some examples of dynamic programming algorithms?What is a linked-list?What is stack?Why do we use stacks?What operations can be performed on stacks?What is a queue in data-structure?Why do we use queues?What operations can be performed on Queues?What is linear searching?What is binary search?What is bubble sort and how bubble sort works?Tell me something about 'insertion sort'?What is selection sort?How insertion sort and selection sorts are different?What is merge sort and how it works?What is shell sort?How quick sort works?What is a graph?How depth first traversal works?How breadth first traversal works?What is a tree?What is a binary tree?What is a binary search tree?What is tree traversal?See the below image of a binary search tree, and traverse it using all available methods –



What is an AVL Tree?What is a spanning tree?How many spanning trees can a graph has?How Kruskal's algorithm works?How Prim's algorithm finds spanning tree?What is a minimum spanning tree (MST) ?What is a heap in data structure?What is a recursive function?What is tower of hanoi?What is fibonacci series?What is hashing?What is interpolation search technique?What is the prefix and post fix notation of (a + b) * (c + d) ?

Data Structure Using C(Vedio Tutorials)

Students are directed to go to the below mentioned vedio tutorial on youtube and save them for their preparation. They are very simple and best for understanding:

Sr.No	Topic	Vedio Lectures by	Remarks
1	Introduction to asymtotic notations	Sh.Ravindrababu Ravula	
2	Increment and decrement operator part 1	Sh.ivyanshu Soni	
3	Insertion of an element in an array at specific position	Sh.Sundeep Saradhi	
4	Linear Search	Sh.Sundeep Saradhi	
5	Binary Search	Sh.Sundeep Saradhi	
6	Insertion of an element in an array at specific position	Sh.Sundeep Saradhi	
7	Bubble sort	Sh.Sundeep Saradhi	
8	One Dimensional Array	Sh.Sundeep Saradhi	

9	Two Dimensional Array	Sh.Sundeep Saradhi	
10	Structures	Sh.Sundeep Saradhi	
11	Structures and functions	Sh.Sundeep Saradhi	
12	Parameter Passing	Sh.Sundeep Saradhi	
13	Stack Operations Logic and program	Sh.Srinivas	
14	Operations on Queue	Sh.Srinivas	
15	Circular Queue	Sh.Srinivas	
16	Infix to postfix Conversions	Sh.Srinivas	
17	Singly Linked List	Sh.Srinivas	
18	Doubly Linked List	Sh.Srinivas	
19	Linked list	Mycodeschool	
20	Insertion at end in singly linked list	Geekyshows	
21	Traversing a singly linked list	Geekyshows	
22	Insertion in an empty singly linked list	Geekyshows	
23	Program to insert node in singly linked list	Geekyshows	
	FOR LAB SESSIONS		
	Students are directed to do the lab sessions on www.vlab.co.in		

Summary sheet having digital links of E-notes

Name of the Polytechnic	Branch	Semester	Subject Name	Chapter Sr.no.	Chapter/Unit Name (as per HSBTE Curriculum)	Sub topic Sr. No.	Subject sub-topic Name	e-notes Digital Link (Pdf /docx. only) on google drive
Govt Polytechnic, Jhajjar	Computer Engg.	4th	Data Structure Using C	1	**Fundamental Notations**	1.1	Problem solving concept top down and bottom up design, structured programming	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						1.2	Concept of data types, variables and constants	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						1.3	Concept of pointer variables and constants	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
				2	**Arrays**	2.1	Concept of Arrays	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						2.2	Storage representation of multi-dimensional arrays.	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						2.3	Operations on arrays with Algorithms (searching, traversing, inserting, deleting)	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
				3	**Linked Lists**	3.1	Introduction to linked list	https://drive.google.com/open?id

								=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						3.2	Representation of linked lists in Memory	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						3.3	Operations on linked list (Insertion, deletion and traversals)	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						3.4	Application of linked lists	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						3.5	Doubly linked lists	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						3.6	Operations on doubly linked lists (Insertion, deletion and traversals)	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
				4	**Stacks, Quesues and Recursion**	4.1	Introduction to stacks	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.2	Representation of stacks	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.3	Implementation of stacks	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.4	Applications of stacks	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.5	Introduction to queues	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.6	Implementation of queues	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.7	Circular Queues	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.8	De-queues	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.9	Application of Queues	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						4.1	Recursion	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
				5	**Trees**	5.1	Concept of Trees	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						5.2	Representation of Binary tree in memory	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						5.3	Traversing Binary Trees (Pre order, Post order and In order)	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						5.4	Searching, inserting and deleting binary search trees	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						5.5	Introduction to Heap	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
				6	**Sorting and Searching**	6.1	Introduction to sorting and searching	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						6.2	Search algorithm (Linear and Binary)	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy
						6.3	Sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Selection Sort, Merge Sort, Heap Sort	https://drive.google.com/open?id=1NnuOiVRwN-yiLvKVWgk911sDC2LHDnuy