PrepInsta Handbook for Coding
For Placement Preparation

PrepInsta Technologies Pvt Ltd

Preface:

This book contains all the information regarding DSA which is asked in Placement Tests. Nowadays you will see many books and online pages providing information on DSA. Mostly those books contain one section i.e either the coding questions or the theoretical part. There is no such proper book providing all the updated information at one single place.

This book contains various questions and theory knowledge of all the types asked in Placement tests. This book carries all the theory and practice questions along with examples and shortcut methods.

It is hoped that the subject matter will instill trust in the applicants, and that the book will assist them in finding an ideal teacher.

Disclaimer: This book is made and published under the complete knowledge and expertise of the Author, however if there will be any error then it will be taken care of in the next Revised edition. Constructive suggestions and feedback are most welcome by our esteemed readers.

First Edition

May 2021
Edition: 1 (2021)

Published on GSM Paper with PrepInsta WaterMark

Publisher Address: K-3, Director Vidyamandir Shastri Nagar, Meerut, UP, 250004

Type settings : PrepInsta Technologies Private Limited

Printer Name: Creative Print Arts

Price- Rs.999/-

- The correct price of the book is already mentioned on this page. Any revised price on the frontend or backend of the book is not acceptable.

Address of the Publisher:

K-3, Director Vidyamandir Shastri Nagar, Meerut, UP, 250004

Publication Team:

PrepInsta Technologies Private Limited

Contents

\

Chapter 1. Linear search

What is Linear Search ?

Searching, in normal ways, can be coined as" to find the hidden thing". In data structure.the searching algorithm is used to find whether a given number is present and if it is present then at what location it occurs.

There are two types of searching algorithms present in data structure through which searching any data becomes easier.

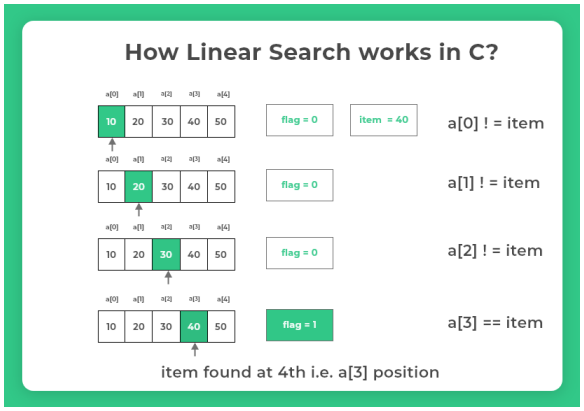
1. Linear search or sequential search
2. Binary search

Implementation:-

- Step 1- The list given is the list of elements in an unsorted array.
- Step 2- The element to be searched is called an item, as mentioned in the image, so the value ***item is*** compared with all the elements of the array starting from the 0th element.
- Step 4- As soon as any index contains the same value, we stop further comparisons and change ***flag*** value from 0 to 1
- Step 4- If the value is not found until the end of the array then stop and print value not found

Algorithm:-

1. i=1
2. if i>n,go to step 7
3. if A[i]=x,go to step 6
4. i=i+1
5. Go to step 2
6. print element x found at i
7. print element not found
8. exit



C Code:-

```
#include <stdio.h>  //header files
int main () //main function
{
 int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
 int item, i, flag;

 printf("\nEnter Item which is to be find\n");
 scanf("%d",&item);

 //looping logic
 for (i = 0; i< 10; i++)
 {
  // if else condition
   if(a[i] == item)
   {
     flag = i+1;
     break;
   }
   else
     flag = 0;
 }
```

```
  if(flag != 0)
  {
    //printing the element which is to find
    printf("\nItem found at location %d\n",flag);
  }
  else
  {
    printf("\nItem not found\n");
  }
 return 0;
}
```

C++ Code

```
#include <iostream>
using namespace std;
int main()
{
 int list[100],size,i,search_el;//Initializing variables.

 cout<<"Enter size of the list: ";//Taking inputs.
 cin>>size;

 cout<<"Enter elements of the list: ";
 for(i = 0; i < size; i++)
 cin>>list[i];

 cout<<"Enter the element to be search: ";
 cin>>search_el;

 //Searching element using linear search.
 for(i = 0; i < size; i++)
 {
   if(search_el == list[i])
   {
     cout<<"Element is found at "<< i+1 <<" place in the
list";
     break;
   }
 }
 if(i == size)
   cout<<"Element is not present in the list";

}
```

Java Code

```
import java.util.*;
public class Main
{
public static int search(int ar[], int s)
{
   int l = ar.length;
   for(int i = 0; i < l; i++)
   {
      if(ar[i] == s)
         return i;
   }
   return -1;
}
public static void main(String args[])
{
   int ar[] = { 2, 3, 4, 10, 40 };
   System.out.println("Enter a number you want to search
for - ");
   Scanner s= new Scanner(System.in);
   int num=s.nextInt();
   int result = search(ar, num)+1;
   if(result == -1)
      System.out.print("Element not found.");
   else
      System.out.print("Element found at index " + result);
}
}
```

Output:-

```
Enter Item  which is to be find
23
Item found at location  2
Enter Item  which is to be find
3
Item not found
```

More about Linear Search

Pros

1. Very easy to understand and implement

2. Quick to write the code

Cons

1. Time complexity is bad

2. Binary Search gives better, efficient and faster results

Time Complexity For Linear Search

Best

O(1)

Average

O(n)

Worst

O(n)

Space Complexity

O(1)

Average Comparisons

(n+1)/2

Question 1. The average number of key comparisons done in a successful sequential search in a list of length it is –

1. (n-1)/2
2. Log n
3. (n+1)/2
4. (n)/2

(Amazon – Mettl Test)

Solution & Explanation

Just say if you have to find a given element in a sequential search. It can be found in the first try, then 1 comparison is required similarly…total comparisons can be 1+2+…+n = n(n+1)/2

Avg will be n(n+1)/2 divided by n (total elements) = (n+1)/2

Ans. Option C

Question 2. The worst-case occurred in the linear search algorithm when

1. The element in the middle of an array
2. Item present in the last
3. Item present in the starting
4. Item has the maximum value

(TCS NQT)

Solution & Explanation

If the element is situated at the end of the array, it takes maximum time to search for that element.

Ans. Option B

Chapter 2. Binary Search

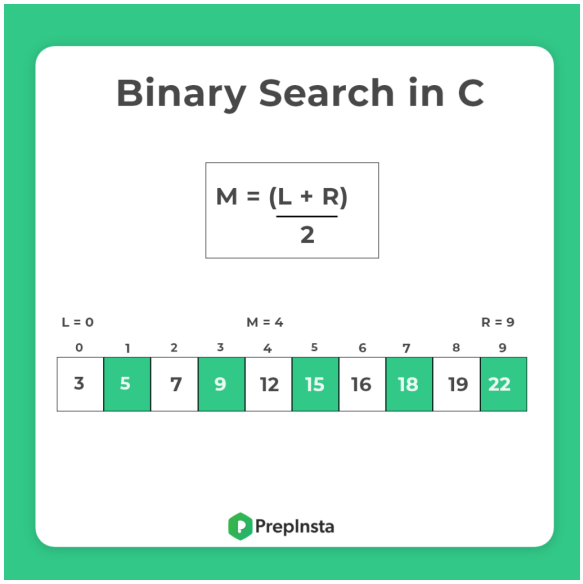
What is Binary Search ?

Binary search is a very fast and efficient searching algorithm. It requires to be list be in a sorted order, ie; either in ascending or descending.In this method, to search an element you might compare that respective element present in the center of the list and if it's same then the search is successfully finished and if not, then the list is divided into two parts:one from 0th element to the centered element and other from the centered element to the last element.

How Binary Search works?

1. The searching algorithm proceed from any of two halves
2. Depends upon whether the element you are searching is greater or smaller than the central element
3. If the element is small then, searching is done in first half
5. If it is big then searching is done in the second half.

It is a fast search algorithm with the complexity of O(log n).



Implementation of Binary Search

- Step 1- In the Given array, first you need to find the middle of the list using formula mid=(min+max)/2.

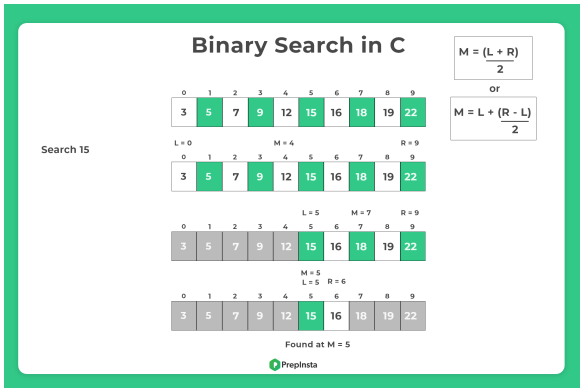
Step 2-If the searched element ie; say 15(in context with the image given below) is in the center then the search is terminated, as we can see here it is not.

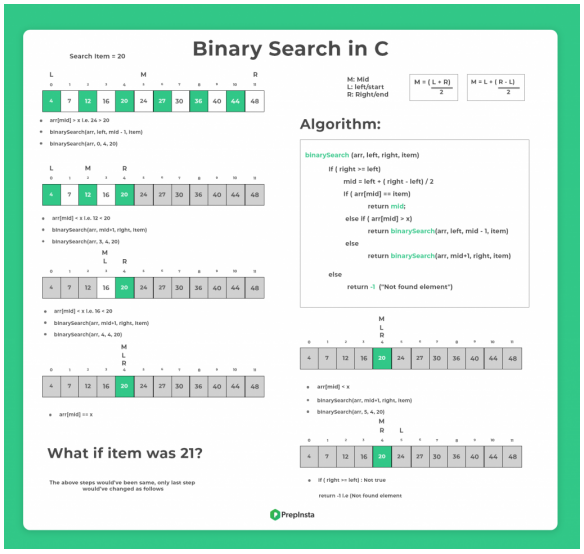
Step 3- With the help of this algorithm we will try to find the value. First of all you have to define two variables low and high so low is 3 and high is 22. Now you have to find the mid element.so we will find the mid value of the formula.

Step 4- We would know that in which half the searching would be done.

Step 5- So we would go for the second half as the searching element i.e; 15 is greater than the mid element i.e 12.

Step 6- The process repeats until we find the element.





Algorithm of Binary Search
Step 1- while low < high

do mid <-(low+high)

Step 2- if V=A[mid]

return mid

Step 3- if V=A[mid]

low<- mid+1

Step 4- else high<- mid-1

Step 5- return Nil

C Code

```
#include<stdio.h>  //header files

int binarySearch(int[], int, int, int);
```

```
int main() //main function
{
 int arr[10] = {9, 26, 33, 47, 53, 60, 75, 80, 86, 99};
  int item, location = -1;

  printf("Enter the item which you want to find ");
  scanf("%d",&item);

  location = binarySearch(arr, 0, 9, item);

  if(location != -1) //looping logic
  {
    printf("Item found at location %d",location); //printing
the element
  }
  else
  {
    printf("Item not found");
  }
  return 0;
}

int binarySearch(int a[], int start, int last, int item)
{
  int mid;
  if(last >= start)
  {
    mid = (start + last)/2;
    if(a[mid] == item){
       return mid+1;
    }
    else if(a[mid] < item){
       return binarySearch(a,mid+1,last,item);
    }
    else{
       return binarySearch(a,start,mid-1,item);
    }
  }
  return -1;
}
```

C++ Code

```
#include <iostream>//Header file.
using namespace std;

int binarySearch(int[], int, int, int);

int main() //Main function.
{
  int arr[10] = {19, 26, 37, 53, 56, 61, 77, 82, 87, 91};
  int search_element, location=-1;

  cout<<"Enter the element which you want to search: ";
  cin>>search_element;

  location = binarySearch(arr, 0, 9, search_element);

  if(location != -1)
  {
    cout<<"Search element found at"<<location<<"
location"; //Printing the result.
  }
  else
  {
    cout<<"Search element not present";
  }
  return 0;
}

int binarySearch(int a[], int left, int right, int
search_element)
{
  int middle;
  if(right >= left)
  {
    middle = (left + right)/2;
    if(a[middle] == search_element) //Checking if elemnet is
present at middle.
    {
      return middle+1;
    }
    else if(a[middle] < search_element) //Checking if
elemnet is present in greater half.
    {
      return binarySearch(a,middle+1,right,search_element);
    }
    else //Checking if elemnet is present in samller half.
    {
      return binarySearch(a,left,middle-1,search_element);
    }

  }
  return -1;
}
```

Java Code

```
import java.util.*;
class Main {
    int search(int arr[], int x)
    {
        int l = 0, r = arr.length - 1;
        while (l <= r) {
            int m = l + (r - l) / 2;

            if (arr[m] == x)   //if x is in the middle
                return m;
            if (arr[m] < x) //if x is greater, search the right half
                l = m + 1;
```

```
        else        //if x is smaller, search the left half
            r = m - 1;
      }
      return -1; //not found
   }
   public static void main(String args[])
   {
      Main s=new Main();
      Scanner sc= new Scanner(System.in);
      System.out.println("Enter the element to be searched
");
      int num=sc.nextInt();
      int arr[]={3,5,6,7,10,14,15,75,88,96,99};
      int result = s.search(arr, num);
      if (result == -1)
         System.out.println("No match found in the Array");
      else
         System.out.println("Match found at index " +
result);
   }
}
```

Output:-
Enter the item which you want to find 26
Item found at location 2
Enter the item which you want to find 3
Item not found

Time Complexity For Binary Search

Best

O(1)

Average

O(log n)

Worst

O(log n)

Space Complexity

O(1)

Average Comparisons

Log(N+1)

Exercise

Question 1.Binary search algorithm can't be applied to

1. Pointed Array List
2. Sorted Binary Tree
3. Unordered Array
4. Sorted Linked List

(Cisco, Walmart Labs)

Solution & ExplanationYou cannot use a binary search on an unsorted list. Your best bet is to probably iterate over the list's items until you find the element you are looking for, an O(n) operation, i.e. do linear search

Ans. C

Question 2. Which of the following searching algorithms works on the divide and conquer?

1. Binary Search
2. Linear
3. Sequential
4. Merge Sort

(CoCubes, Atos Syntel)

Solution & ExplanationBinary search only checks the middle element of the array. when it checks the middle element and middle element is not equal to the given element then we divide the list in two subparts.Ans: Both A & D

Chapter 3. Classification of Sorting Algorithms

Stable Vs Unstable Sorting

Stable sort sorts the identical elements in their same order

as they appear in the input.

Examples: Bubble sort, Insertion sort, Merge Sort,

Counting sort

In *unstable sort*, order of identical elements is not

guranteed to stay in the same order as they appeared in the input. Examples: Quick Sort, Heap Sort, Selection sort

- Stability is not an issue when equal elements are indistinguishable such as with integers.
- Any given sorting algorithm which is not stable can be modified to stable.
- Generally, unstable sorting is faster than stable sorting.
- We need a stable sort to sort the cards by their numbers and suit.

Internal Vs External Sort

Internal sorting is that which takes place entirely within the main memory of the computer. It comes into play when the dataset is small.

External sorting is that which can handle massive amounts of data at a time. Data resides in the hard disk in case of external sorting.

Comparison based Sorting and Counting based Sorting

In Comparison based sorting, a comparator is required to compare numbers or items. This comparator defines the ordering to arrange the elements.

Examples: Merge Sort, Quick Sort, Heap Sort

Counting based sorting is the algorithm that performs the sorting without comparing the elements rather by making certain assumptions about the data they are going to sort**.**

Examples: Counting sort, Radix sort

In-Place and Not-In-Place Sorting

In-Place Sorting means to sort the array by modifying the element order directly within the array.

- No auxiliary data structure is used.
- There can be only a constant amount of extra space usually less than log(n). So this algorithm is space efficient.

Examples: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort

Not-In-Place Sorting is that which uses auxiliary data structure to sort the array.

Examples: Merge Sort (It requires additional O(n) space to perform the merge operation), Quick Sort

Chapter 4. Bubble Sort

Sorting is the process of arranging the data in some logical order. Bubble sort is an algorithm to sort various linear data structures.

The logical order can be ascending and descending in case of numeric values or dictionary order in case of alphanumeric values.

Bubble Sort is very simple and easy to implement sorting technique.

In bubble sort technique,each pair of elements is compared. Elements are swapped if they are not in order**.**

The worst case complexity of bubble sort is $O(n^2)$.

Implementation of Bubble Sort

Let’s take an example ,we have a list of number stored in array

Logic starts with comparison of the first two elements and if the left element is greater than the right element,they swap their position. comparison continues till the end of the array.

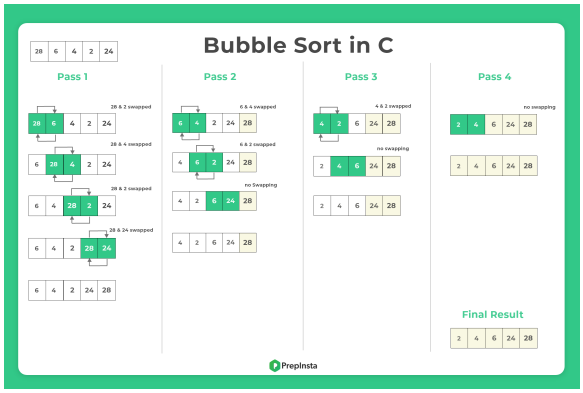
If we talk about the example taken in the below image, the first will be in between 28 and 6.

Since the left element i.e; 6 is smaller than the right element i.i; 28, they will get swapped.

Similarly, in the next iteration we will compare 28 and 4, and again they will get swapped.

This process will continue until the loop reaches the end of the array, this will be regarded as Pass 1, at the end of which we have the largest element on the right most end.

Passes like this will continue until we have all the elements in their correct order, you can refer to the image below, for better understanding.



Algorithm of Bubble Sort

Step 1: Repeat for round=1,2,3…..N-1

Step 2: Repeat For i=0,1,2…N-1 round

Step 3: IF A[i] > A[i+1]

SWAP A[i] and A[i+1]

[END OF INNER LOOP]

[END OF OUTER LOOP

Step 4: Return

C Code:-

```
#include <stdio.h>

/* Function to print array */
void display(int arr[], int size)
{
  int i;
  for (i=0; i < size; i++)
     printf("%d ", arr[i]);
  printf("\n");
}

// Main function to run the program
int main()
{
  int array[] = {5, 3, 1, 9, 8, 2, 4,7};
  int size = sizeof(array)/sizeof(array[0]);

  printf("Before bubble sort: \n");
  display(array, size);

    int i, j, temp;
 for (i = 0; i < size-1; i++){

    // Since, after each iteration righmost i elements are
sorted
    for (j = 0; j < size-i-1; j++)
       if (array[j] > array[j+1])
       {
         temp = array[j]; // swap the element
         array[j] = array[j+1];
         array[j+1] = temp;
       }
  }
  printf("After bubble sort: \n");
  display(array, size);
  return 0;
}
```

C++ Code

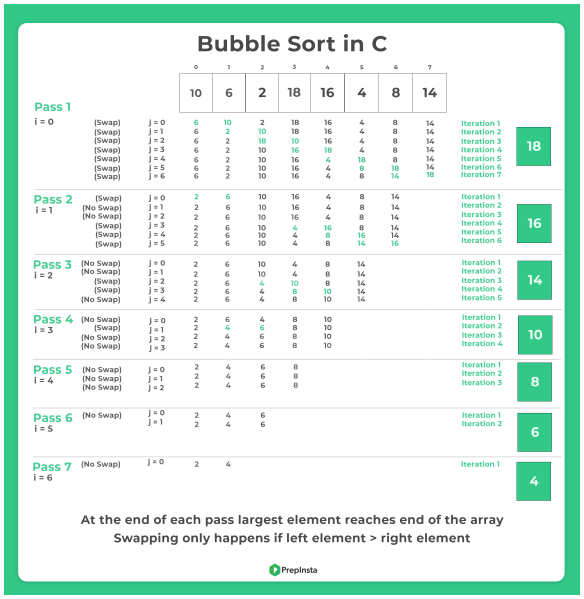
```
#include <iostream>
using namespace std;
void swap(int *var1, int *var2)
{
   int temp = *var1;
   *var1 = *var2;
```

```
    *var2 = temp;
}
//Here we will implement bubbleSort.
void bubbleSort(int arr[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i++)
      //Since, after each iteration rightmost i elements are
sorted.
      for (j = 0; j < n-i-1; j++)
          if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
// Function to print array.
void display(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout<<arr[i]<<"\t";
    cout<<endl;
}
//Main function to run the program.
int main()
{
    int array[] = {5, 3, 1, 9, 8, 2, 4,7};
    int size = sizeof(array)/sizeof(array[0]);
    cout<<"Before bubble sort: \n";
    display(array, size);//Calling display function to print
unsorted array.
    bubbleSort(array, size);
    cout<<"After bubble sort: \n";
    display(array, size);//Calling display function to print
sorted array.
    return 0;
}
```

Java Code

```
class Main
{
  void bubbleSort(int a[])
     {
        int len = a.length; // calculating the length of array
        for (int i = 0; i < len-1; i++)
          for (int j = 0; j < len-i-1; j++) if (a[j] > a[j+1])
//comparing the pair of elements
              {
               // swaping a[j+1] and a[i]
                 int temp = a[j];
                 a[j] = a[j+1];
                 a[j+1] = temp;
               }
      }
   /* Prints the array */
    void printArray(int a[])
      {
        int len = a.length;
        for (int i=0; i<len; i++)
        System.out.print(a[i] + " "); //printing the sorted
array
        System.out.println();
       }

    // Main method to test above
      public static void main(String args[])
        {
          Main ob = new Main();
          int a[] = {64, 34, 25, 12, 22, 11, 90};
          ob.bubbleSort(a);//calling the bubbleSort function
          System.out.println("Sorted array");
          ob.printArray(a); //calling the printArray function
        }
}
```

Output:
Before bubble sort
5 3 1 9 8 2 4 7
After bubble sort:
1 2 3 4 5 7 8 9



Performance Based Analysis of Bubble Sort Algorithm
Pros
1. Easy to implement
2. Cool to implement
3. Gives the largest value of the array in the first iteration itself.

4. Or the smallest value of array in the first iteration itself (Minor tweak in implementation)
5. No demand for large amounts of memory for operation

Cons
1. Noob (Bad) algorithm 😀
2. Very horrible time complexity of $O(n^2)$

Interesting Facts

1. Average and worst case time complexity: $o(n^2)$
2. Best case time complexity: n when array is already sorted.
3. Worst case: when the array is reverse sorted.

Time Complexity For Bubble Sort

Best

$\Omega(n)$

Average

$\Theta(n^2)$

Worst

$O(n^2)$

Chapter 5. Insertion Sort

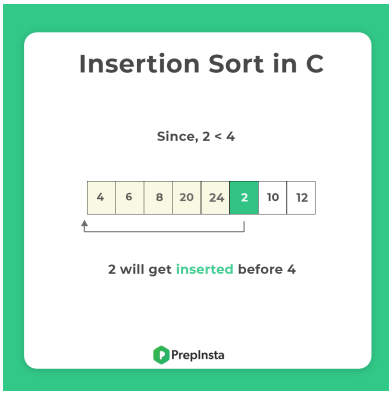
How Insertion Sort works

In this technique we pick an Element and then insert it at the appropriate position in ascending and descending order.

Insertion Sort

If we have an element then it requires a (n-1) pass to sort. In each pass we insert the current element at the appropriate position so that the elements in current range are in order.

- In insertion sort A Sublist (or sorted array) is maintained which is always sorted. This algorithm is not suitable for large data sets.
- The average and worst complexity of an insertion sort is $O(n^2)$.
- This is less efficient on a list containing more numbers of elements.
- The main advantage of the insertion sort is moderity.it is also exhibits a good performance.

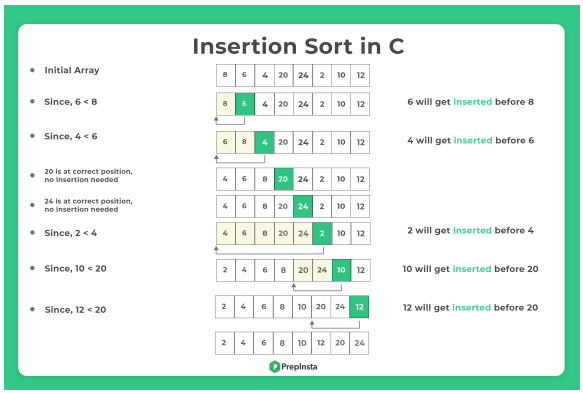


Execution of Insertion Sort

Pass 1- First of all to give the unsorted array.In this array the first element should be fixed.so the compares second and third element.In this array 5>2 then they are swapped .

Pass 2- In this array algorithm compares the next two elements and arranges the element with their position.

Pass 3- the final array is Sorted.



Algorithm of Insertion Sort

Insertion Sort-(A,N): A is an array with N elements.

Step 1- I=1;

Step 2- Repeat Step 3 and 5 While I<N.

Step 3-temp=A[I],J=I-1;

Step 4-Repeat While J>=0 and temp<A[J]

A[J+1]=A[J] and J=J-1

Step 5-A[j+1]=temp,I=I=1

Step 6-Exit

C code:-

```
#include <stdio.h>

// Here we are implementing Insertion sort
void insertionSort(int array[], int size)
{
  int i, target, j;
  for (i = 1; i < size; i++)
  {
   target = array[i];
   j = i - 1;

   /* Here the elements in b/w arrary[0 to i-1]
   which are greater than target are moved
   ahead by 1 position each*/
   while (j >= 0 && array[j] > target)
   {
    array[j + 1] = array[j];
    j = j - 1;
   }
   array[j + 1] = target;
  }
}

/* Function to print array */
void display(int arr[], int size)
{
  int i;
  for (i=0; i < size; i++)
     printf("%d ", arr[i]);
  printf("\n");
}

// Main function to run the program
int main()
{
  int array[] = {5, 3, 1, 9, 8, 2, 4,7};
  int size = sizeof(array)/sizeof(array[0]);

  printf("Before Insertion sort: \n");
  display(array, size);

  insertionSort(array, size);

  printf("After Insertion sort: \n");
  display(array, size);
  return 0;
}
```

C++ Code

```
#include <iostream>
using namespace std;

//Function to print array.
void display(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
   cout<< arr[i]<<"\t";
   cout<<"\n";
```

```
}

//Main function to run the program.
int main()
{
   int array[] = {5, 3, 1, 9, 8, 2, 4,7};
   int size = sizeof(array)/sizeof(array[0]);

   cout<<"Before Insertion sort: \n";
   display(array, size);//Using dispaly function to print
unsorted array.

   int i, target, j;
   for (i = 1; i < size; i++)
   {
      target = array[i];
      j = i - 1;

      /* Here the elements in b/w arrary[0 to i-1]
      which are greater than target are moved
      ahead by 1 position each*/
      while (j >= 0 && array[j] > target)
      {
         array[j + 1] = array[j];
         j = j - 1;
      }
      array[j + 1] = target;
   }
   cout<<"After Insertion sort: \n";
   display(array, size);//Using dispaly function to print
sorted array.
   return 0;
}
```

Java Code

```
// Java Code for implementation of Insertion Sort
class PrepInsta
            {
                        // Main method
  public static void main(String args[])
   {
      int a[] = { 12,11,13,5,6 };

      PrepInsta ob = new PrepInsta();
      ob.sort(a);

      printArray(a);
   }
   /*Function to sort array using insertion sort*/
   void sort(int a[])
   {
      int len = a.length;//calculating the length of the array
      for (int i = 1; i < len; i++)
            {
         int key = a[i];
         int j = i - 1;

         /* Shift elements of a[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
         while (j >= 0 && a[j] > key)
                                                {
            a[j + 1] = a[j];
            j = j - 1;
           }
             a[j + 1] = key;
         }
   }

   /* A utility function to print array of size n*/
   static void printArray(int a[])
   {
      int len = a.length;
      for (int i = 0; i < len; ++i)
         System.out.print(a[i] + " ");
         System.out.println();
   }
}
```

Output:
Before Insertion sort:
5 3 1 9 8 2 4 7
After Insertion sort:
1 2 3 4 5 7 8 9

Advantages and Disadvantages of Insertion Sort
Advantages

1. It is simple, small and easy to write.
2. It doesn't use a lot of overhead.
3. It uses in place sorting, thus O(1) space requirements
4. If data is almost sorted, then it can be very fast approaching O(n) and faster than Merge sort (for sorted data, and small N, else merge sort is faster)
5. Efficient for (quite) small data sets.

Disadvantages

1. Poor average time complexity of $O(n^2)$
2. If there are many elements then it is inefficient
3. Many items needs to be shifted to one insertion

Properties

1. You will encounter the best case if the data is already or nearly sorted
2. It will give worst case results if the array is sorted in opposite order as required

Time Complexity For Insertion Sort

Best

$\Omega(n)$

Average

$\Theta(n^2)$

Worst

$O(n^2)$

Chapter 6. Selection Sort

What is Selection

In this Sorting technique the list is divided into two parts. The first one is the left end and the second one is the right end . The selection Sort is a very simple sorting algorithm.

Steps for Selection Sort in C

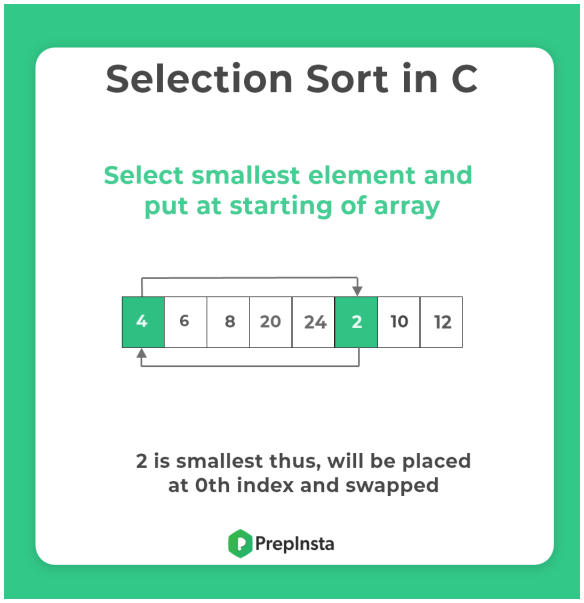
There are the following Step of selection sort algorithms.

Step 1-Select the smallest value in the list.
Step 2-Swap smallest value with the first element of the list.
Step 3-Again select the smallest value in the list (exclude first value).
Step 4- Repeat above step for (n-1) elements until the list is sorted.

Steps for Selection Sort in C

There are the following Step of selection sort algorithms.

Step 1-Select the smallest value in the list.
Step 2-Swap smallest value with the first element of the list.
Step 3-Again select the smallest value in the list (exclude first value).
Step 4- Repeat above step for (n-1) elements until the list is sorted.



Implementation of Selection Sort

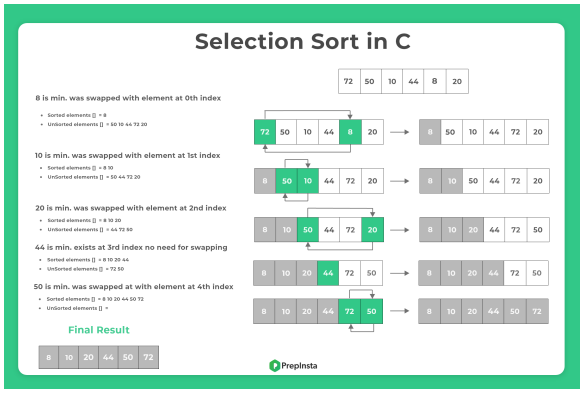
We have been given an unsorted array. which we'll make a sorted array using selection sort. first of all find the smallest value in the array and then swap the smallest value with the starting value.

According to the image below, 8 is the smallest value in this array so 8 is swapped with the first element that is 72.

Similarly, in the next iteration we'll find the next smallest value, excluding the starting value so in this array 10 is the second smallest value, which is to be swapped with 50.

These iterations will continue until we have the largest element to the right most end, along with all the other elements in their correct position.

And then we can finally say that the given array is converted into a sorted array.



Algorithm of selection sort

Algorithm SELECTION(A,N): This algorithm sorts the array A with N elements.

1-[loop]

Repeat Step 2 and 3 for K=0,1,2,..N-2:

2-Call MIN(A,K,N)

3-[Interchange A[K] and A[LOC].]

Set TEMP=A[K],A[K]=A[LOC] and

A[LOC]=TEMP.

[End of Step 1 loop]

4-Exit.

C Code:-

```
// C program for implementation of selection sort
#include <stdio.h>

/* Display function to print values */
void display(int array[], int size)
{
  int i;
  for (i=0; i < size; i++)
  {
    printf("%d ",array[i]);
  }
  printf("\n");
}

// The main function to drive other functions
int main()
{
  int array[] = {5, 3, 1, 9, 8, 2, 4, 7};
  int size = sizeof(array)/sizeof(array[0]);

  printf("This is how array looks before sorting: \n");
  display(array, size);

  int i, j, min_idx,temp;

    // Loop to iterate on array
    for (i = 0; i < size-1; i++)
    {
       // Here we try to find the min element in array
       min_idx = i;
       for (j = i+1; j < size; j++)
       {
       if (array[j] < array[min_idx])
          min_idx = j;
       }
       // Here we interchange the min element with first one
           temp = array[min_idx];
            array[min_idx] = array[i];
            array[i] = temp;}
  printf("This is how array looks after sorting: \n");
  display(array, size);

return 0;
}
```

C++ Code:-

```
#include <iostream>
using namespace std;
//Display function to print values.
void display(int array[], int size)
{
  int i;
  for (i=0; i < size; i++)
  {
    cout<<array[i]<<"\t";
  }
  cout<<"\n";
}

//The main function to drive other functions.
int main()
{
  int array[] = {5, 3, 1, 9, 8, 2, 4, 7};
  int size = sizeof(array)/sizeof(array[0]);

  cout<<"Before sorting: \n";
  display(array, size);//Using dispaly function to print
unsorted array.

  int i, j, min_idx,temp;

    //Loop to iterate elements of array.
    for (i = 0; i < size-1; i++)
    {
        //Here we try to find the min element in the array.
        min_idx = i;
        for (j = i+1; j < size; j++)
        {
        if (array[j] < array[min_idx])
            min_idx = j;
        }
        //Here we interchange the min element with the first
one.
        temp = array[min_idx];
        array[min_idx] = array[i];
        array[i] = temp;}
        cout<<"After sorting: \n";
        display(array, size); //Using dispaly function to print
sorted array.

    return 0;

}
```

Java Code:

```
class PrepInsta
{
            // Main method, responsible for the execution of
the code
  public static void main(String args[])
  {
     PrepInsta ob = new PrepInsta();
     int a[] = {19,17,25,43,15}; //initializing values to the
array
     ob.sort(a); //calling sort method
     System.out.println("Sorted array");
     ob.printArray(a); //calling printArray method
  }

  void sort(int a[])
  {
     int len = a.length;      //calculating the length of the
array
     // One by one move boundary of unsorted subarray
     for (int i = 0; i < len-1; i++)
     {
        // Find the minimum element in unsorted array
        int min = i;
        for (int j = i+1; j < len; j++)
           if (a[j] < a[min])
              min = j;

        // Swap the found minimum element with the first
element
        int temp = a[min];
        a[min] = a[i];
        a[i] = temp;
     }
  }

  // Prints the sorted array
  void printArray(int a[])
  {
     int len = a.length;
     for (int i=0; i<len; ++i)    //printing the sorted array
        System.out.print(a[i]+" ");
     System.out.println();
  }


}
```

Output

This is how array looks before sorting:
5 3 1 9 8 2 4 7
This is how array looks after sorting:
1 2 3 4 5 7 8 9

Performance

The Selection Sort best and worst case scenarios both follow the time complexity format O(n^2) as the sorting operation involves two nested loops. The size of the array again affects the performance.[1]

Strengths

- The arrangement of elements does not affect its performance.
- Uses fewer operations, so where data movement is costly it is more economical

Weaknesses

- The comparisons within unsorted arrays requires O(n^2) which is ideal where n is small

Time Complexity For Selection Sort

Best

$\Omega(n^2)$

Average

$\Theta(n^2)$

Worst

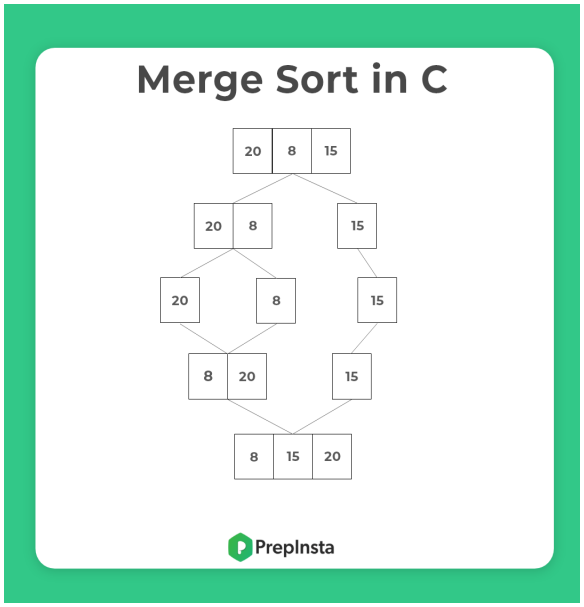
$O(n^2)$

Chapter 7. Merge Sort

How Merge Sort Works

Merge Sort is an example of the divide and conquer approach.It divides the array into equal halves and then combines in a sorted manner.In merge sort the unsorted list is divided into N sublists, each having one element.

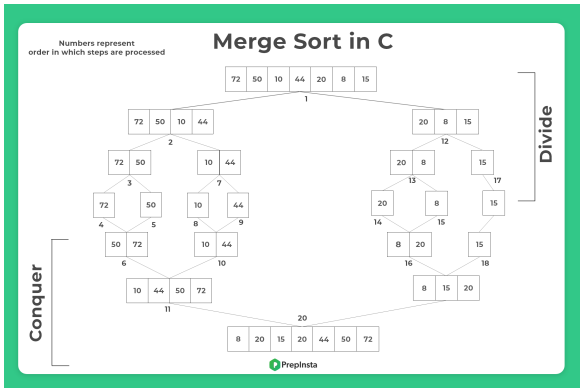
1. The complexity of the merge sort algorithm is O(NlogN) where N is the number of elements to sort.
2. It can be applied to files of any size.
3. In Merge sort all the elements are divided into two sub-array again and again until one element is left.
4. After the completion of the divide and conquer the elements are merged to make a sorted array .
5. This sorting is less efficient and it require more space as compare to other sorting,



Execution of Merge sort

This is an unsorted array and it will make a sorted array by using a merge sort algorithm. In this array it will use the divide technique . First of all the array will be divided into N sub arrays. and the array divides until each has only one element.

In this array each element should be divided into each other.the sorted element will be merged by using conquer technique. and the last element will be merged. And the final array or list is sorted by using the merge sort algorithm.



C Code:-

```
#include <stdio.h>

void mergeSort(int[],int,int);
void merge(int[],int,int,int);

void display(int arr[], int size){
  int i;
  for(i = 0; i < size; i++){
     printf("%d ",arr[i]);
  }
  printf("\n");
}

void main()
{
  int a[10]= {11, 9, 6, 19, 33, 64, 15, 75, 67, 88};
  int i;

  int size = sizeof(a)/sizeof(a[0]);
  display(a, size);

  mergeSort(a,0,size-1);
  display(a, size);
}

void mergeSort(int a[], int strt, int end)
{
  int mid;
  if(strt<end)
  {
     mid = (strt+end)/2;

     mergeSort(a,strt,mid);
     mergeSort(a,mid+1,end);
     merge(a,strt,mid,end);
  }
}

void merge(int a[], int strt, int mid, int end)
{
  int i=strt,j=mid+1,p,index = strt;
  int temp[10];

  while(i<=mid && j<=end)
  {
     if(a[i]<a[j])
     {
        temp[index] = a[i];
        i = i+1;
     }
     else
     {
        temp[index] = a[j];
        j = j+1;
     }

     index++;
  }
  if(i>mid)
  {
     while(j<=end)
     {
        temp[index] = a[j];
        index++;
        j++;
     }
  }
  else
  {
     while(i<=mid)
     {
        temp[index] = a[i];
        index++;
        i++;
     }
  }
  p = strt;
  while(p<index)
  {
     a[p]=temp[p];
     p++;
  }
}
```

C++ Code:-

```
#include <iostream>
using namespace std;

void mergeSort(int[],int,int);
void merge(int[],int,int,int);

//Display function to print values.
```

```
void display(int arr[], int size)
{
   int i;
   for(i = 0; i < size; i++)
   {
      cout<<arr[i]<<"\t";
   }
   cout<<"\n";
}

int main()
{
   int a[10]= {8, 4, 5, 1, 3, 9, 0, 2, 7, 6};
   int i;

   int size = sizeof(a)/sizeof(a[0]);
   cout<<"Before sorting: \n";
   display(a, size);//Using display function to print unsorted
array.
   cout<<"After sorting: \n";
   mergeSort(a,0,size-1);
   display(a, size);//Using display function to print sorted
array.
}
//Dividing the list into two sublist, sorting them and
merging them.
void mergeSort(int a[], int strt, int end)
{
   int mid;
   if(strt<end)
   {
      mid = (strt+end)/2;

      mergeSort(a,strt,mid);//Divide
      mergeSort(a,mid+1,end);//Conqure
      merge(a,strt,mid,end);//Combine
   }
}
//Combining two sublist.
void merge(int a[], int strt, int mid, int end)
{
   int i=strt,j=mid+1,p,index = strt;
   int temp[10];

   while(i<=mid && j<=end)
   {
      if(a[i]<a[j])
      {
         temp[index] = a[i];
         i = i+1;
      }
      else
      {
         temp[index] = a[j];
         j = j+1;
      }

      index++;
   }
   if(i>mid)
   {
      while(j<=end)
      {
         temp[index] = a[j];
         index++;
         j++;
      }
   }
   else
   {
      while(i<=mid)
      {
         temp[index] = a[i];
         index++;
         i++;
      }
   }
   p = strt;
   while(p<index)
   {
      a[p]=temp[p];
      p++;
   }
}
```

Java Code:-

```
//Java Program for Merge Sorting
public class Main {
        public static void display(int[] arr, int size)   //this
function display the array
        {
                for(int i = 0; i < size; i++) {
                        System.out.print(arr[i] + " ");
                }
                System.out.println();
        }

        public static void main(String[] args)   // main
function of the program
    {
                int[] a = {11, 9, 6, 19, 33, 64, 15, 75,
67, 88};
                int i;

                int size = (int)a.length;
                display(a, size);
```

```
                mergeSort(a, 0, size - 1);
                display(a, size);
        }

        static void mergeSort(int[] a, int strt, int end)
//this function apply merging and sorting in the array
    {
                int mid;
                if(strt < end)
        {
                        mid = (strt + end) / 2;

                        mergeSort(a, strt, mid);
                        mergeSort(a, mid + 1, end);
                        merge(a, strt, mid, end);
                }
        }

        static void merge(int[] a, int strt, int mid, int end)
//after sorting this function merge the array
    {
                int i = strt, j = mid + 1, p, index = strt;
                int[] temp = new int[10];

                while(i <= mid && j <= end) {
                        if(a[i] < a[j]) { temp[index] =
a[i]; i = i + 1; } else { temp[index] = a[j]; j = j + 1; }
index++; } if(i > mid) {
                        while(j <= end) {
                                temp[index] = a[j];
                                index++;
                                j++;
                        }
                } else {
                        while(i <= mid) {
                                temp[index] = a[i];
                                index++;
                                i++;
                        }
                }
                p = strt;
                while(p < index) {
                        a[p] = temp[p];
                        p++;
                }
        }
}
```

Output

Input array - 11 9 6 19 33 64 15 75 67 88
Output array - 6 9 11 15 19 33 64 67 75 88

Merge Sort Advantages

1. Fast ! Time complexity : O(N Log N)
2. Reliable, it gives same time complexity in all cases.
3. Tim sort variant of this really powerful (Not important for Placements)

Merge Sort Disadvantages

1. Space Complexity sucks !!!
2. Space Complexity : O(n), most other algorithm have O(1)

Time Complexity For Bubble Sort

Best

Ω(nlogn)

Average

Θ(nlogn)

Worst

O(nlogn)

Chapter 8. Quick Sort

How Quicksort works

Quick sort is an algorithm of the divide and conquer type. That is,the problem of sorting a set is reduced to the problem of sorting two smaller sets.

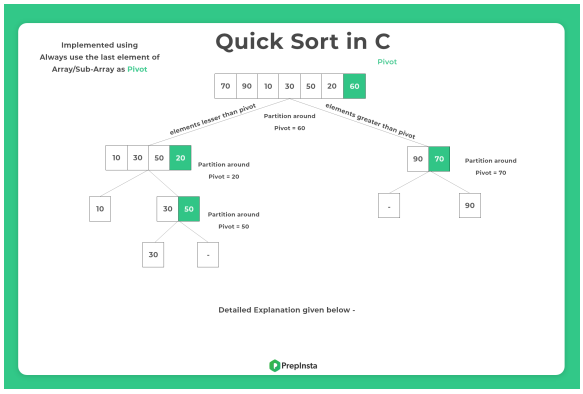
How does it work?
Quick sort works in the following way –

- Choose an item from array called as pivot
- Move all the elements smaller than pivot to left partition
- Move all the elements greater than pivot to the right partition.

Choose a new pivot item in each partition and keep doing the same process again until partitions of one element each aren't formed.

How to choose Pivot?
These are the following ways you can choose pivot –

- First element as pivot.
- Last element as pivot (We use this in our examples & Code)
- Random element as pivot.
- Median element as pivot.



Execution of quick sort
Quick sort works recursively, Partitioning moves all smaller elements to left of pivot and greater to right side of pivot, thus each time two sub array partitions are formed. Thus again, we do partitioning in each of these sub arrays individually.

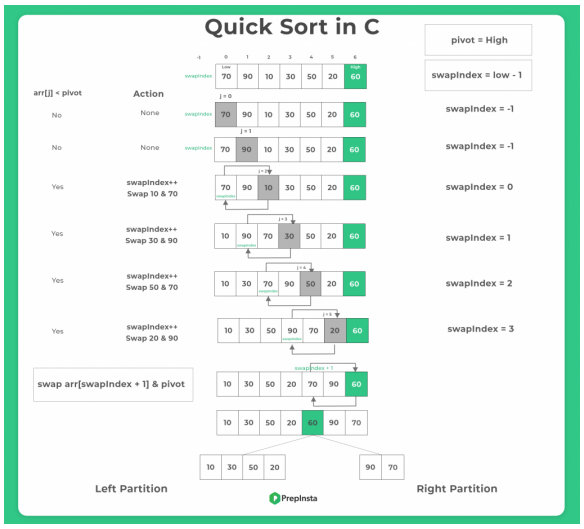
Algorithm for quick sort in C++

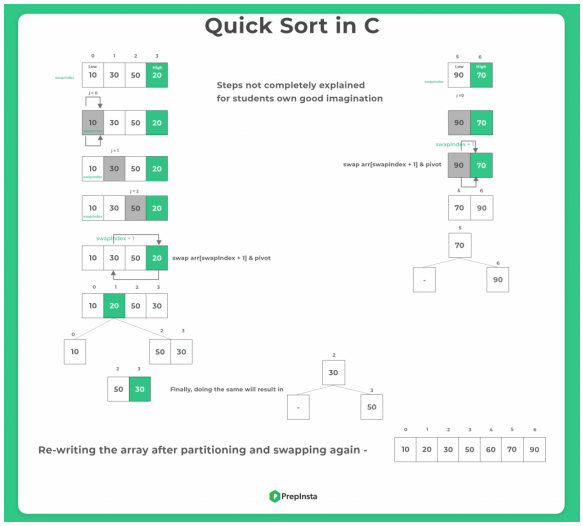
- If p < r then
- q= Partition (A, p, r)
- Quick Sort (A, p, q)
- Quick Sort (A, q + r, r)

Partition Algorithm

PARTITION (A, p, r)

- x ← A[p]
- i ← p-1
- j ← r+1
- while TRUE do
- Repeat j ← j-1
- until A[j] ≤ x
- Repeat i ← i+1
- until A[i] ≥ x
- if i < j
- then exchange A[i] ↔
- A[j]
- else return j





C Code:

```
#include<stdio.h>

// A utility function to swap two elements
void swap(int* x, int* y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}

/* Partition function to do Partition
elements on the left side of pivot elements would be
smaller than pivot
elements on the right side of pivot would be greater than
the pivot
*/
int partition (int array[], int low, int high)
{
  // pivot element selected as right most element in array
each time
  int pivot = array[high];
  int swapIndex  = (low - 1);   //swapping index

  for (int j = low; j <= high- 1; j++)
  {
     // Check if current element is smaller than pivot
element
     if (array[j] < pivot)
     {
        swapIndex ++;   // increment swapping index
        swap(&array[swapIndex], &array[j]);
     }
  }
  swap(&array[swapIndex + 1], &array[high]);
  return (swapIndex + 1);
}

//Recursive function to apply quickSort
void quickSort(int array[], int low, int high)
{
  if (low < high)
  {
    /* indexPI is partitioning index, partition() function will
     return index of partition */
     int indexPI  = partition(array, low, high);

     quickSort(array, low, indexPI  - 1);  // left partition
     quickSort(array, indexPI  + 1, high); // right partition
  }
}

/* Function to display the array */
void display(int array[], int size)
{
  int i;
  for (i=0; i < size; i++)
     printf("%d ", array[i]);
}

//Main function to run the program
int main()
{
  int array[] = {70, 90, 10, 30, 50, 20, 60};
  int size = sizeof(array)/sizeof(array[0]);
  quickSort(array, 0, size-1);
  printf("Array after Quick Sorting: ");
  display(array, size);
  return 0;
}
```

C++ Code:-

```
#include <iostream>
using namespace std;

//Function to swap two elements.
void swap(int* x, int* y)
{
   int temp = *x;
   *x = *y;
   *y = temp;
}

/* Partition function to do Partition
elements on the left side of pivot elements would be
smaller than pivot
elements on the right side of pivot would be greater than
the pivot
*/
```

```
int partition (int array[], int low, int high)
{
   //Pivot element selected as right most element in array
each time.
   int pivot = array[high];
   int swapIndex  = (low - 1);   //swapping index.

   for (int j = low; j <= high- 1; j++)
   {
      //Check if current element is smaller than pivot
element.
      if (array[j] < pivot)
      {
         swapIndex ++;    //increment swapping index.
         swap(&array[swapIndex], &array[j]);
      }
   }
   swap(&array[swapIndex + 1], &array[high]);
   return (swapIndex + 1);
}

//Recursive function to apply quickSort
void quickSort(int array[], int low, int high)
{
   if (low < high)
   {
     /* indexPI is partitioning index, partition() function
will
      return index of partition */
      int indexPI  = partition(array, low, high);

      quickSort(array, low, indexPI  - 1);  //left partition
      quickSort(array, indexPI  + 1, high); //right partition
   }
}

//Function to display the array
void display(int array[], int size)
{
   int i;
   for (i=0; i < size; i++)
      cout<< array[i]<<"\t";
}

//Main function to run the program
int main()
{
   int array[] = {7, 9, 1, 3, 5, 2, 6, 0, 4, 8};
   int size = sizeof(array)/sizeof(array[0]);
   cout<<"Before Sorting: \n";
   display(array, size);
   quickSort(array, 0, size-1);
   cout<<"After Sorting: \n";
   display(array, size);
   return 0;
}
```

Java Code:-

```
/* Java program for Merge Sort */
public class MergeSort {
  void merge(int arr[], int l, int m, int r)
  {
     int n1 = m - l + 1;
     int n2 = r - m;
     int L[] = new int[n1];
     int R[] = new int[n2];
     for (int i = 0; i < n1; ++i)
        L[i] = arr[l + i];
     for (int j = 0; j < n2; ++j)
        R[j] = arr[m + 1 + j];
     int i = 0, j = 0;
     int k = l;
     while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
           arr[k] = L[i];
           i++;
        }
        else {
           arr[k] = R[j];
           j++;
        }
        k++;
     }
     while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
     }
     while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
     }
  }
  void sort(int arr[], int l, int r)
  {
     if (l < r) {
        int m = (l + r) / 2;
        sort(arr, l, m);
        sort(arr, m + 1, r);
        merge(arr, l, m, r);
     }
  }
  static void printArray(int arr[])
  {
     int n = arr.length;
     for (int i = 0; i < n; ++i)
```

```
        System.out.print(arr[i] + " ");
    System.out.println();
  }
  public static void main(String args[])
  {
    int arr[] = { 12, 11, 13, 5, 6, 7 };

    System.out.println("Given Array");
    printArray(arr);

    MergeSort ob = new MergeSort();
    ob.sort(arr, 0, arr.length - 1);

    System.out.println("\nSorted array");
    printArray(arr);
  }
}
```

Quick Sort Performance Analysis

Advantages

1. As the name suggests !!! Is quick as a horse !!

2. Reliable algorithm and used a lot in industry

Disadvantages

- Needs additional spaces for temporary arrays, thus space complexity is O(Log n)
- Very difficult to understand
- It the first element is chosen as pivot by some idiot then it causes worst case complexity of $O(n^2)$

Chapter 9. Counting sort

Counting Sort, is an integer sorting algorithm, is a sorting technique in which we sort a collection of elements based on numeric keys between the specific range. In the counting algorithm we don't compare elements while sorting.it is often used as a subroutine in other sorting algorithms. By counting It operates the no. of objects having each distinct key value, after this it performs some arithmetic to calculate the position of each key value in the output sequence.

It is used as a subroutine in another sorting algorithm for example radix sort. If we compare counting sort to bucket sort then we see that bucket sort requires a large amount of preallocated memory or linked list and dynamic array to hold the sets of items within each bucket, whereas counting sort instead stores a single number per bucket.

Time complexity of counting sort is O((n).

Algorithm of Counting Sort

Counting_Sort(A,B,K)

1-let C[0,k] be a new array

2- for i<-0 to k

3- C[i]<-0

4- for j<-1 to length[A]

5- do C[A[j]] <- C[A[j]]+1

// C[i] now contains the number of elements equal to i.

6- for i<-1 to k

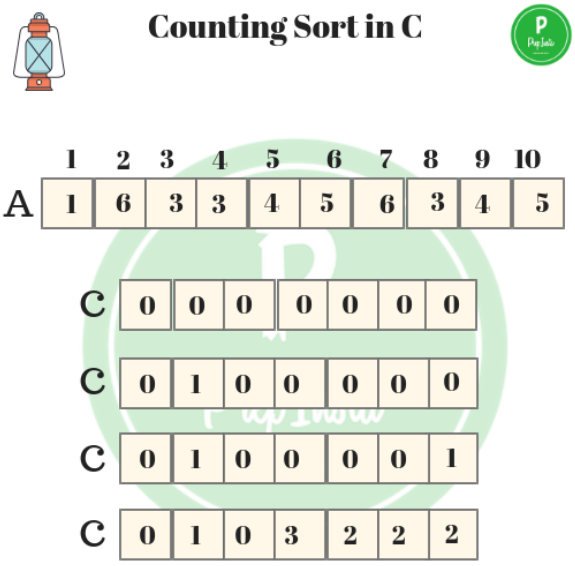
7-do C[i]<- C[i]+C[i-1]

// C[i] now contains the number of elements less than or equal to i.

8- for j<- length[A] down to 1

9-do B[C[A[j]]]<- A[j]

10- C[A[j]]<-C[A[j]]-1.



Execution of Counting Sort

According to algorithm let us take an example

Step 1-

i=0 to 6 since k=6(Largest element in array A)

C[i]<-0

Step 2-

j=1 to 10 since,length[A]=10

C[A[j]]<-C[A[j]]+1

For j=1

C[A[1]]<-C[1]+1= 0+1 =1

C[1]<-1

For j=2

C[A[2]]<-C[6]+1= 0+1 =1

C[6]<-1

similarly for j=5,6,7,8,9,10

Step 3-

For i= 1 to 6

C[i]<-C[i]+C[i-1]

For i=1

C[1]<-C[1]+C[0]

C[1]<-1+0 = 1

For i=2

C[2]<-C[2]+C[1]

C[1]<-1+0 = 1

Similarly for i=4,5,6

Step 4-

For j=10 to 1

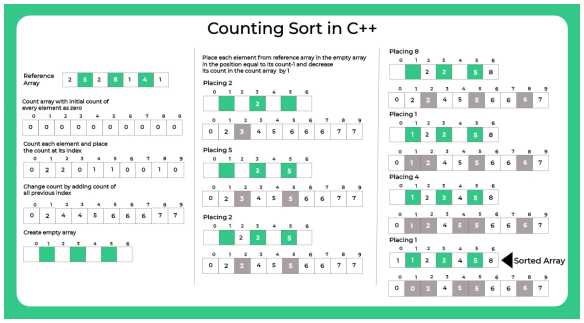
B[C[A[j]]]<-A[j]

C[A[j]]<-C[A[j]]-1

C code-

```
//sorting in c


//counting sort
#include
   //function for counting sort
void counting_sort(int a[], int k, int n)
{
int i, j;
int b[15], c[100];
  for (i = 0; i <= k; i++)
    c[i] = 0;
  for (j = 1; j <= n; j++)
    c[a[j]] = c[a[j]] + 1;
 for (i = 1; i <= k; i++)
   c[i] = c[i] + c[i-1];
 for (j = n; j >= 1; j--)
{
 b[c[a[j]]] = a[j];
 c[a[j]] = c[a[j]] - 1;
}
 printf("The Sorted array is : ");
   // printing the output
 for (i = 1; i <= n; i++)
 printf("%d ", b[i]);
}
 /* End of counting_sort() */
 // main function
 int main()
{
int n, k = 0, a[15], i;
 printf("Enter the number of input : ");
   // user input
  scanf("%d", &n);
  printf("\nEnter the elements to be sorted :\n");
 for (i = 1; i <= n; i++)
{
  scanf("%d", &a[i]);
 if (a[i] > k) {
   k = a[i];
}
}
// function calling
  counting_sort(a, k, n);
  printf("\n");
  return 0;
}
```

C++ code-




```
#include <iostream>
using namespace std;

void countSort(int array[], int size)
{

 int output[10];
 int count[10];
 int max = array[0];

 //Find the largest element of the array
 for (int i = 1; i < size; i++)
 {
  if (array[i] > max)
    max = array[i];
 }

 //Initializing count array with zeros.
 for (int i = 0; i <= max; ++i)
 {
  count[i] = 0;
 }

 //Storing the count of each element
 for (int i = 0; i < size; i++) {
  count[array[i]]++;
 }

 //Storing the cumulative count of each array
 for (int i = 1; i <= max; i++) {
  count[i] += count[i - 1];
 }
```

```
  /*Finding the index of each element of the original array
in count array,
  and place the elements in output array*/
  for (int i = size - 1; i >= 0; i--) {
    output[count[array[i]] - 1] = array[i];
    count[array[i]]--;
  }


  for (int i = 0; i < size; i++) {
    array[i] = output[i];
  }
}

//printing function
void display(int array[], int size)
{
  for (int i = 0; i < size; i++)
    cout << array[i] << "\t";
  cout << endl;
}


int main()
{
  int array[] = {2, 5, 2, 8, 1, 4, 1};
  int n = sizeof(array) / sizeof(array[0]);
  cout<<"Unsorted array\n";
  display(array, n);
  countSort(array, n);
  cout<<"Sorted array\n";
  display(array, n);
}
```

Java code-

```
import java.util.*;

public class Main
{

    static void cSort(int[] arr)
    {
        int max = Arrays.stream(arr).max().getAsInt();
        int min = Arrays.stream(arr).min().getAsInt();
        int range = max - min + 1;
        int c[] = new int[range];
        int output[] = new int[arr.length];
        for (int i = 0; i < arr.length; i++)
        {
            c[arr[i] - min]++;
        }

        for (int i = 1; i < c.length; i++)
        {
            c[i] += c[i - 1];
        }
        for (int i = arr.length - 1; i >= 0; i--)
        {
            output[c[arr[i] - min] - 1] = arr[i];
            c[arr[i] - min]--;
        }

        for (int i = 0; i < arr.length; i++)
        {
            arr[i] = output[i];
        }
    }

    static void display(int[] arr)
    {
        for (int i = 0; i < arr.length; i++)
        {
            System.out.print(arr[i] + " ");
        }
        System.out.println("");
    }
        public static void main(String[] args)
    {
        int[] arr = {-4, -8, 0, -1, 8, 5, 1, 10};
        cSort(arr);
        display(arr);
    }
}
```

Time Complexity

Best O(n + k)

Average O(n + k)

Worst-O(n + k)

Chapter 10. Radix sort-

It is a sorting algorithm that is used to sort elements. Radix sort is the method that many people begin to use when alphabetizing a large list of name or numeric number.Specifically ,

The list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes,where the first class consist of those name that begin with “A”, the second class consists of those name that begin with “B”,and so on.During the second pass, each class is alphabetized according to the second letter of the name.And so on.

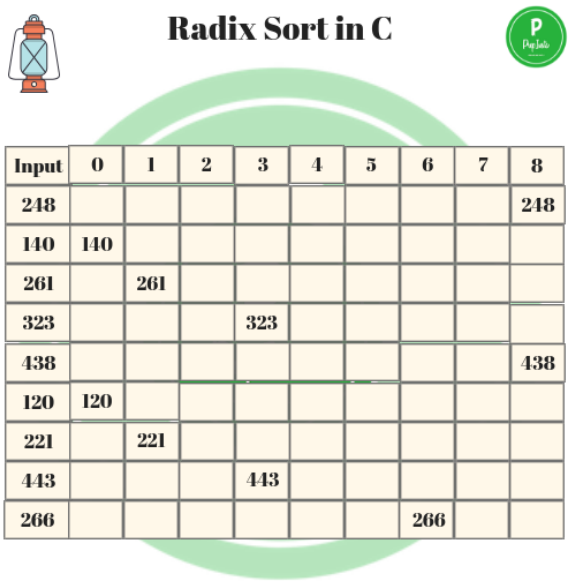
- The time complexity of Radix sort is O(kn) and space complexity of Radix sort is O(k+n).the running time of Radix appears to be better than Quick Sort for a wide range of input numbers.

Implementation of Radix sort

Let us take an example of radix sort

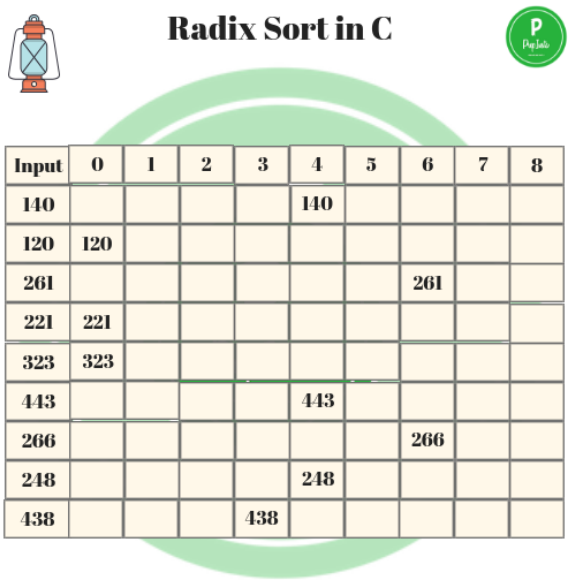
Given to a list ,the numbers would be sorted in three phases. the list is (248,140,261,323,438,120,221,443,266).

(A)- In the first phase, the once digits are sorted into a list.. The numbers are collected box by box ,from pocket 8 to 0.(Note 261 will now be at the bottom of the pile and 438 at the top of the pile.) The cards are now rein put to the sorter.



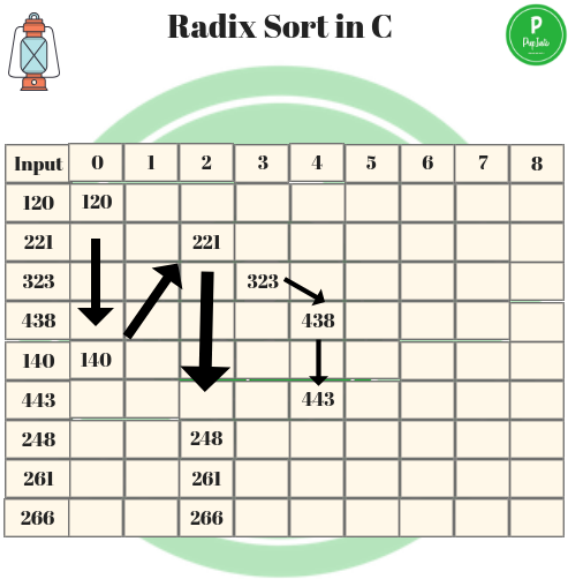
Input	0	1	2	3	4	5	6	7	8
248									248
140	140								
261		261							
323				323					
438									438
120	120								
221		221							
443				443					
266							266		

(B)- In the second phase, the tens digits are sorted into pockets,Again the card are collected pocket by pocket and rein put to the sorter.



Input	0	1	2	3	4	5	6	7	8
140					140				
120	120								
261							261		
221	221								
323	323								
443					443				
266							266		
248					248				
438				438					

In the last phase and third phase, we will notify that the hundred digit are sorted into pockets.



Input	0	1	2	3	4	5	6	7	8
120	120								
221			221						
323				323					
438					438				
140	140								
443					443				
248			248						
261			261						
266			266						

So the final list is by ascending order to sorted digits or number is (120,140, 221,248,261,266,323,438,443)

Algorithm of Radix sort

1- for i<-1 to d do

2-use a stable sort to array A on digit i

//counting sort will do the job

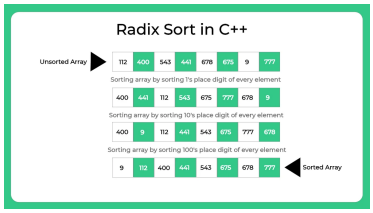
The code for radix sort assumes that each element in the n-element array A has d- digits,where digit 1 is the lowest -order digit and d is the highest-order digit.

C code:

```
#include <stdio.h> //including library files
int max(int a[]); //declaring functions
void radix_sort(int a[]); //declaring functions
void main()   //defining main()
{
  int i;
  int a[10]={92,106,365,90,33,19,72,56,45,12};
//initializing array with values
  radix_sort(a);
  printf("\n The sorted list is: \n");  //printing the sorted list
  for(i=0;i<10;i++)
     printf(" %d\t", a[i]);
}

int max(int a[])  //giving the definition to max()
{
  int max=a[0], i;  //finiding the largest number
  for(i=1;i<10;i++)
  {
     if(a[i]>max)
     max = a[i];
  }
  return max;
}
void radix_sort(int a[])  //providing definition to radix_sort
{
  int bucket[10][10], bucket_count[10]; //declaring
variables
  int i, j, k, remainder, NOP=0, divisor=1, max, pass;
//declaring variables
  max =  max(a);
  while( max>0)
  {
     NOP++;
      max/=10;
  }
  for(pass=0;pass<NOP;pass++) // Initialization of the
buckets
  {
```

```
    for(i=0;i<10;i++)
    bucket_count[i]=0;
    for(i=0;i<10;i++)
    {
       // sort the numbers according to the digit at passth
place
       remainder = (a[i]/divisor)%10;
       bucket[remainder][bucket_count[remainder]] = a[i];
       bucket_count[remainder] += 1;
    }
    // collect the numbers after PASS pass
    i=0;
    for(k=0;k<10;k++)
    {
       for(j=0;j<bucket_count[k];j++)
       {
          a[i] = bucket[k][j];
          i++;
       }
    }
    divisor *= 10;

}
}
```

C++ code




```
#include <iostream>
using namespace std;

//Function to get the largest element from an array
int getMax(int array[], int n)
{
  int max = array[0];
  for (int i = 1; i < n; i++)
    if (array[i] > max)
      max = array[i];
  return max;
}

//Using counting sort to sort the elements in the basis of
significant places
void countSort(int array[], int size, int place)
{
  const int max = 10;
  int output[size];
  int count[max];

  for (int i = 0; i < max; ++i)
    count[i] = 0;

  //Calculate count of elements
  for (int i = 0; i < size; i++)
    count[(array[i] / place) % 10]++;

  //Calculating cumulative count
  for (int i = 1; i < max; i++)
    count[i] += count[i - 1];

  //Placing the elements in sorted order
  for (int i = size - 1; i >= 0; i--)
  {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]--;
  }

  for (int i = 0; i < size; i++)
    array[i] = output[i];
}

//Main function to implement radix sort
void radixsort(int array[], int size)
{
  //Getting maximum element
  int max = getMax(array, size);

  //Applying counting sort to sort elements based on place
value.
  for (int place = 1; max / place > 0; place *= 10)
    countSort(array, size, place);
}

//Printing an array
void display(int array[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    cout << array[i] << "\t";
  cout << endl;
}

int main()
{
  int array[] = {112, 400, 543, 441, 678, 675, 9, 777};
  int n = sizeof(array) / sizeof(array[0]);
  cout<<"Before sorting \n";
  display(array, n);
  radixsort(array, n);
  cout<<"After sorting \n";
  display(array, n);
}
```

Output:
Before sorting
112 400 543 441 678 675
9 777
After sorting
9 112 400 441 543 675
678 777

Java code-

```
//Radix sort Java implementation
import java.io.*;
import java.util.*;

class PrepInsta
	{
		/*Main Method to check for
above function*/
	>public static void main (String[] args)
	 {
		int a[] = {17, 45, 75, 90, 82,
24, 12, 60};
		int len = a.length;
		radixsort(a,len);
		print(a,len);
	 }

	// A utility function to get maximum value in a[]
	static int getMax(int a[], int n)
	{
		int mx = a[0];
		for (int i = 1; i < len; i++)
			if (a[i] > mx)
				mx = a[i];
		return mx;
	}

	// A function to do counting sort of arr[]
according to
	// the digit represented by exp.
	static void countSort(int a[], int len, int exp)
	{
		int output[] = new int[len]; // output
array
		int i;
		int count[] = new int[10];
		Arrays.fill(count,0);

		// Store count of occurrences in count[]
		for (i = 0; i < len; i++)
			count[ (a[i]/exp)%10 ]++;

		// Change count[i] so that count[i] now
contains
		// actual position of this digit in
output[]
		for (i = 1; i < 10; i++)
			count[i] += count[i - 1];

		// Build the output array
		for (i = len - 1; i >= 0; i--)
		{
			output[count[ (a[i]/exp)%10 ]
- 1] = a[i];
			count[ (a[i]/exp)%10 ]--;
		}

		// Copy the output array to arr[], so that
arr[] now
		// contains sorted numbers according to
curent digit
		for (i = 0; i < len; i++)
			a[i] = output[i];
	}

	// The main function to that sorts arr[] of size n
using
	// Radix Sort
	static void radixsort(int a[], int len)
	{
		// Find the maximum number to know
number of digits
		int m = getMax(a, len);

		// Do counting sort for every digit. Note
that instead
		// of passing digit number, exp is
passed. exp is 10^i
		// where i is current digit number
		for (int exp = 1; m/exp > 0; exp *= 10)
			countSort(a, len, exp);
	}

	// A utility function to print an array
	static void print(int a[], int len)
	{
		for (int i=0; i<len; i++)
			System.out.print(a[i]+" ");
	}

}
```

Time complexity of radix sort
BEST- O(d(n + k))
Average- O(d(n + k))
Worst- O(d(n + k))

Chapter 11. Heap sort

A heap is a complete binary tree which is represented using array or sequential representation.

It is a special balanced binary tree data structure where the root node is compared with its children and arranged accordingly.Normally in max heap parent node always has a value greater than child node.

The worst and Average complexity of heap sort is O(n log2 n).

- From the given array, build the initial max heap.
- Interchange the root element with the last element.
- Use repetitive downward operation from the root node to rebuild the heap of size one less than the starting.

Algorithm of heap sort

MAX-HEAPIFY(A,i)

1- i<-left[i]

2- r<-right[i]

3- if l<heap-size[A]and A[l]>A[i]

4- then largest<-1

5- else largest<-i

6- if r<heap-size[A] and A[r]>A[largest]

7- then largest<-r

8- if largest!=i

9- then exchange A[i]<->A[largest]

10- MAX-HEAPIFY[A,largest]

HEAP-SORT(A)

1- BUILD-MAX-HEAP(A)

2- for i<-length[A] down to 2

3- do exchange A[1]<-> heap-size[A]-1

4- heap-size[A]<-heap-size[A]-1

5- MAX-HEAPIFY(A,1)

BUILD-MAX-HEAP(A)

1- heap-size[A]<-length[A]

2- for i<-(length[A]/2) down to 1 do

3- MAX-HEAPIFY(A,i)

Implementation of heap sort

As we know that heap sort is based on the tree structure.

so we will take an example of heap sort based on the tree.

Originally the given array is :[6,14,3,25,2,10,20,7,6]

First we call Build-max heap

heap size[A]=9

so,i=4 to 1, call MAX HEAPIFY (A,i)

i.e., first we call MAX HEAPIFY (A,4)

A[l]=7, A[i]=A[4] = 25,A[r] = 6

l<- left[4] = 8

r<-right[4] = 9

8<9 and 7>25 (false)

Then,largest <-4

9<9 and 6>25(false)

Then,largest = 4

A[i]<-> A[4]

Now call MAX HEAPIFY (A,2)

the A[length] =9

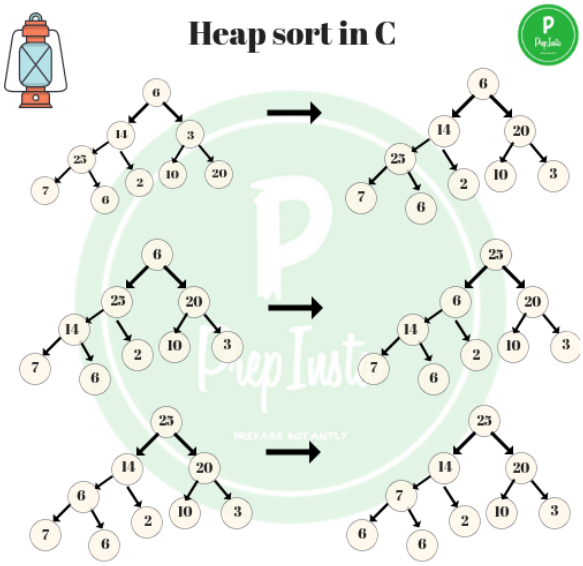
So, i=A/2 = 9/2 =4

i=4

since the parent node is greater than its child node.

If the parent node is greater than the child node then the parent node replaces it with the lesser child node.

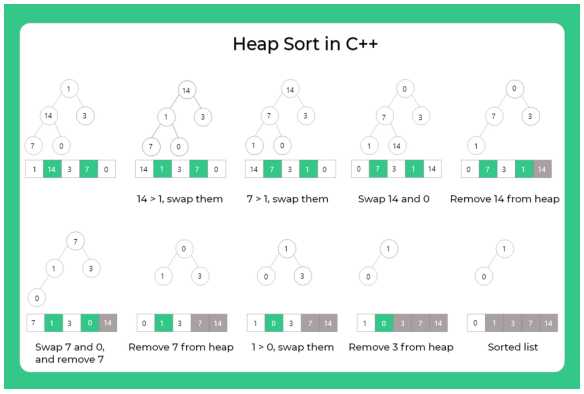
similarly for i=3,2,1 we get the following heap tree.



C code:

```
#include<stdio.h> // including library files
int temp;

void heapify(int arr[], int size, int i)//declaring functions
{
int max = i;
int left = 2*i + 1;
int right = 2*i + 2;

if (left < size && arr[left] >arr[max])
max= left;

if (right < size && arr[right] > arr[max])
max= right;

if (max!= i)
{
// performing sorting logic by using temporary variable
temp = arr[i];
arr[i]= arr[max];
arr[max] = temp;
heapify(arr, size, max);
}
}

void heapSort(int arr[], int size)// providing definition to heap sort
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
heapify(arr, size, i);
for (i=size-1; i>=0; i--)
{
// swaping logic
temp = arr[0];
arr[0]= arr[i];
arr[i] = temp;
heapify(arr, i, 0);
}
}

void main() // defining main()
{
int arr[] = {58, 134, 3, 67, 32, 89, 15, 10,78, 9};
// array initializing with their elements.
int i;
int size = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, size);

printf("printing sorted elements\n"); // printing the sorted array
for (i=0; i<size; ++i)
printf("%d\n",arr[i]);
}
```

C++ code:




```
#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i)
```

```
{
   int largest = i;
   int l = 2*i + 1;
   int r = 2*i + 2;

   //If left child is larger than root
   if (l < n && arr[l] > arr[largest])
      largest = l;

   //If right child largest
   if (r < n && arr[r] > arr[largest])
      largest = r;

   //If root is nor largest
   if (largest != i)
   {
      swap(arr[i], arr[largest]);

      //Recursively heapifying the sub-tree
      heapify(arr, n, largest);
   }
}

void heapSort(int arr[], int n)
{

   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);

   //One by one extract an element from heap
   for (int i=n-1; i>=0; i--)
   {
      //Moving current root to end
      swap(arr[0], arr[i]);

      //Calling max heapify on the reduced heap
      heapify(arr, i, 0);
   }
}

//Function to print array
void display(int arr[], int n)
{
   for (int i = 0; i < n; i++)
   {
      cout << arr[i] << "\t";
   }
   cout << "\n";
}


int main()
{
   int arr[] = {1, 14, 3, 7, 0};
   int n = sizeof(arr)/sizeof(arr[0]);
   cout << "Unsorted array  \n";
   display(arr, n);

   heapSort(arr, n);

   cout << "Sorted array  \n";
   display(arr, n);
}
```

Java code:

```
// Java program for implementation of Heap Sort
public class PrepInsta
{
   //Main() for the execution of the program
   public static void main(String args[])
     {
       int a[] = {12, 11, 13, 5, 6, 7};
       int len = a.length;

        PrepInsta ob = new PrepInsta();
        ob.sort(a);

        System.out.println("Sorted array is");
        printArray(a);
     }
       public void sort(int a[])
          {
            int len = a.length;

           // Build heap (rearrange array)
           for (int i = len / 2 - 1; i >= 0; i--)
           heapify(a, len, i);

           // One by one extract an element from heap
           for (int i=len-1; i>=0; i--)
                {
                 // Move current root to end
                  int temp = a[0];
                  a[0] = a[i];
                  a[i] = temp;

                  // call max heapify on the reduced heap
                  heapify(a, i, 0);
                  }
            }

      // To heapify a subtree rooted with node i which is
      // an index in arr[]. n is size of heap
     void heapify(int a[], int len, int i)
       {
         int largest = i; // Initialize largest as root
         int l = 2*i + 1; // left = 2*i + 1
```

```
        int r = 2*i + 2; // right = 2*i + 2

         // If left child is larger than root
         if (l < len && a[l] > a[largest])
         largest = l;

        // If right child is larger than largest so far
         if (r < len && a[r] > a[largest])
         largest = r;

        // If largest is not root
         if (largest != i)
              {
                int swap = a[i];
                a[i] = a[largest];
                a[largest] = swap;

               // Recursively heapify the affected sub-tree
                 heapify(a, len, largest);
               }
            }

       /* A utility function to print array of size n */
       static void printArray(int a[])
            {
               int len = a.length;
               for (int i=0; i<len; ++i)
               System.out.print(a[i]+" ");
               System.out.println();
            }

    }
```

Output:
Unsorted array
1 14 3 7 0
Sorted array
0 1 3 7 14

Time complexity of heap sort

Best - O(nlogn)
Average - O(nlogn)
Worst - O(nlogn)

Chapter 12. Stacks

Introduction to Stacks in Data Structures

Stack is one of the basic linear Data structures that we use for storing our data. Data in a stack is stored in a serialized manner. One important thing about using a Stack is that the data first entered in the stack will be at the last of the stack. This is one of the reasons why we also called Stack a LIFO Data Structure, i.e; Last in First Out. We'll be discussing more about this feature later in this article.

Basic Terminology of Stack

Before we move on further with understanding the Stack Data Structure, we need to learn about the basic terminology that is associated with this data structure so that understanding stack will be a little easy for us.

The basic terminology that we will be using in Stack is :

1. Top – This refers to the topmost element of the stack, or in other words the element last entered in the stack.
2. Push () – This is one of the operations that we can perform on stack, for inserting data in this data structure.
3. Pop () – This operation deals with deleting the data from the stack. It deletes the top-most data from the stack.
4. Peek () – This operation helps us in looking at the topmost element of the data without removing it from the stack.

A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space for push operation, the stack is then considered to be in an overflow state.

Below is simple representation of a stack with push and pop operations –

Applications of Stacks in Data Structures

There are a lot of places where we use stack without even realizing it. Let's see some of the most common uses of stack data structure.

Real life examples of Stack

1. Stacking up dishes after washing them.
2. Deck of cards
3. Stacking a pile of boxes in our store rooms.
4. Libraries stack pile of books and articles.

Some Coding/Programming Applications of Stack

1. Recursion
2. Backtrackking
3. Memory managament
4. Process Managament

Advantages of Using Stack over other data structures

1.Manages the data in a Last In First Out(LIFO) method which is not possible with Linked list and array.

2. When a function is called the local variables are stored in a stack, and it is automatically destroyed once returned.

3. A stack is used when a variable is not used outside that function.

4. It allows you to control how memory is allocated and deallocated.

5. Stack automatically cleans up the object.

6. Not easily corrupted

7. Variables cannot be resized.

Disadvantages of Using Stack over other data structures

1. Stack memory is very limited.

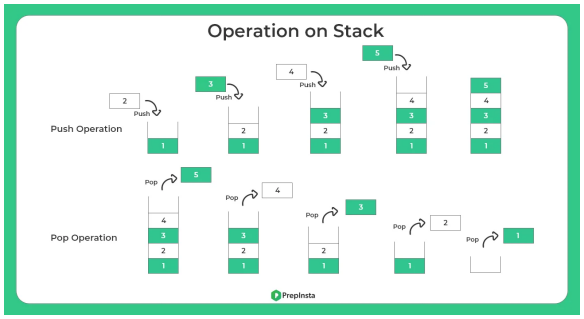
2. Creating too many objects on the stack can increase the risk of stack overflow.

3. Random access is not possible.

4. Variable storage will be overwritten, which sometimes leads to undefined behavior of the function or program.

5. The stack will fall outside of the memory area, which might lead to an abnormal termination.

Operations on Stack



There are many operations that can be used to manipulate a stack. A stack is a data structure in which the operations are executed in constant time. There are a number of operations that can be performed on a stack, but there are two major operations that are used to manipulate a stack.

These operations are:

1. *Push()*

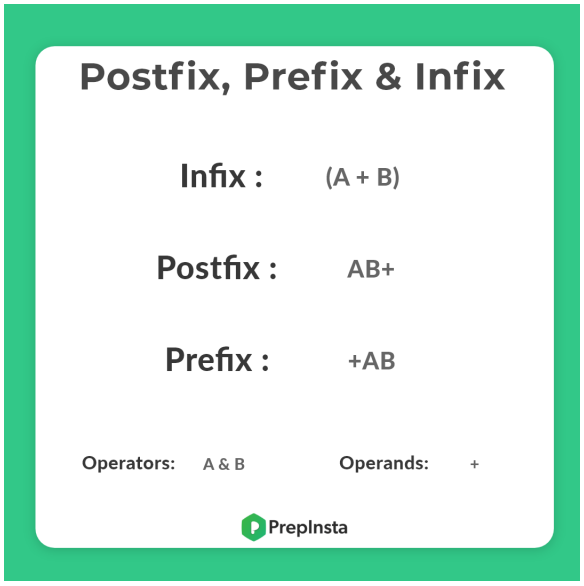
2. Pop()

Implementation of pop and push in C:

```
#include<stdio.h>


#include<stdlib.h>

#define size 10       //size of the stack

int top_of_stack=-1,s[size];

void push();
void pop();
void show();

void main()
{
        int option;
```

```
	while(1)
	{
		printf("\nSelect from the following
options- ");

printf("\n\n1.Push\n2.Pop\n3.Show\n4.Exit");
		printf("\n\nEnter the operation you
want to execute: ");
		scanf("%d",&option);

		switch(option)
		{
			case 1: push();
				break;
			case 2: pop();
				break;
			case 3: show();
				break;
			case 4: exit(0);

			default: printf("\nInvalid
option. Please select a Valid operation");
		}
	}
}

void push()
{
	int value;

	if(top_of_stack == size-1)
	{
		printf("\nOperation failed. Stack is
full.");
	}
	else
	{
		printf("\nEnter the value you want to
push in the stack:");
		scanf("%d",&value);
		top_of_stack = top_of_stack + 1;
		s[top_of_stack] = value;
	}
}

void pop()
{
	if(top_of_stack == -1)
	{
		printf("\nOperation failed. Stack is
empty");
	}
	else
	{
		printf("\nPopped element is:
%d",s[top_of_stack]);
		top_of_stack=top_of_stack-1;
	}
}

void show()
{
	int i;

	if(top_of_stack==-1)
	{
		printf("\nStack is empty");
	}
	else
	{
		printf("\nElements of the stack are:\n");
		for(i = top_of_stack; i>=0; --i)
			printf("%d\n",s[i]);
	}
}
```

Infix Prefix Postfix Conversion (Stack)



Introduction

- Operation : Any Expression of algebraic format (Example : A + B)
- Operands : A and B or 5 & 6 are operands
- Operators : +. -, %,*,/ etc are operators

Infix Notation

Infix is the day to day notation that we use of format A + B type. The general form can be classified as *(a op b)* where *a* and *b* are operands(variables) and *op* is Operator.

- Example 1 : A + B
- Example 2 : A * B + C / D

Postfix Notation

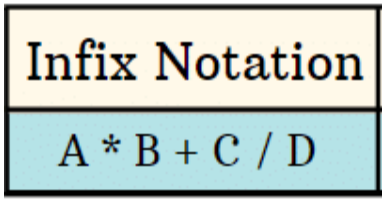
Postfix is notation that compiler uses/converts to while reading left to right and is of format *AB+* type. The general form can be classified as *(ab op)* where *a* and *b* are operands(variables) and *op* is Operator.

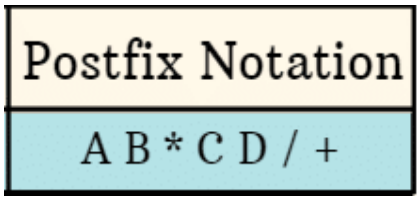
- Example 1 : AB+
- Example 2 : AB*CD/+

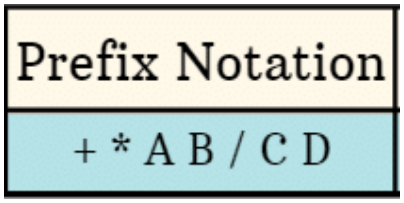
Prefix Notation

Prefix is notation that compiler uses/converts to while reading right to left (some compilers can also read prefix left to right) and is of format *+AB* type. The general form can be classified as *(op ab)* where *a* and *b* are operands(variables) and *op* is Operator.

- Example 1 : +AB
- Example 2 : +*AB/CD







Why are Postfix/Prefix Expressions faster than Infix?
For Infix Expression which is format A+B*C, if the compiler is reading left to right then it can't evaluate A+B first until it has read whole expression and knows expression is actually A + (B*C) i.e. B * C needs to be implemented first

Postfix for above infix is ABC*+. Now, as soon as the compiler sees two operands followed by an operator it can implement it without caring for precedence.

- Assume ABC*+
- ABC*+ (BC* is implemented as B*C and result is put back)
- AX+ (Assuming X stores result of BC* i.e. B*C)
- Now finally AX+ can be implemented as A+X

Note
Compiler converts our Infix Expression into postfix or Prefix as expressions are operated as stacks and postfix and prefix are faster to implement as the compiler doesn't need to care about precedence at all.

Compiler converts the infix expression to postfix/prefix at compile time, so at runtime your calculations are always happening in post-prefix. A website's code may be compiled once a week but it may need to run 1 million times a week.
Which is why we prefer that runtime execution should be as fast as possible. If calculations happen faster than we have optimized our page load time hugely right?

Problem

- Infix: a + b * c + d can be written as a + (b * c) + d
- Now, for all + – / * associativity is left to right we will write it as
- (a + (b * c)) + d and thus further ((a + (b * c)) + d)
- Solving and converting innermost bracket to postfix
- Step 1 –((a + bc*)+ d)
- Step 2 – Consider bc* as separate operand x the innermost bracket now looks like ((a + x)+ d)
    - Applying postfix it looks like – (ax+ + d)replacing x here (abc*+ + d)
- Step 3 – Consideringabc*+as separate operand z, the final bracket looks like – (z + d)the result would be zd+
    - replacing z value = abc*+d+

Algorithm (For Code/Manual Calculation)

First Start scanning the expression from left to right
If the scanned character is an operand, output it, i.e. print it

Else

If the precedence of the scanned operator is higher than the precedence of the operator in the stack(or stack is empty or has'(‘), then push operator in the stack

Else, Pop all the operators that have greater or equal precedence than the scanned operator. Once you pop them push this scanned operator. (If we see a parenthesis while popping then stop and push scanned operator in the stack)

If the scanned character is an ‘(‘, push it to the stack.
If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
Now, we should repeat steps 2 – 6 until the whole infix i.e. whole characters are scanned.

Print output

Do the pop and output (print) until stack is not empty

Infix to Prefix

Problem (This is how to convert manually for MCQ Question in the exam)

- Infix: (a / b + c) - (d + e * f) can be written as ((a / b) + c) - (d + (e * f))
- Now, we have done the above according to associativity
- Solving and converting innermost bracket to prefix
- Step 1 –(/ab + c) - (d + *ef)
- Step 2 – Consider /ab and *ef as separate operand x and y
- the innermost bracket now looks like (x + c) - (d + y)
    - Applying prefix it looks like – (+xc - +dy)replacing x and y here (+/abc - +d*ef)
- Step 3 – Considering+/abc and+d*efas separate operand z and w, the final bracket looks like – (z - w)the result would be -zw

- replacing z and w value = -+/abc+d*ef

Algorithm (For Code/Manual Calculation)

Given Infix - ((a/b)+c)-(d+(e*f))

Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.

Step 2: Obtain the postfix expression of the expression obtained from Step 1.

Step 3: Reverse the postfix expression to get the prefix expression

This is how you convert manually for theory question in the exam

1. String after reversal –))f*e(+d(-)c+)b/a((
2. String after interchanging right and left parenthesis – ((f*e)+d)-(c+(b/a))
3. Apply postfix – Below is postfix Reverse Postfix Expression
4. Final prefix: -+/abc+d*ef

Implementation of Infix expression to Postfix expression

```
#include<stdio.h>
char s[100];
int top_of_stack = -1;
void push(char a)
{
  s[++top_of_stack] = a;
}
char pop()
{
  if(top_of_stack == -1)
    return -1;
  else
    return s[top_of_stack--];
}

int precedence(char a)
{
  if(a == '(')
    return 0;
  if(a == '+' || a == '-')
    return 1;
  if(a == '*' || a == '/')
    return 2;
}

void main()
{
  char expression[20];
  char *exp, a;
  printf("Enter the expression you want to convert:  ");
  scanf("%s",expression);
  exp = expression;
  while(*exp != '\0')
  {
    if(isalnum(*exp))
```

```
        printf("%c",*exp);
    else if(*exp == '(')
        push(*exp);
    else if(*exp == ')')
    {
        while((a = pop()) != '(')
            printf("%c", a);
    }
    else
    {
        while(precedence(s[top_of_stack]) >=
precedence(*exp))
            printf("%c",pop());
        push(*exp);
    }
    exp= exp + 1;
  }
  while(top_of_stack != -1)
  {
    printf("%c",pop());
  }
}
```

Infix to postfix conversion code in C:

// C Program for Implementation of stack (array) using structure

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// A structure to represent a stack
struct Stack {
   int top;
   int maxSize;
   // we are storing string in integer array, this will not give
error
   // as values will be stored in ASCII and returned in
ASCII thus, returned as string again
   int* array;
};

struct Stack* create(int max)
{
   struct Stack* stack = (struct Stack*)malloc(sizeof(struct
Stack));
   stack->maxSize = max;
   stack->top = -1;
   stack->array = (int*)malloc(stack->maxSize *
sizeof(int));
   return stack;
}

// Checking with this function is stack is full or not
// Will return true is stack is full else false
//Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
   if(stack->top == stack->maxSize - 1){
      printf("Will not be able to push maxSize reached\n");
   }
   // Since array starts from 0, and maxSize starts from 1
   return stack->top == stack->maxSize - 1;
}

// By definition the Stack is empty when top is equal to -1
// Will return true if top is -1
int isEmpty(struct Stack* stack)
{
   return stack->top == -1;
}

// Push function here, inserts value in stack and increments
stack top by 1
void push(struct Stack* stack, int item)
{
   if (isFull(stack))
      return;
   stack->array[++stack->top] = item;
}
```

```
// Function to remove an item from stack.  It decreases top
by 1
int pop(struct Stack* stack)
{
   if (isEmpty(stack))
      return INT_MIN;
   return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
   if (isEmpty(stack))
      return INT_MIN;
   return stack->array[stack->top];
}

// A utility function to check if the given character is
operand
int checkIfOperand(char ch)
{
   return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <=
'Z');
}

// Fucntion to compare precedence
// If we return larger value means higher precedence
int precedence(char ch)
{
   switch (ch)
   {
   case '+':
   case '-':
      return 1;

   case '*':
   case '/':
      return 2;

   case '^':
      return 3;
   }
   return -1;
}

// The driver function for infix to postfix conversion
int covertInfixToPostfix(char* expression)
{
   int i, j;

   // Stack size should be equal to expression size for safety
   struct Stack* stack = create(strlen(expression));
   if(!stack) // just checking is stack was created or not
      return -1 ;

   for (i = 0, j = -1; expression[i]; ++i)
   {
      // Here we are checking is the character we scanned is
operand or not
      // and this adding to to output.
      if (checkIfOperand(expression[i]))
         expression[++j] = expression[i];

      // Here, if we scan character '(', we need push it to the
stack.
      else if (expression[i] == '(')
         push(stack, expression[i]);

      // Here, if we scan character is an ')', we need to pop
and print from the stack
      // do this until an '(' is encountered in the stack.
      else if (expression[i] == ')')
      {
         while (!isEmpty(stack) && peek(stack) != '(')
            expression[++j] = pop(stack);
         if (!isEmpty(stack) && peek(stack) != '(')
            return -1; // invalid expression
         else
            pop(stack);
      }
      else // if an operator
      {
         while (!isEmpty(stack) &&
precedence(expression[i]) <= precedence(peek(stack)))
            expression[++j] = pop(stack);
         push(stack, expression[i]);
      }

   }

   // Once all initial expression characters are traversed
   // adding all left elements from stack to exp
   while (!isEmpty(stack))
      expression[++j] = pop(stack);

   expression[++j] = '\0';
   printf( "%s", expression);
}

int main()
{
char expression[] = "((a+(b*c))-d)";
   covertInfixToPostfix(expression);
   return 0;
}
```

Infix to prefix conversion code in C:

```
#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// A structure to represent a stack

struct Stack {

  int top;

  int maxSize;

  // we are storing string in integer array, this will not give
error

  // as values will be stored in ASCII and returned in ASCII
thus, returned as string again

  int* array;

};


struct Stack* create(int max)

{

  struct Stack* stack = (struct Stack*)malloc(sizeof(struct
Stack));

  stack->maxSize = max;

  stack->top = -1;

  stack->array = (int*)malloc(stack->maxSize *
sizeof(int));

  return stack;

}


// Checking with this function is stack is full or not

// Will return true is stack is full else false

//Stack is full when top is equal to the last index

int isFull(struct Stack* stack)

{

  if(stack->top == stack->maxSize - 1){

     printf("Will not be able to push maxSize reached\n");

  }

  // Since array starts from 0, and maxSize starts from 1

  return stack->top == stack->maxSize - 1;

}

// By definition the Stack is empty when top is equal to -1

// Will return true if top is -1

int isEmpty(struct Stack* stack)

{

  return stack->top == -1;

}


// Push function here, inserts value in stack and increments
stack top by 1

void push(struct Stack* stack, int item)

{

  if (isFull(stack))

     return;

  stack->array[++stack->top] = item;

}


// Function to remove an item from stack.  It decreases top
by 1
```

```
int pop(struct Stack* stack)
{
  if (isEmpty(stack))
     return INT_MIN;
  return stack->array[stack->top--];
}
// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
  if (isEmpty(stack))
     return INT_MIN;
  return stack->array[stack->top];
}
// A utility function to check if the given character is
operand
int checkIfOperand(char ch)
{
  return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}
// Fucntion to compare precedence
// If we return larger value means higher precedence
int precedence(char ch)
{
  switch (ch)
  {
  case '+':
  case '-':
     return 1;
  case '*':
  case '/':
     return 2;
  case '^':
     return 3;
  }
  return -1;
}

// The driver function for infix to postfix conversion
int getPostfix(char* expression)
{
  int i, j;
  // Stack size should be equal to expression size for safety
  struct Stack* stack = create(strlen(expression));
  if(!stack) // just checking is stack was created or not
     return -1 ;

  for (i = 0, j = -1; expression[i]; ++i)
  {
     // Here we are checking is the character we scanned is
operand or not
     // and this adding to to output.
     if (checkIfOperand(expression[i]))
        expression[++j] = expression[i];
```

```
    // Here, if we scan character ‘(‘, we need push it to the
stack.
    else if (expression[i] == '(')
        push(stack, expression[i]);

    // Here, if we scan character is an ‘)’, we need to pop
and print from the stack
    // do this until an ‘(‘ is encountered in the stack.
    else if (expression[i] == ')')
    {
        while (!isEmpty(stack) && peek(stack) != '(')
            expression[++j] = pop(stack);
        if (!isEmpty(stack) && peek(stack) != '(')
            return -1; // invalid expression
        else
            pop(stack);
    }
    else // if an opertor
    {
        while (!isEmpty(stack) &&
precedence(expression[i]) <= precedence(peek(stack)))
            expression[++j] = pop(stack);
        push(stack, expression[i]);
    }

  }
  // Once all inital expression characters are traversed
  // adding all left elements from stack to exp
  while (!isEmpty(stack))
    expression[++j] = pop(stack);

  expression[++j] = '\0';
}
void reverse(char *exp){

  int size = strlen(exp);
  int j = size, i=0;
  char temp[size];
  temp[j--]='\0';
  while(exp[i]!='\0')
  {
    temp[j] = exp[i];
    j--;
    i++;
  }
  strcpy(exp,temp);
}
void brackets(char* exp){
  int i = 0;
  while(exp[i]!='\0')
  {
    if(exp[i]=='(')
        exp[i]=')';
    else if(exp[i]==')')
        exp[i]='(';
    i++;
```

```
  }
}
void InfixtoPrefix(char *exp){
  int size = strlen(exp);
  // reverse string
  reverse(exp);
  //change brackets
  brackets(exp);
  //get postfix
  getPostfix(exp);
  // reverse string again
  reverse(exp);
}
int main()
{
  printf("The infix is: ");
  char expression[] = "((a/b)+c)-(d+(e*f))";
  printf("%s\n",expression);
  InfixtoPrefix(expression);
  printf("The prefix is: ");
  printf("%s\n",expression);
  return 0;
}
```

Output –

The Infix is: ((a/b)+c)-(d+(e*f))

The prefix is: -+/abc+d*ef

Introduction to stack

More about Stacks

1. Stack can be defined as a Data Structure that serves as saving the data in a particular fashion.
2. In linear data structures like an array and linked list a user is allowed to insert or delete any element to and from any location respectively.
3. However, in a Stack, both, insertion and deletion, is permitted at one end only.
4. A Stack works on the LIFO (Last In – First Out) basis, i.e, the first element that is inserted in the stack would be the last to be deleted; or the last element to be inserted in the stack would be the first to be deleted.

Stack Operations

Push: Adding a new item to the Stack Data Structure, in other words pushing new item to Stack DS.

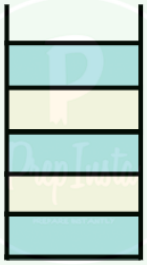
If Stack is full, then it is said to be in an overflow condition

Pop: Removing an item from the stack, i.e. popping an item out.

If a stack is empty then it is said to be in an underflow condition

1. Peek: This basically returns the topmost item in the stack, in other words, peek what is the topmost item.
2. IsEmpty: This returns True If the stack is empty else returns False

Data Structure: Stack



Representation of stack.

Stack as a data structure can be represented in two ways.

- Stack as an Array.
- Stack as a struct
- Stack as a Linked List.

Operations on a Stack

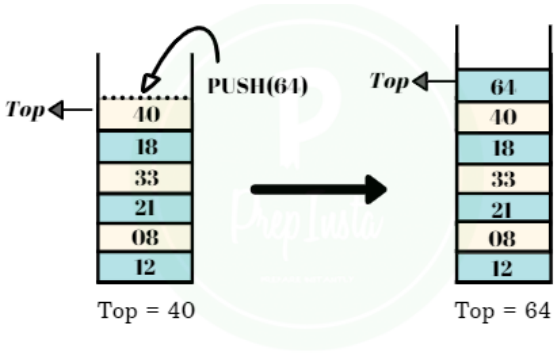
There are a number of operations we can perform on a stack as per our need which are as follows:

- Push().
- Pop().
- Top/Peak().
- isEmpty().

The time complexity of each of these operations is constant time, i.e , *O(1)*.

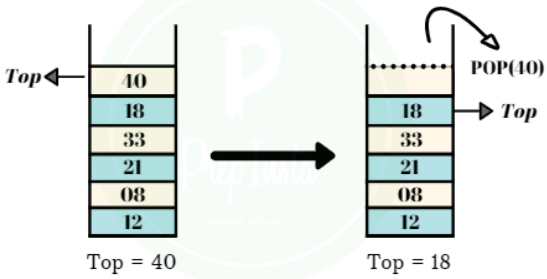
1. Push()

- When we require to add an element to the stack we perform *Push()* operation.
- Push() operation is synonymous of insertion in a data structure.



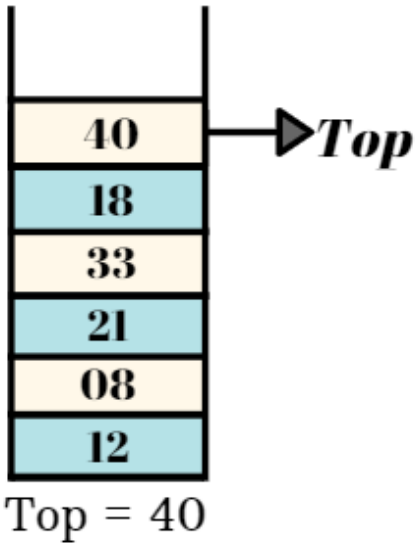
2. Pop()

- When we require to remove an element to the stack we perform *Pop()* operation.
- Pop() operation is synonymous of deletion in a data structure.



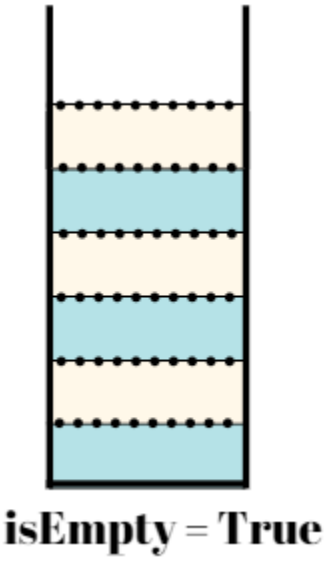
3. Top()

- This operation returns the top most or peak element of the stack.
- The value of top changes with each push() or pop() operation.



4. isEmpty()

- This operation returns *true* if the stack is found to be empty.
- Empty stack symbolizes that *top = -1*.



How does the stack work?

- Top: An integer variable Top is used, to keep track of topmost element in the stack

1. When we initialise the stack, there is no element in a stack so, Top value is initialised to Top = -1
2. The value is -1 as if using an array to implement a stack if there is 1 element the location would be a[0], thus, for no element we use – 1.
3. Push: on pushing new element the Top value increments by 1, top++
4. Pop: on popping a new element the top value decrements by 1, top–
5. Before pushing we check if the stack is full or not
6. Before popping we check if the stack is empty or not

Code for stack using structure in C:

```
// C Program for Implementation of stack (array) using
structure
#include  <limits.h>
#include  <stdio.h>
#include  <stdlib.h>

// A structure to represent a stack
struct Stack {
   int top;
   int maxSize;
   int* array;
};

struct Stack* create(int max)
{
   struct Stack* stack = (struct Stack*)malloc(sizeof(struct
Stack));
   stack->maxSize = max;
   stack->top = -1;
   stack->array = (int*)malloc(stack->maxSize *
sizeof(int));
   //here above memory for array is being created
   // size would be 10*4 = 40
   return stack;
}

// Checking with this function is stack is full or not
// Will return true is stack is full else false
//Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
   if(stack->top == stack->maxSize - 1){
```

```
        printf("Will not be able to push maxSize reached\n");
    }
    // Since array starts from 0, and maxSize starts from 1
    return stack->top == stack->maxSize - 1;
}

// By definition the Stack is empty when top is equal to -1
// Will return true if top is -1
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}

// Push function here, inserts value in stack and increments
stack top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("We have pushed %d to stack\n", item);
}

// Function to remove an item from stack.  It decreases top
by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

int main()
{
struct Stack* stack = create(10);

    push(stack, 5);
    push(stack, 10);
    push(stack, 15);

    int flag=1;
    while(flag)
    {
        if(!isEmpty(stack))
            printf("We have popped %d from stack\n",
pop(stack));
        else
            printf("Can't Pop stack must be empty\n");

        printf("Do you want to Pop again? Yes: 1 No: 0\n");
        scanf("%d",&flag);
    }
    return 0;
}
```

O/P

```
We have pushed 5 to stack
We have pushed 10 to stack
We have pushed 15 to stack
We have popped 15 from the stack
Do you want to Pop again? Yes: 1 No: 0
0
```

Code in C++:

```
#include <bits/stdc++.h>
using namespace std;


struct Node {
    int data;
    Node *next = NULL;
};
class Stack {
    Node *top = NULL;
    public:
        void push();  // Used to push element
        void pop();  // Used to pop element
        void diplay(); // Used to display the stack
        ~Stack();     // Used to remove the memory
};
void Stack::push() {
    Node *temp = new Node; // Creating New node
    cout << "Enter Data: ";
    int value;
    cin >> value;
    temp -> data = value; // Adding value
    temp -> next = top;   // Pointing to top
    top = temp;  // Changing top to current element
}
void Stack::pop() {
    if (top == NULL) {   // Check whether empty or not
        cout <<"Stack Empty\n";
    }
    else {
        Node *temp = top;
        cout << temp -> data << " deleted\n"; // Data Deleted
        top = top -> next; // Changing top to next element
        delete temp;  // Deleting the memory
```

```
    }
}
void Stack::diplay() {
    Node *temp = top;
    while (temp != NULL) {  // Printing until encounter
NULL position
        cout << temp -> data << " ";
        temp = temp -> next;
    }
    cout << endl;
}
Stack::~Stack() { // Destructor Called to delete data

    while (top != NULL) {
        Node *temp = top;
        cout << temp -> data << " deleted\n";
        top = top -> next;
        delete temp;
    }
}
int main() {
    Stack st;
    char ch;
    do {
        cout <<"Enter \n";
        cout << "1. P for push operation\n";
        cout << "2. R for pop operation\n";
        cout << "3. D for display operation\n";
        cout << "4. Q to quit\n";
        cin >> ch;
        switch(ch) {
            case 'P' : st.push(); break;
            case 'R' : st.pop(); break;
            case 'D' : st.diplay(); break;
        }
    }while (ch != 'Q');
    return 0;
}
```

Output:

```
Enter
1. P for push operation
2. R for pop operation
3. D for display operation
4. Q to quit
P
Enter Data: 20
Enter
1. P for push operation
2. R for pop operation
3. D for display operation
4. Q to quit
P
Enter Data: 21
Enter
1. P for push operation
2. R for pop operation
3. D for display operation
4. Q to quit
R
21 deleted
Enter
1. P for push operation
2. R for pop operation
3. D for display operation
4. Q to quit
P
Enter Data: 11
Enter
1. P for push operation
2. R for pop operation
3. D for display operation
4. Q to quit
D
11 20
Enter
1. P for push operation
2. R for pop operation
3. D for display operation
4. Q to quit
Q
11 deleted
20 deleted
```

Code in java:

```
/* Java program to implement stack
operations using array*/
class Stack {
    static  int MAX = 100;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack


    boolean isEmpty()
    {
        return (top < 0);
    }
    Stack()
    {
        top = –1;
    }


    boolean push(int x)
    {
        if (top >= (MAX – 1)) {
            System.out.println("Overflow condition reached");
```

```
            return false;
        }
        else {
            a[++top] = x;
            System.out.println(x + " pushed into stack");
            return true;
        }
    }


    int pop()
    {
        if (top < 0) {
            System.out.println("Underflow condition reached");
            return 0;
        }
        else {
            int x = a[top–];
            return x;
        }
    }


    int peek()
    {
        if (top < 0) {
            System.out.println("Underflow condition");
            return 0;
        }
        else {
            int x = a[top];
            return x;
        }
    }
}
class Prepinsta {
    public static void main(String args[])
    {
        Stack stk = new Stack();
        stk.push(20);
        stk.push(40);
        stk.push(60);
        System.out.println(stk.pop());
    }
}
```

Java code to implement stack using linked list :

```
// Java Code for Linked List Implementation

public class LinkedList {

stacknode node;

static class stacknode {
    int data;
    stacknode next;

    stacknode(int data) { this.data = data; }
}

public boolean isEmpty()
{
    if (node == null) {
        return true;
    }
    else
        return false;
}

public void push(int data)
{
    stacknode newNode = new stacknode(data);

    if (node == null) {
        node = newNode;
    }
    else {
        stacknode temp = node;
        node = newNode;
        newNode.next = temp;
    }
    System.out.println(data);
}

public int pop()
{
    int popped = Integer.MIN_VALUE;
    if (node == null) {
        System.out.println("Stack is Empty");
    }
    else {
        popped = node.data;
        node = node.next;
    }
    return popped;
}

public int peek()
{
    if (node == null) {
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
    }
    else {
        return node.data;
```

```
    }
  }

  public static void main(String[] args)
  {

    LinkedList sll = new LinkedList();

    slk.push(20);
    slk.push(40);
    slk.push(60);

    System.out.println(slk.pop());

    System.out.println("Top element is " + slk.peek());
  }
}
```

Output :

60

Time complexity

Push- O(1)
Pop - O(1)
Display - O(n)
Space complexity - O(n)

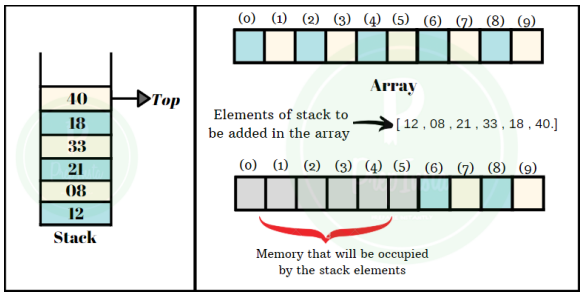
Representation of stack as array

A stack is a data structure that can be represented as an array. Let us learn more about Array representation of a stack –

An array is used to store an ordered list of elements. Using an array for representation of stack is one of the easy techniques to manage the data. But there is a major difference between an array and a stack.

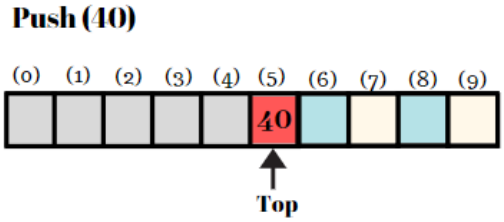
- Size of an array is fixed.
- While, in a stack, there is no fixed size since the size of stack changed with the number of elements inserted or deleted to and from it.

Despite the difference, an array can be used to represent a stack by taking an array of maximum size; big enough to manage a stack.



Step 1:

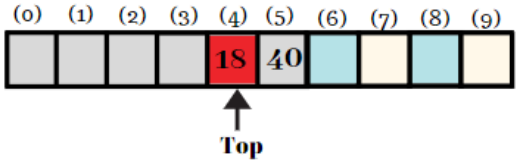
- Push (40).
- Top = 40
- Element is inserted at a[5].



Step 2:

- Push (18).
- Top = 18
- Element is inserted at a[4].

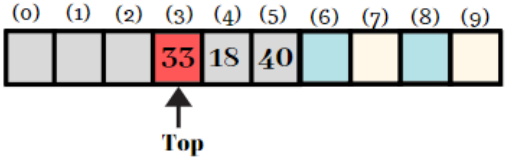
Push (18)



Step 3:

- Push (33).
- Top = 33
- Element is inserted at a[3].

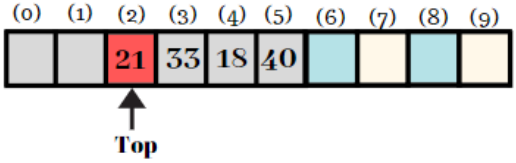
Push (33)



Step 4:

- Push (21).
- Top = 21
- Element is inserted at a[2].

Push (21)



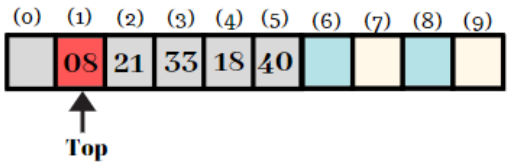
Step 5:

- Push (08).
- Top = 08
- Element is inserted at a[1].

Step 6:

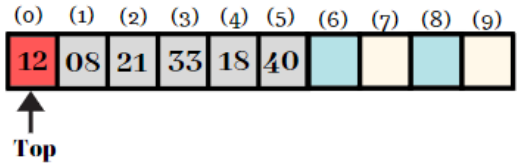
- Push (12).
- Top = 12

Push (08)



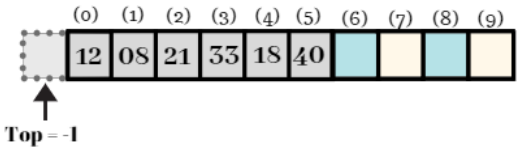
- Element is inserted at a[0].

Push (12)



Step 7:

- Top = -1.
- End of array.



Code in C(using array):

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

#define size 20

int stack[size],top_of_stack;

void Push(int [],int); //to insert elements in the stack

void Pop (int []);//to delete elements from the stack

void display(int []); //to display the elements

void main()
{
   int num=0;
   int option=0;
   top_of_stack=-1;

   while(1)
   {
      printf("\n 1: Push \n 2: Pop \n 3: Display \n 4: Exit \n
Enter one of your choices: ");
      scanf("%d",&option);

      switch(option)
      {
         case 1:
            printf("Enter the item you want to insert :");
            scanf("%d",&num);
            Push(stack,num);
         break;

         case 2:
            Pop(stack);
         break;

         case 3:
            display(stack);
         break;

         case 4:
            exit(0);

         default:
            printf("\nPlease enter a correct option\n");
         break;
      }
   }

}

void Push(int stack[],int num)
{
   if(top_of_stack==size-1)
   {
      printf("\nCannot add element, Stack is full\n");
      return;
   }
   top_of_stack++;
   stack[top_of_stack]=num;
}

void Pop(int stack[])
{
   int del_num;
   if(top_of_stack==-1)
   {
      printf("Empty Stack.\n");
      return;
   }

   del_num=stack[top_of_stack];
   top_of_stack--;
   printf("Element deleted: %d \n",del_num);
   return;
}

void display(int stack[])
{
   int i=0;
   if(top_of_stack==-1)
   {
      printf("Empty Stack .\n");
      return;
   }

   printf("Top of stack is: \n%d ",stack[top_of_stack]);
   for(i = top_of_stack ; i >=0 ; i--)
   {
      printf(" %d ",stack[i]);
   }
   printf("\n\n");
}
```

O/P

1: Push

2: Pop

```
3: Display
4: Exit

Enter one of your choices: 1
Enter the item you want to insert :1
1: Push
2: Pop
3: Display
4: Exit

Enter one of your choices: 1
Enter the item you want to insert :2
1: Push
2: Pop
3: Display
4: Exit
Enter one of your choices: 1

Enter the item you want to insert :3

1: Push
2: Pop
3: Display
4: Exit
Enter one of your choices: 2

Element deleted: 3
1: Push
2: Pop
3: Display
4: Exit
Enter one of your choices: 3

Top of stack is:
2  1

1: Push
2: Pop
3: Display
4: Exit
Enter one of your choices: 4
```

Code in java(using array):

```
/* Java program to implement stack
operations using array*/
class Stack {
    static  int MAX = 100;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack


    boolean isEmpty()
    {
        return (top < 0);
    }
    Stack()
    {
        top = –1;
    }


    boolean push(int x)
    {
        if (top >= (MAX – 1)) {
            System.out.println(“Overflow condition reached”);
            return false;
        }
        else {
            a[++top] = x;
            System.out.println(x + ” pushed into stack”);
            return true;
        }
    }


    int pop()
    {
        if (top < 0) {
            System.out.println(“Underflow condition reached”);
            return 0;
        }
        else {
            int x = a[top–];
            return x;
        }
    }


    int peek()
    {
        if (top < 0) {
            System.out.println(“Underflow condition”);
            return 0;
        }
        else {
            int x = a[top];
            return x;
        }
    }
}
class Prepinsta {
    public static void main(String args[])
    {
        Stack stk = new Stack();
        stk.push(20);
        stk.push(40);
```

```
        stk.push(60);
        System.out.println(stk.pop());
    }
}
```

Output :

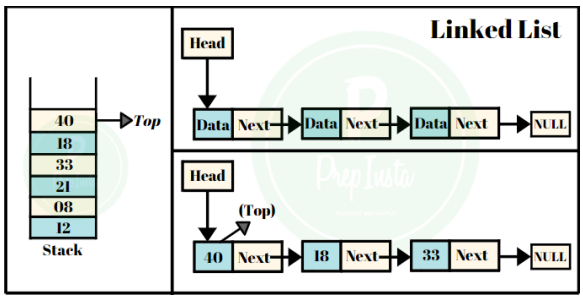
60

Representation of stack as linked list

First, we have to understand what a linked list is, to understand linked list representation of a stack. A Linked List is a Data Structure which consists of two parts:

- The data/value part.
- The pointer which gives the location of the next node in the list.

Stack can also be represented by using a linked list. We know that in the case of arrays we face a limitation , i.e , array is a data structure of limited size. Hence before using an array to represent a stack we will have to consider an enough amount of space to suffice the memory required by the stack.



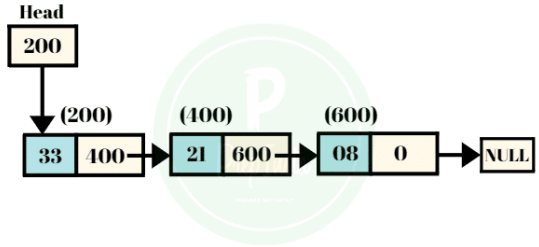
However, a Linked List is a much more flexible data structure. Both the push and pop operations can be performed on either ends.. But We prefer performing the Push and pop operation at the beginning. The Stack operations occur in constant time. But if we perform the push and pop operations at the end of the linked list it takes time *O(n)*.

But performing the operations at the beginning occurs in constant time. Let us understand this with the help of an example.

Let us consider a linked list as shown here.

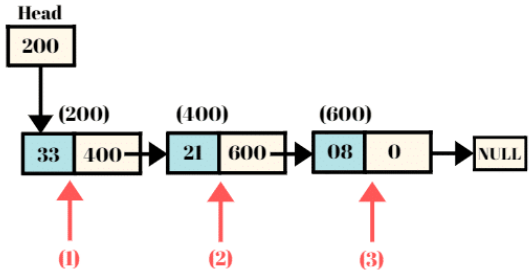
In the given data we can see that we have-

- **Head = 200.**
- **Top = 33.**

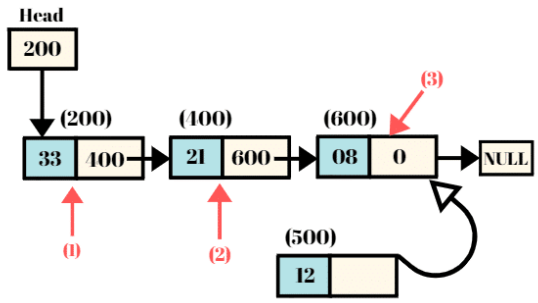


Push / Pop At the end Of a Linked List

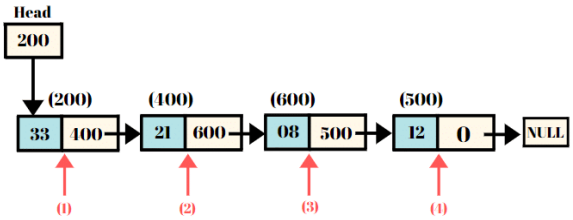
Consider the Linked List as shown above. Now if we want to push/pop a node at the end, we have to traverse all the *n* number of nodes, until we reach the end of the linked list. We traverse the whole list from the beginning to the end.



- Now if we want to insert a node at the end of the linked list, we traverse from the first node to the second node to the last node.
- We find the end of the list where the node points to the null and the node to be inserted is as shown here.



- The new node is pushed at the end of the list.
- The second last node , i.e, 08 now points to the location of the last node.
- The newly inserted node at the end now points to null.

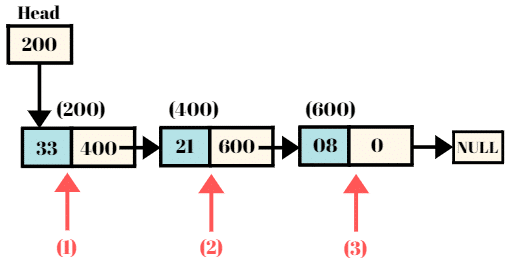


Push / Pop operation at the beginning of the Linked List

Pushing a new node at the beginning of a linked list takes place in constant time. When inserting the new node the number of traversals that occur is equal to 1. So it is much faster than the previous method of insertion

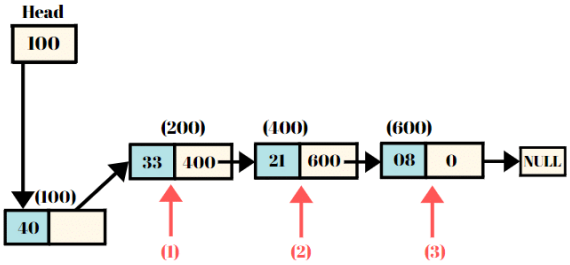
The time complexity of inserting a new node at the beginning of the linked list is *O(1)*.

Let us consider the previous example only. We are given a Linked List as follows:

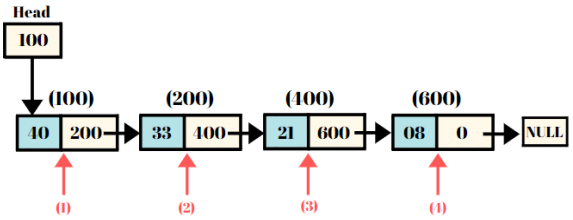


A new node to be inserted to the Linked List is shown here.

The new node *40* is to be inserted at the beginning of the list.



- The new node is inserted at the beginning where we have just a single traversal after spotting the head of the Linked List.
- The value of head changes from 200 to 100.
- The pointer of the new node stores the address of the next node.
- The final linked list is shown here.



Code in C(using linked list)

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

void Push();  /*to insert the element*/

void pop();  /*to delete the element*/

void display();  /*to display the elements in a stack*/

struct n
{
int element;
struct n *nxt;
};
struct n *hd;

void main ()
{
   int option=0;
   while(option != 4)
   {
      printf("\nSelect from the following
options\n1.Push\n2.Pop\n3.Show\n4.Exit \nEnter one of
your choices:");
      scanf("%d",&option);
      switch(option)
      {
         case 1:
         {
            Push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("Exit");
            break;
         }
         default:
         {
            printf("\nKindly enter a valid option\n");
         }
   };
}
}
void Push ()
{
   int element;
   struct n *ptr = (struct n*)malloc(sizeof(struct n));
   if(ptr == NULL)
   {
      printf("Stack full. Operation failed");
   }
   else
   {
      printf("Enter the element you want to insert:  ");
      scanf("%d",&element);
      if(hd==NULL)
      {
         ptr->element = element;
         ptr -> nxt = NULL;
         hd=ptr;
      }
      else
      {
         ptr->element = element;
         ptr->nxt = hd;
         hd=ptr;

      }
      printf("Element is inserted");

   }
}

void pop()
{
   int num;
   struct n *ptr;
   if (hd == NULL)
   {
      printf("Not enough elements. Underflow!");
```

```
    }
    else
    {
        num = hd->element;
        ptr = hd;
        hd = hd->nxt;
        free(ptr);
        printf("Element deleted: %d",num);

    }
}
void display()
{
    int i;
    struct n *ptr;
    ptr=hd;
    if(ptr == NULL)
    {
        printf("The stack is empty\n");
    }
    else
    {
        printf("The stack elements are as follows:  \n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->element);
            ptr = ptr->nxt;
        }
    }
}
```

Code in java(using linked list)

```
// Java Code for Linked List Implementation

public class LinkedList {

    stacknode node;

    static class stacknode {
        int data;
        stacknode next;

        stacknode(int data) { this.data = data; }
    }

    public boolean isEmpty()
    {
        if (node == null) {
            return true;
        }
        else
            return false;
}

public void push(int data)
{
    stacknode newNode = new stacknode(data);

    if (node == null) {
        node = newNode;
    }
    else {
        stacknode temp = node;
        node = newNode;
        newNode.next = temp;
    }
    System.out.println(data);
}

public int pop()
{
    int popped = Integer.MIN_VALUE;
    if (node == null) {
        System.out.println("Stack is Empty");
    }
    else {
        popped = node.data;
        node = node.next;
    }
    return popped;
}

public int peek()
{
    if (node == null) {
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
    }
    else {
        return node.data;
    }
}


public static void main(String[] args)
{

    LinkedList sll = new LinkedList();

    slk.push(20);
    slk.push(40);
    slk.push(60);

    System.out.println(slk.pop());

    System.out.println("Top element is " + slk.peek());
```

```
    }
}
```

Output :

Top element is 60

Infix To Postfix Conversion using Stack

Any operation can be expressed in Infix, postfix and prefix, we shall see how to convert infix to prefix operation via manual calculation and via code.

Overview

- Infix – Any operation of format a op b format example a + b is called an infix operation
- Postfix – An operation or expression can also be written in the format of a b op i.e. a b + which is similar to writing a + b in infix. All we are doing is shifting operator to the right of operands

Why do we need a postfix operator?

For a compiler, it is easier to read postfix or prefix operations. As a compiler either reads from right to left or left to right. Let us understand this with the help of an example –

1. Let us assume compiler starts reading from right to left
2. It reads c + d the operation first and it would be efficient if it could've implemented it, but, it can't as next operation is c * b which has higher precedence and must be implemented first.
3. Thus, if the compiler reads a notation in which, it can keep on implementing operations as soon as it sees them right!

The corresponding Postfix would be: abc*+d+

Steps to convert

Any infix op1 oper op2 can be written as op1 op2 oper

Where op1 = Operand 1

op2 = Operand2

oper = Operation

Example a + b can be written as ab+ in postfix

Problem

1. Infix: a + b * c + d can be written as a + (b * c) + d

2. Now, for all + – / * associativity is left to right we will write it as

3. (a + (b * c)) + d and thus further ((a + (b * c)) + d)

4. Solving and converting innermost bracket to postfix

Step 1 –((a + bc*)+ d)

Step 2 – Consider bc* as separate operand x the innermost bracket now looks like ((a + x)+ d)

Applying postfix it looks like – (ax+ + d)replacing x here (abc*+ + d)

Step 3 – Consideringabc*+as separate operand z, the final bracket looks like – (z + d)the result would be zd+ replacing z value = abc*+d+

Algorithm

1. First Start scanning the expression from left to right

2. If the scanned character is an operand, output it, i.e. print it

3. Else

*If the precedence of the scanned operator is higher than the precedence of the operator in the stack(or stack is empty or has'('), then push operator in the stack

*Else, Pop all the operators that have greater or equal precedence than the scanned operator. Once you pop them push this scanned operator. (If we see a parenthesis while popping then stop and push scanned operator in the stack)

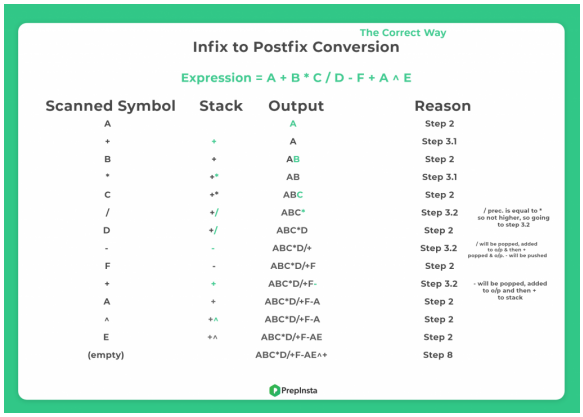
1. If the scanned character is an '(', push it to the stack.

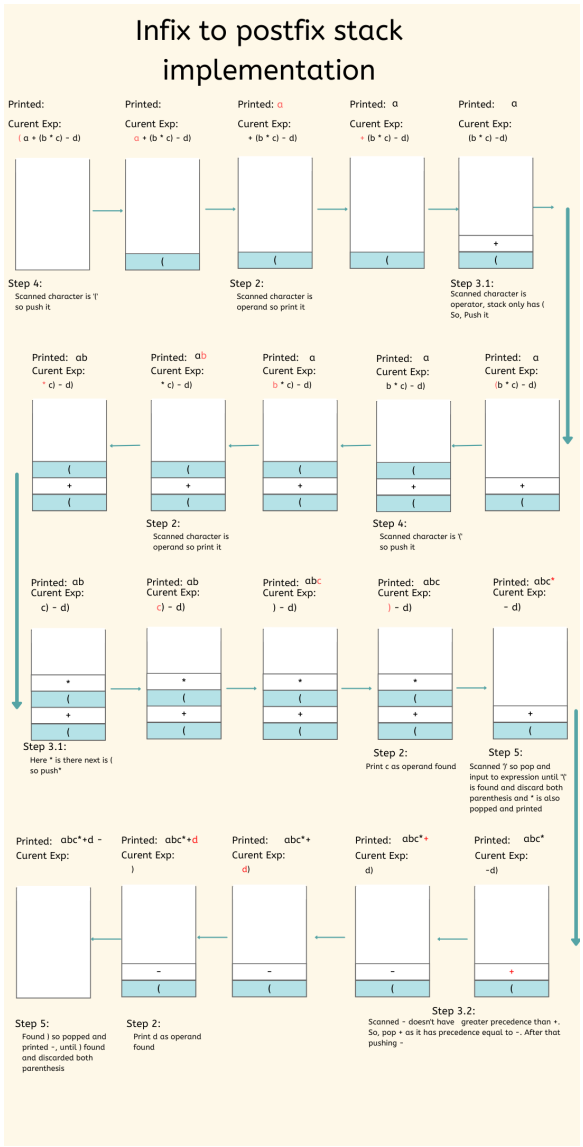
2. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.

3. Now, we should repeat the steps 2 – 6 until the whole infix i.e. whole characters are scanned.

4. Print output

5.Do the pop and output (print) until stack is not empty





Program for Infix to Postfix in C

// C Program for Implementation of stack (array) using structure

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// A structure to represent a stack
struct Stack {
  int top;
  int maxSize;
  // we are storing string in integer array, this will not give
error
```

```
  // as values will be stored in ASCII and returned in ASCII
thus, returned as string again
  int* array;
};

struct Stack* create(int max)
{
  struct Stack* stack = (struct Stack*)malloc(sizeof(struct
Stack));
  stack->maxSize = max;
  stack->top = -1;
  stack->array = (int*)malloc(stack->maxSize *
sizeof(int));
  return stack;
}

// Checking with this function is stack is full or not
// Will return true is stack is full else false
//Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
  if(stack->top == stack->maxSize - 1){
     printf("Will not be able to push maxSize reached\n");
  }
  // Since array starts from 0, and maxSize starts from 1
  return stack->top == stack->maxSize - 1;
}

// By definition the Stack is empty when top is equal to -1
// Will return true if top is -1
int isEmpty(struct Stack* stack)
{
  return stack->top == -1;
}

// Push function here, inserts value in stack and increments
stack top by 1
void push(struct Stack* stack, int item)
{
  if (isFull(stack))
     return;
  stack->array[++stack->top] = item;
}

// Function to remove an item from stack.  It decreases top
by 1
int pop(struct Stack* stack)
{
  if (isEmpty(stack))
     return INT_MIN;
  return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
  if (isEmpty(stack))
     return INT_MIN;
  return stack->array[stack->top];
}

// A utility function to check if the given character is
operand
int checkIfOperand(char ch)
{
  return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

// Function to compare precedence
// If we return larger value means higher precedence
int precedence(char ch)
{
  switch (ch)
  {
  case '+':
  case '-':
     return 1;

  case '*':
  case '/':
     return 2;

  case '^':
     return 3;
  }
  return -1;
}

// The driver function for infix to postfix conversion
int covertInfixToPostfix(char* expression)
{
  int i, j;

  // Stack size should be equal to expression size for safety
  struct Stack* stack = create(strlen(expression));
  if(!stack) // just checking is stack was created or not
     return -1 ;

  for (i = 0, j = -1; expression[i]; ++i)
  {
     // Here we are checking is the character we scanned is
operand or not
     // and this adding to to output.
     if (checkIfOperand(expression[i]))
        expression[++j] = expression[i];

     // Here, if we scan character '(', we need push it to the
stack.
```

```
    else if (expression[i] == '(')
        push(stack, expression[i]);

    // Here, if we scan character is an ‘)’, we need to pop
and print from the stack
    // do this until an ‘(‘ is encountered in the stack.
    else if (expression[i] == ')')
    {
        while (!isEmpty(stack) && peek(stack) != '('
            expression[++j] = pop(stack);
        if (!isEmpty(stack) && peek(stack) != '('
            return -1; // invalid expression
        else
            pop(stack);
    }
    else // if an operator
    {
        while (!isEmpty(stack) &&
precedence(expression[i]) <= precedence(peek(stack)))
            expression[++j] = pop(stack);
        push(stack, expression[i]);
    }

  }

  // Once all initial expression characters are traversed
  // adding all left elements from stack to exp
  while (!isEmpty(stack))
    expression[++j] = pop(stack);

  expression[++j] = '\0';
  printf( "%s", expression);
}

int main()
{
char expression[] = "((a+(b*c))-d)";
  covertInfixToPostfix(expression);
  return 0;
}
```

Output

```
abc*+d-
```

Infix To Postfix Conversion using Stack in C++

```
// C++ Program infix to postfix expression using stack
(array) in data structure
#include<iostream>
#include<string>
#define MAX 20
using namespace std;

char stk[20];
int top=-1;
// Push function here, inserts value in stack and increments
stack top by 1
void push(char oper)
{
  if(top==MAX-1)
  {
    cout<<"stackfull!!!!";
  }

  else
  {
    top++;
    stk[top]=oper;
  }
}
// Function to remove an item from stack.  It decreases top
by 1
char pop()
{
  char ch;
  if(top==-1)
  {
    cout<<"stackempty!!!!";
  }
  else
  {
    ch=stk[top];
    stk[top]='\0';
    top--;
    return(ch);
  }
  return 0;
}
int priority ( char alpha )
{
  if(alpha == '+' || alpha =='-')
  {
    return(1);
  }

  if(alpha == '*' || alpha =='/')
  {
    return(2);
  }

  if(alpha == '$')
  {
    return(3);
```

```
    }

    return 0;
}
string convert(string infix)
{
    int i=0;
    string postfix = "";
    while(infix[i]!='\0')
    {
        if(infix[i]>='a' && infix[i]<='z'|| infix[i]>='A'&&
infix[i]<='Z')
        {
            postfix.insert(postfix.end(),infix[i]);
            i++;
        }
        else if(infix[i]=='(' || infix[i]=='{'  || infix[i]=='[')
        {
            push(infix[i]);
            i++;
        }
        else if(infix[i]==')' || infix[i]=='}'  || infix[i]==']')
        {
            if(infix[i]==')')
            {
                while(stk[top]!='(')
                {               postfix.insert(postfix.end(),pop());
                }
                pop();
                i++;
            }
            if(infix[i]==']')
            {
                while(stk[top]!='[')
                {
                    postfix.insert(postfix.end(),pop());
                }
                pop();
                i++;
            }

            if(infix[i]=='}')
            {
                while(stk[top]!='{')
                {
                    postfix.insert(postfix.end(),pop());
                }
                pop();
                i++;
            }
        }
        else
        {
            if(top==-1)
            {
                push(infix[i]);
                i++;
            }

            else if( priority(infix[i]) <= priority(stk[top])) {
                postfix.insert(postfix.end(),pop());

                while(priority(stk[top]) == priority(infix[i])){
                    postfix.insert(postfix.end(),pop());
                    if(top < 0) {
                        break;
                    }
                }
                push(infix[i]);
                i++;
            }
            else if(priority(infix[i]) > priority(stk[top])) {
                push(infix[i]);
                i++;
            }
        }
    }
    while(top!=-1)
    {
        postfix.insert(postfix.end(),pop());
    }
    cout<<"The converted postfix string is : "<<postfix; //it
will print postfix conversion
    return postfix;
}

int main()
{
    int cont;
    string infix, postfix;
    cout<<"\nEnter the infix expression : "; //enter the
expression
    cin>>infix;
    postfix = convert(infix);
    return 0;
}
```

Output:-

```
Enter the infix expression : a=b*2+5
The converted postfix string is : ab*=+25
```

Infix to postfix in java

```
import java.util.Stack;
```

```
public class InfixToPostfix {


    public static boolean isOperator(char op){

        if(op == '+' || op == '-' || op == '*' || op == '/' || op
=='%')

            return true;

        return false;

    }


    /*  precedence to an operator */

    public static int precedence(char op){

        switch (op){

            case '+' :

            case '-' :

                return 1;

            case '/' :

            case '*' :

                return 2;

            case '%' :

                return 3;

            default :

                return 4;

        }

    }


    /* This function tell if the op1 has lower precedence than
op2 */

    public static boolean isLowerPrecedence(char op1, char
op2){

        if(precedence (op1) < precedence(op2))

            return true;

        return false;

    }

    public static String convertInixToPostfix(String infix){

        Stack<Character> stack = new Stack();

        StringBuilder sb = new StringBuilder();


        for(int i=0; i<infix.length(); i++){

            char currentCharacter = infix.charAt(i);

            //If it's an operand, put it on output string

            if(!isOperator(currentCharacter)){

                sb.append(currentCharacter);

            }

            else{

                if(stack.empty()) stack.push(currentCharacter);

                else{

                    while(!stack.empty()

                    && isLowerPrecedence(currentCharacter,
stack.peek())){

                        sb.append(stack.pop());

                    }

                    stack.push(currentCharacter);

                }

            }

        }

        while(!stack.empty()) sb.append(stack.pop());


        return sb.toString();

    }


    public static void main(String[] args){
```

```
        System.out.println(convertInixToPostfix("4*5+6/7"));
    }
}
```

OUTPUT:

input:

3+6*4/8

output:

36*48/+

ADVANTAGE OF POSTFIX:

Any formula can be expressed without parenthesis.

It is very convenient for evaluating formulas on a computer with stacks.

Postfix expression doesn't have the operator precedence.

Postfix is slightly easier to evaluate.

It reflects the order in which operations are performed.

You need to worry about the left and right associativity.

Infix to prefix expression using stacks:

As we know for the compiler it may be difficult to read an infix expression as it can't start operations and assignments until it has read the whole expression and know about precedence of various operations in the expression.

While prefix expression is easier for the compiler, as it is already sorted in accordance with precedence and compiler, can start assignments and operations, without caring about precedence.
What are infix and prefix expressions?

1. Infix: Expressions of format (A + B) are called as infix expressions, these are just like mathematical expressions

Example – ((a / b) + c) - (d + (e * f))

2. Prefix: Expressions wherein the operator comes before the operands are prefix expression like – Infix: (A + B) can be expressed as +AB

Example – Prefix result would be : -+/abc+d*ef

3. Postfix: Expression operator comes after the operands are prefix expression like – Infix: (A + B) can be expressed as AB+

Example – Prefix result would be: ab/c+def*+-

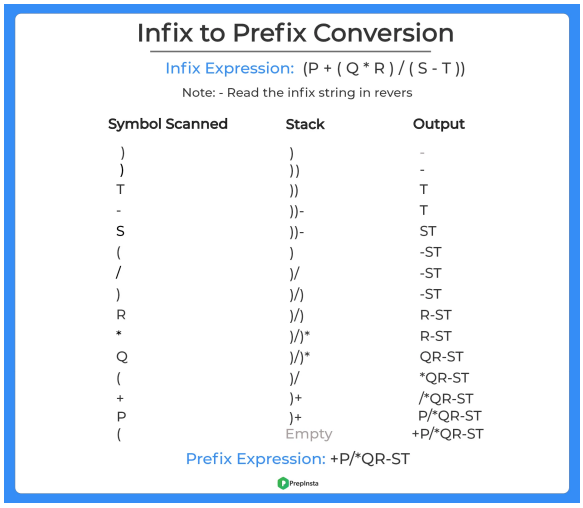
Steps to convert

- Any infix op1 oper op2 can be written as op1 op2 oper
    - Where op1 = Operand 1
    - op2 = Operand2
    - oper = Operation
- Example a + b can be written as ab+ in prefix

Problem (This is how to convert manually for MCQ Question in the exam)

- Infix: (a / b + c) - (d + e * f) can be written as ((a / b) + c) - (d + (e * f))
- Now, we have done the above according to associativity
- Solving and converting innermost bracket to prefix
- Step 1 –(/ab + c) - (d + *ef)
- Step 2 – Consider /ab and *ef as separate operand x and y

- the innermost bracket now looks like (x + c) - (d + y)
    - Applying prefix it looks like – (+xc - +dy)replacing x and y here (+/abc - +d*ef)
- Step 3 – Considering/abc and+d*efas separate operand z and w, the final bracket looks like – (z - w)the result would be -zw
    - replacing z and w value = -+/abc+d*ef



C Code for Infix to Prefix Conversion

```
#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>



// A structure to represent a stack

struct Stack {

  int top;

  int maxSize;

  // we are storing string in integer array, this will not give error

  // as values will be stored in ASCII and returned in ASCII thus, returned as string again

  int* array;

};

struct Stack* create(int max)

{

  struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));

  stack->maxSize = max;

  stack->top = -1;

  stack->array = (int*)malloc(stack->maxSize * sizeof(int));

  return stack;

}

// Checking with this function is stack is full or not

// Will return true is stack is full else false

//Stack is full when top is equal to the last index

int isFull(struct Stack* stack)

{

  if(stack->top == stack->maxSize - 1){

     printf("Will not be able to push maxSize reached\n");

  }

  // Since array starts from 0, and maxSize starts from 1

  return stack->top == stack->maxSize - 1;

}

// By definition the Stack is empty when top is equal to -1
```

```
// Will return true if top is -1
int isEmpty(struct Stack* stack)
{
  return stack->top == -1;
}

// Push function here, inserts value in stack and increments stack top by 1
void push(struct Stack* stack, int item)
{
  if (isFull(stack))
    return;
  stack->array[++stack->top] = item;
}

// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack)
{
  if (isEmpty(stack))
    return INT_MIN;
  return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
  if (isEmpty(stack))
    return INT_MIN;
  return stack->array[stack->top];
}

// A utility function to check if the given character is operand
int checkIfOperand(char ch)
{
  return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

// Fucntion to compare precedence
// If we return larger value means higher precedence
int precedence(char ch)
{
  switch (ch)
  {
  case '+':
  case '-':
    return 1;

  case '*':
  case '/':
    return 2;

  case '^':
    return 3;
  }
```

```
  return -1;
}

// The driver function for infix to postfix conversion
int getPostfix(char* expression)
{
  int i, j;
  // Stack size should be equal to expression size for safety
  struct Stack* stack = create(strlen(expression));
  if(!stack) // just checking is stack was created or not
     return -1 ;
  for (i = 0, j = -1; expression[i]; ++i)
  {
     // Here we are checking is the character we scanned is operand or not
     // and this adding to to output.
     if (checkIfOperand(expression[i]))
        expression[++j] = expression[i];
     // Here, if we scan character '(', we need push it to the stack.
     else if (expression[i] == '(')
        push(stack, expression[i]);

     // Here, if we scan character is an ')', we need to pop and print from the stack
     // do this until an '(' is encountered in the stack.
     else if (expression[i] == ')')
     {
        while (!isEmpty(stack) && peek(stack) != '(')
           expression[++j] = pop(stack);
        if (!isEmpty(stack) && peek(stack) != '(')
           return -1; // invalid expression
        else
           pop(stack);
     }
     else // if an opertor
     {
        while (!isEmpty(stack) && precedence(expression[i]) <= precedence(peek(stack)))
           expression[++j] = pop(stack);
        push(stack, expression[i]);
     }
  }
  // Once all inital expression characters are traversed
  // adding all left elements from stack to exp
  while (!isEmpty(stack))
     expression[++j] = pop(stack);

  expression[++j] = '\0';

}

void reverse(char *exp){
```

```
  int size = strlen(exp);
  int j = size, i=0;
  char temp[size];

  temp[j--]='\0';
  while(exp[i]!='\0')
  {
     temp[j] = exp[i];
     j--;
     i++;
  }
  strcpy(exp,temp);
}
void brackets(char* exp){
  int i = 0;
  while(exp[i]!='\0')
  {
     if(exp[i]=='(')
        exp[i]=')';
     else if(exp[i]==')')
        exp[i]='(';
     i++;
  }
}
void InfixtoPrefix(char *exp){
  int size = strlen(exp);

  // reverse string
  reverse(exp);
  //change brackets
  brackets(exp);
  //get postfix
  getPostfix(exp);
  // reverse string again
  reverse(exp);
}

int main()
{
  printf("The infix is: ");

  char expression[] = "((a/b)+c)-(d+(e*f))";
  printf("%s\n",expression);
  InfixtoPrefix(expression);

  printf("The prefix is: ");
  printf("%s\n",expression);

  return 0;
}
```

Output –

The Infix is: ((a/b)+c)-(d+(e*f))

The prefix is: -+/abc+d*ef

C++ Program for Infix To Prefix Conversion using stack data structures:-

```
#include <iostream>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
using namespace std;
//definition of functions
struct Stack *create (int max);
int stackFull (struct Stack *stack);
int stackEmpty (struct Stack *stack);
void pushElement (struct Stack *stack, int item);
int popElement (struct Stack *stack);
int peekElement (struct Stack *stack);
int checkOperand (char ch);
int precedence (char ch);
int postfix (char *expression);
void reverse (char *exp);
void brackets (char *exp);
void conversionInfixToPrefix (char *exp);
// A structure to represent a stack
struct Stack
{
  int top;
  int maxSize;
  int *array;
};
int main ()
{
  int n = 10;
  cout << "The infix expression is: \n";
  char expression[] = "(P+(Q*R)/(S-T))";
  cout << expression << "\n";
  conversionInfixToPrefix (expression);
  cout << "The prefix expression is: \n";
  cout << expression;
  return 0;
}
//stack implementation
struct Stack * create (int max)
{
  struct Stack *stack = (struct Stack *) malloc (sizeof (struct
Stack));
  stack->maxSize = max;
  stack->top = -1;
  stack->array = (int *) malloc (stack->maxSize * sizeof
(int));
  return stack;
}

// Checking with this function is stack is full or not
int stackFull (struct Stack *stack)
{
  if (stack->top == stack->maxSize - 1)
    {
      cout << "Will not be able to push maxSize reached\n";
    }
  // We know array index from 0 and maxSize starts from 1
  return stack->top == stack->maxSize - 1;
}

// if Stack is empty when top is equal to -1 and return true
int stackEmpty (struct Stack *stack)
{
  return stack->top == -1;
}

// Push function it inserts value in stack and increments
stack top by 1
void pushElement (struct Stack *stack, int item)
{
  if (stackFull (stack))
    return;
  stack->array[++stack->top] = item;
}

// pop Function it remove an item from stack and decreases
top by 1
int popElement (struct Stack *stack)
{
  if (stackEmpty (stack))
    return INT_MIN;
  return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peekElement (struct Stack *stack)
{
  if (stackEmpty (stack))
    return INT_MIN;
  return stack->array[stack->top];
}

// A function check the given character is operand
int checkOperand (char ch)
{
  return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}
```

```
// Fucntion to compare precedence if return larger value
means higher precedence
int precedence (char ch)
{
 switch (ch)
   {
   case '+':
   case '-':
    return 1;

   case '*':
   case '/':
    return 2;

   case '^':
    return 3;
   }
 return -1;
}

// The function for infix to postfix conversion
int postfix (char *expression)
{
 int i, j;
 struct Stack *stack = create (strlen (expression));
 if (!stack)
   return -1;

 for (i = 0, j = -1; expression[i]; ++i)
   {
    // checking the character we scanned is operand or not
    if (checkOperand (expression[i]))
          expression[++j] = expression[i];

    // if we scan character push it to the stack
    else if (expression[i] == '(')
          pushElement (stack, expression[i]);

    //if we scan character we need to pop and print from the
stack
    else if (expression[i] == ')')
          {
           while (!stackEmpty (stack) && peekElement
(stack) != '(')
             expression[++j] = popElement (stack);
           if (!stackEmpty (stack) && peekElement (stack)
!= '(')
             return -1;                    // invalid
expression
           else
             popElement (stack);
          }
    else                          // if an operator
          {
           while (!stackEmpty (stack)
                  && precedence (expression[i]) <=
                  precedence (peekElement (stack)))
             expression[++j] = popElement (stack);
           pushElement (stack, expression[i]);
          }

   }

 // if all first expression characters are scanned
// adding all left elements from stack to expression
 while (!stackEmpty (stack))
   expression[++j] = popElement (stack);
   expression[++j] = '\0';
}

void reverse (char *exp)
{                                          //reverse function
for expression

 int size = strlen (exp);
 int j = size, i = 0;
 char temp[size];

 temp[j--] = '\0';
 while (exp[i] != '\0')
   {
    temp[j] = exp[i];
    j--;
    i++;
   }
 strcpy (exp, temp);
}

void brackets (char *exp)
{
 int i = 0;
 while (exp[i] != '\0')
   {
    if (exp[i] == '(')
          exp[i] = ')';
    else if (exp[i] == ')')
          exp[i] = '(';
    i++;
   }
}

void conversionInfixToPrefix (char *exp)
{
 int size = strlen (exp);
 reverse (exp);
 brackets (exp);
 postfix (exp);
```

```
  reverse (exp);
}
```
Output:-
The infix expression is:
(P+(Q*R)/(S-T))
The prefix expression is:
+P/*QR-ST

Java code :

```
import java.util.Stack;
public class InfixToPreFix {
   static int precedence(char c){
      switch (c){
         case '+':
         case '-':
            return 1;
         case '*':
         case '/':
            return 2;
         case '^':
            return 3;
      }
      return –1;
   }

   static StringBuilder infixToPreFix(String expression){

      StringBuilder result = new StringBuilder();
      StringBuilder input = new StringBuilder(expression);
      input.reverse();
      Stack<Character> stack = new Stack<Character>();

      char [] charsExp = new String(input).toCharArray();
      for (int i = 0; i < charsExp.length; i++) {

         if (charsExp[i] == '(') {
            charsExp[i] = ')';
            i++;
         }
         else if (charsExp[i] == ')') {
            charsExp[i] = '(';
            i++;
         }
      }
      for (int i = 0; i <charsExp.length ; i++) {
         char c = charsExp[i];

         if(precedence(c)>0){
            while(stack.isEmpty()==false &&
precedence(stack.peek())>=precedence(c)){
               result.append(stack.pop());
            }
            stack.push(c);
         }else if(c==')'){
            char x = stack.pop();
            while(x!='('){
               result.append(x);
               x = stack.pop();
            }
         }else if(c=='('){
            stack.push(c);
         }else{
            //character is neither operator nor "("
            result.append(c);
         }
      }

      for (int i = 0; i <=stack.size() ; i++) {
         result.append(stack.pop());
      }
      return result.reverse();
   }

   public static void main(String[] args) {
      String exp = "A+B*(C^D-E)";
      System.out.println("Infix Expression: " + exp);
      System.out.println("Prefix Expression: " +
infixToPreFix(exp));
   }
}
```
Output:

Infix Expression: A+B*(C^D-E)
Prefix Expression: +A*B-^CDE

Postfix to prefix conversion using stack

As we already know that any given operation in a compiler can be expressed in 3 ways which are Infix, postfix and prefix, we shall see how to convert postfix to prefix operation via manual calculation and via code.

What postfix, infix, prefix rules are –

- Infix: (X + Y)
- Postfix – The postfix will look like, XY+
- Prefix: The prefix will look like, +YX

Infix : (X + Y) / (U – V)

- Postfix – The postfix will look like, XY+UV-/
- Prefix – The prefix will look like, /+XY-UV

If we want to get postfix to infix all of us generally convert postfix to the infix and then convert Infix to prefix, as this is easier to do, and we even recommend doing the same.

*What when Postfix is given and you want to calculate Prefix directly?

*Or if you want to code the same, doing temporary conversion to Infix will be time-consuming and longer code will be there.

Following is the algorithm-

A + B, where A and B are operands and + is operator, the whole A + B is expression.

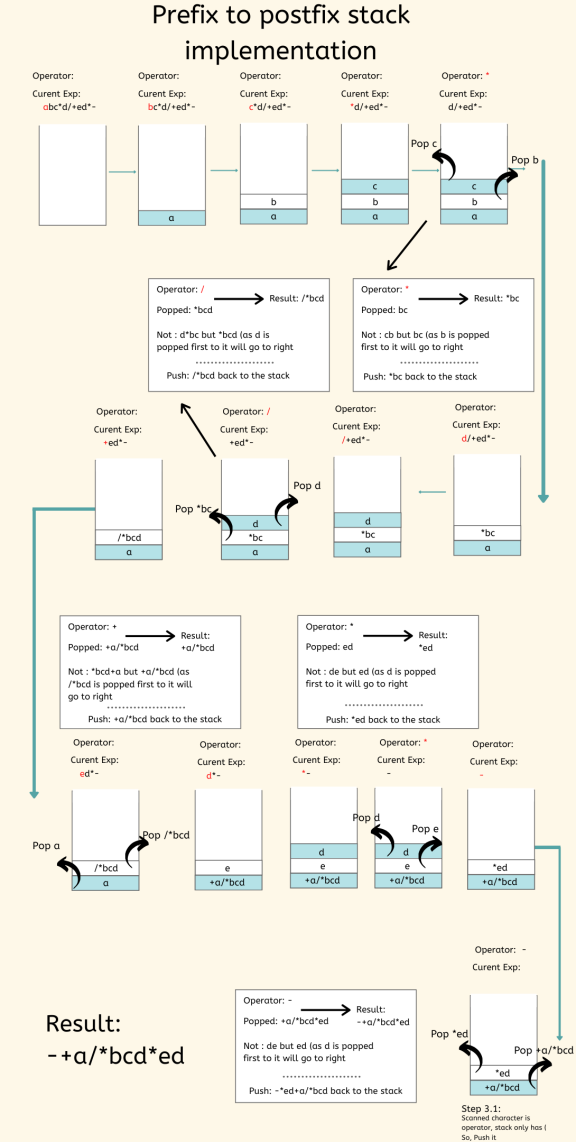
1.Start reading the expression from L – R i.e. Left to right

If you encounter an operand, then do push in the stack

1. If we encounter an operator then, pop two operands from the stack, and concatenate them, in +AB type of format
2. And push the resultant sting back to the stack
3. Repeat the above steps till end of the expression

- Postfix: abc*d/+ed*-
- Prefix: -+a/*bcd*ed
- Infix: ((a+((b*c)/d))-(e*d))

Below is an implementation for it



Code for Postfix to Prefix in C

However, to code, we will apply different logic as it is difficult to push into the stack when we have a 2D array in C. For example, pushing a character ‘a’ is fine but pushing *bc which is a string and in a stack which already is an array needs a 2D array, so we use other implementations in C. Please follow the code below and you will understand how this implementation works.

Note – It has a different logic than the above rule explained.

C code to conversion of postfix to prefix

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

#define MAX 20

char str[MAX], stack[MAX];
int top = -1;

void push(char c)
{
 stack[++top] = c;
}

char pop()
{
 return stack[top--];
}

// A utility function to check if the given character is operand
int checkIfOperand(char ch)
{
 return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

//function to check if it is an operator
int isOperator(char x)
{
 switch (x) {
 case '+':
 case '-':
 case '/':
 case '*':
   return 1;
 }
 return 0;
}

void postfixToprfix()
{
 int n, i, j = 0;
 char c[20];
 char a, b, op;

 printf("Enter the postfix expression\n");
 scanf("%s", str);
```

```
n = strlen(str);

for (i = 0; i < MAX; i++)
  stack[i] = '\0';
printf("Prefix expression is:\t");

for (i = n - 1; i >= 0; i--)
{
  if (isOperator(str[i]))
  {
    push(str[i]);
  } else
  {
    c[j++] = str[i];
    while ((top != -1) && (stack[top] == '#'))
    {
      a = pop();
      c[j++] = pop();
    }
    push('#');
  }
}
c[j] = '\0';

i = 0;
j = strlen(c) - 1;
char d[20];

while (c[i] != '\0') {
  d[j--] = c[i++];
}

printf("%s\n", d);

}
int main()
{
 postfixToprfix();

 return 0;
}

import java.util.Stack;
public class PostFixToPreFix {

   static boolean isOperator(char x){
       switch (x){
           case '-':
           case '+':
           case '/':
           case '*':
           case '^':
```

```
            return true;
        }
        return false;
    }
```

Java code for conversion of postfix to prefix

```
    public static String convert(String exp){

        Stack<String> st = new Stack<>();
        for (int i = 0; i  exp.length() ; i++) {

            char c = exp.charAt(i);

            if(isOperator(c)){
                String s1 = st.pop();
                String s2 = st.pop();
                String temp = c + s2 + s1;
                st.push(temp);
            }else{
                st.push(c+"");
            }
        }
        String result = st.pop();
        return result;
    }

    public static void main(String[] args) {
        String postfix = "AB+CD-*";
        System.out.println("Postfix exp: " + postfix);
        System.out.println("Prefix exp: " + convert(postfix));
    }
}
```

output:

postfix exp:

AB+CD-*

prefix exp:

*+AB-CD

Chapter 13. Linked List

What is a Linked List in Data Structure?

A linked list is a linear data structure. In which we can store the data in a sequence manner. Unlike an array linked list is dynamic data structure the size of a linked list can grow or shrink depending on the situation.

The element of a linked list is called a node. Every element or node contains two main entities. The first entity is the information or the data whereas the second entity is the pointer to the next node.

More Information

The structure of the linked list is like a train. A linked list has the following components –

Data: Data is the value stored in it, it could be anything, an integer or a string or anything else really.
Pointer (Link) – Each linked list contains a pointer which points to the address of the next node in the linked list train.
Apart from this the following are also keywords used in linked lists –

Node – Any single unit of a linked list, its data and pointer(s), is called a node.
Head: First is the head who is defined at the beginning of the list.
Tail: second is tail which defines the end of the list.

What is a linked list

Linked List like an array is a linear data structure. However, the elements or data storage of linked list elements are not in a contiguous manner and we use pointers to create linearity in linked lists.

Why do we use Linked List in Data Structure?

Generally, storage arrays are used. But, there are many many disadvantaged with arrays –

Fixed Size
The size of the array is fixed and needs to be pre-defined
Example – int a[10]; or int a[ ] = {1, 2, 3, 4, 5}
We can't declare an array without declaring the size.
Memory wastage (Not Optimised)
If you declare an array of size 10 and just assign values to first two elements
The memory is allocated to all of them even though we may not end up using them.
Need for Contiguous Memory
Arrays need Contiguous memory, which sometimes is bad.
Example – Consider there are only three memories of size 10, 10 and 400
We need to store an array of size 15
In this case, the array will use memory location size 400 and not combine memory A and B of size 10 and 10 respectively.
Thus wasting a lot of memory

Inserting in Array is expensive

Let us say you have an array a[] = {10, 20, 40, 60, 80} and we need to add 50 in the same sorted format.
Then, we have to move all elements after 40 to one additional position
Also, same goes for deletion if we want to delete 20. Then all elements after it have to be moved to one position in the opposite direction.

Linked Lists Advantages

Ease of insertion and deletion. (How? we will learn this in Linked List insertion Post)
Dynamic Size

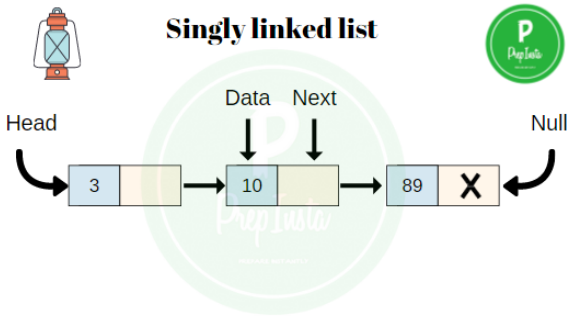
Disadvantages

We have to search elements linearly, and can't do random access
We can't do efficient searches like binary search
Extra space for pointers is needed
Linked lists are not cache friendly as arrays are stored in contiguous format they can be cached easily
Loss of data threat is there, if we lose one pointer location the rest of the linked list can't be accessed.

Types of Linked link

There are three types of linked list, that are mentioned in following ways-

Singly linked list
Doubly linked list
Circular linked list



Singly-linked list

In a singly list there are the two fields on the list.

Data: First field is data that it stores
Link (pointer): Link that refers to the address of the next linked list node.
The last node of linked list points to NULL and means end of the list
In a Singly linked list node, it only contains the address of the next node and not the previous node, so we can only traverse in one direction only.

C Code:

```
struct Node {
int data;
struct Node* next;
};
```

C++ Code:

```
class Node {
public:
int data;
Node* next;
};
```

Java COde:

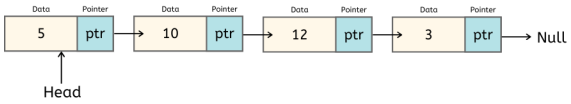
```
class LinkedList {
Node head; // head

/* Linked list Node*/
class Node {
int data;
Node next;

// For creating new node a Constructor
// Next is by default initialized
// as null
Node(int d) { data = d; }
}
}
```

Python Code:

```
# Node class
class Node:

# Function to initialize the node object
def __init__(self, data):
self.data = data # Assign data
self.next = None # Initialize
# next as null

# Linked List class
class LinkedList:

# Function to initialize the Linked
# List object
def __init__(self):
self.head = None
```

Program to Create a Singly List

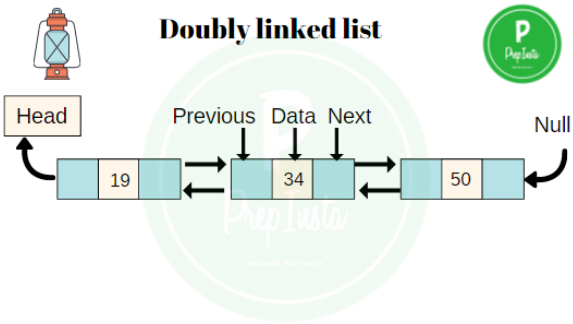
Here we will construct a linked list with 4 elements as follows –



C Code:

```
//We are creating program for linked list creation
#include <stdio.h>
#include <stdlib.h>
//stdlib used to provide a declaration of 'malloc'

// structure of linked list
struct Node {
  int data;
  struct Node* next;
  // Pointer pointing towards next node
};

//function to print the linked list
void display(struct Node* node)
{
  while (node != NULL) {
     printf(" %d ", node->data);
     node = node->next;
  }
}

// main function
int main()
```

```
{
  //creating 4 pointers of type struct Node
  //So these can point to address of struct type variable
  struct Node* head = NULL;
  struct Node* node2 = NULL;
  struct Node* node3 = NULL;
  struct Node* node4 = NULL;

  // allocate 3 nodes in the heap
  head =  (struct Node*)malloc(sizeof(struct Node));
  node2 = (struct Node*)malloc(sizeof(struct Node));
  node3 = (struct Node*)malloc(sizeof(struct Node));
  node4 = (struct Node*)malloc(sizeof(struct Node));

  /* Using malloc method we have created 4 memory
blocks
  Each memory block is of type struct and has int data and
Pointer of type Node
  So it can point towards a node type struct.

    head        node2        node3          node4
     |           |           |              |
     |           |           |              |
  +---+-----+    +----+----+    +----+----+    +----+----+
  |data|next|    |data|next|    |data|next|    |data|next|
  +---+-----+    +----+----+    +----+----+    +----+----+

 # As of now data has garbage value and its pointer
 points towards random addresses*/

  head->data = 5; // data set for head node
  head->next = node2; // next pointer assigned to address of
node2

  node2->data = 10;
  node2->next = node3;

  node3->data = 12;
  node3->next = node4;

  node4->data = 3;
  node4->next = NULL;
  //last node assigned to Null as linked list ends here

  /* Finally linked list looks like

    head        node2        node3          node4
     |           |           |              |
     |           |           |              |
  +---+-----+    +----+----+    +----+----+    +----+----+
  | 5 | next|---->| 10 | next|--->|12 | next|---->| 3 | next| --->
NULL
  +---+-----+    +----+----+    +----+----+    +----+----+
  */

  display(head);

  return 0;
}
```



Doubly linked list

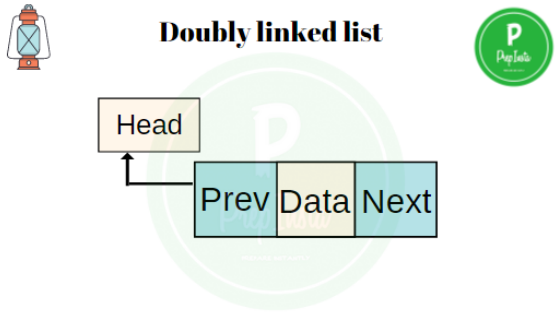
A doubly linked list is a kind of complex linked list, in which it contains the address of the previous node as well as the next node. A doubly linked list contains the following –

1. Data: The data present in the node.
2. Link (Previous) – A pointer to the previous node in the linked list.
3. Link (Next) – A pointer to the next node in the linked list
In this way, for a doubly-linked list we can travel in both directions for traversals.

Introduction to Doubly Linked List

What are Doubly Linked Lists?

Doubly linked lists are an extension to singly linked lists and provide a little more additional features and security to singly-linked lists. Let us see what these linked lists are –



Doubly Linked list

Doubly linked lists have the following –

Data: Like Singly-linked lists, it also contains data that is stored.
Pointer (next) – Contains the address of the next node in the doubly linked list
Pointer (previous) – Contains the address of the previous node in the doubly linked list
Apart from these basic terminologies are same – Node, Head, Tail

Doubly linked list

Advantages

1. It is better as unlike singly linked list, in a doubly-linked list we can traverse in both directions. Thus, if in case any pointer is lost we can still traverse.
Thus, in a Doubly Linked List we can traverse from Head to Tail as well as Tail to Head.
2. Delete operation is quicker if the pointer to the node to be deleted is given to us already.
3. Insertion is quicker in doubly-linked lists.

Disadvantages

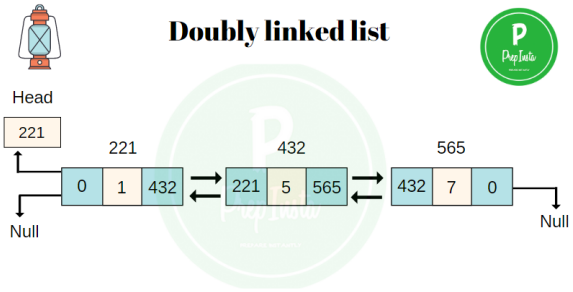
1. Extra space is required for the previous pointer for doubly-linked lists(DLL)
2. All operations require an additional modification of the previous pointer as well along with the next pointer.

Creation of Doubly linked lists

Now we are discussing How to create a Node for a Doubly linked list.

C Code:

```
struct Node {
 int Data;
 Struct Node* next;
 Struct Node*  prev;
};
```



Code for Insertion and Access for Linked list

```
//We are creating program for doubly linked list creation
#include <stdio.h>
#include <stdlib.h>
//stdlib used to provide a declaration of 'malloc'

// structure of linked list
struct Node {
  int data;
  struct Node* next;
  struct Node* prev;
  // Pointer pointing towards next node
};

//function to print the doubly linked list
void display(struct Node* node)
{
  struct Node *end;
  printf("List in Forward direction: ");
  while (node != NULL) {
    printf(" %d ", node->data);
    end = node;
    node = node->next;
  }

  printf("\nList in backward direction: ");

  while (end != NULL) {
    printf(" %d ", end->data);
    end = end->prev;
  }


}
```

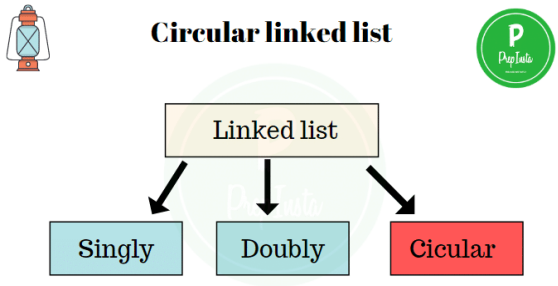
```
// main function
int main()
{
  //creating 4 pointers of type struct Node
  //So these can point to address of struct type variable
  struct Node* head = NULL;
  struct Node* node2 = NULL;
  struct Node* node3 = NULL;
  struct Node* node4 = NULL;

  // allocate 4 nodes in the heap
  head =  (struct Node*)malloc(sizeof(struct Node));
  node2 = (struct Node*)malloc(sizeof(struct Node));
  node3 = (struct Node*)malloc(sizeof(struct Node));
  node4 = (struct Node*)malloc(sizeof(struct Node));

  /* Using malloc method we have created 4 memory
blocks
  Each memory block is of type struct and has int data and
Pointer of type Node
  So it can point towards a node type struct.

    head            node2          node3            node4
     |              |              |              |
     |              |              |              |
+---++---+-----+   +----+----+----+   +----+----+----+
+----+----+----+
|prev|data|next|   |prev|data|next|   |prev|data|next|
|prev|data|next|
+---++---+-----+   +----+----+----+   +----+----+----+
+----+----+----+

 # As of now data has garbage value and its pointer
 points towards random addresses*/

  head->data = 5; // data set for head node
  head->next = node2; // next pointer assigned to address of
node2
  head->prev = NULL;

  node2->data = 10;
  node2->next = node3;
  node2->prev = head;

  node3->data = 12;
  node3->next = node4;
  node3->prev = node2;

  node4->data = 3;
  node4->next = NULL;
  node4->prev = node3;

  //last node assigned to Null as linked list ends here
/*
    head            node2          node3            node4
     |              |              |              |
     |              |              |              |
+---++---+-----+   +----+----+----+   +----+----+----+
+----+----+----+
|prev|  5 |next|--->|prev| 10 |next|--->|prev| 12
|next|--->|prev|  3  |next|
+---++---+-----+   +----+----+----+   +----+----+----+
+----+----+----+

*/


  display(head);

  return 0;
}
```

Circular linked list

This is a type of linked list in which the last node points to the starting node. There is no null at the end. A circular linked list can be a singly circular linked list and doubly circular linked list. In the Circular linked list there exist no nodes which have null at address space.

Introduction to Circular linked list

Circular Linked Lists - Circular linked lists are just like singly linked lists but all of its nodes are connected to form a circle



Circular linked list Advantages
The circular linked list is also two types.

1. Singly linked list
2. Doubly linked list

In a singly circular linked list, the address of the last node's next pointer rather than being NULL is pointed towards the address of the head node.

Similarly, in a doubly linked list, in addition to the address of the last node's next pointer being the address of head node. The previous pointer of the head node is provided to the address of the last node.

Advantages

1. Any node can be starting point and we can still traverse the whole list
2. It can also deem to be useful for implementation as queues as rather than maintaining the Front and Rear, we can use a circular linked list and just maintain the pointer to the first inserted node and it can simply be the next node to the last node.
3. Circular linked lists are commonly used in OS'es for the task that requires repeatedly to be executed by OS.

Disadvantages

1. Doubly Linked List is faster in getting previous node information due to previous pointer.
2. Doubly Linked List can traverse in both forward and backward direction.
3. Finding end of list and loop control is harder (no NULL to mark beginning and end).
4. The entire list can be traversed starting from any node (traverse means visit every node just once).

So, when we want to traverse the node in a circular linked list so we will traverse in a singly circular linked list so, it is just like the singly linked list where tail nodes hold the address of head node so traversal can be done circularly in only one direction.

And the Traversal in a doubly linked list can be done circularly in both directions.

Creation of Circular Linked Lists

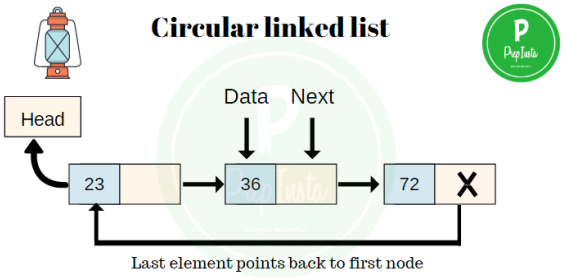
Creation of the list node is same as singly or doubly only implementation with the struct is different

Defining struct for Circular Linked List (Singly)

```
struct node{
  int data;
  struct node *next;
};
```

Defining struct for Circular Linked List (Doubly)

```
struct node{
  int data;
  struct node *next;
  struct node *prev;
};
```



Function to traverse in a Circular linked list

```
void display(struct Node *head)

{

  struct Node *temp = head;

  if (head != NULL)

  {

    do

    {

      printf("%d ", temp->data);

      temp = temp->next;

    }

    while (temp != head);

  }

}
```

Code of Circular linked list

```
//We are creating program for linked list creation
#include <stdio.h>
#include <stdlib.h>
//stdlib used to provide a declaration of 'malloc'
```

```
// structure of Circular linked list
struct Node {
  int data;
  struct Node* next;
  // Pointer pointing towards next node
};

//function to print the Circular linked list
void display(struct Node *head)
{
  struct Node *temp = head;
  if (head != NULL)
  {
    do
    {
      printf("%d ", temp->data);
      temp = temp->next;
    }
    while (temp != head);
  }
}

// main function
int main()
{
  //creating 4 pointers of type struct Node
  //So these can point to address of struct type variable
  struct Node* head = NULL;
  struct Node* node2 = NULL;
  struct Node* node3 = NULL;
  struct Node* node4 = NULL;

  // allocate 3 nodes in the heap
  head =  (struct Node*)malloc(sizeof(struct Node));
  node2 = (struct Node*)malloc(sizeof(struct Node));
  node3 = (struct Node*)malloc(sizeof(struct Node));
  node4 = (struct Node*)malloc(sizeof(struct Node));

  /* Using malloc method we have created 4 memory
blocks
  Each memory block is of type struct and has int data and
Pointer of type Node
  So it can point towards a node type struct.

    head          node2          node3           node4
     |             |             |              |
     |             |             |              |
  +---+-----+    +----+----+    +----+----+    +----+----+
  |data|next|    |data|next|    |data|next|    |data|next|
  +---+-----+    +----+----+    +----+----+    +----+----+

 # As of now data has garbage value and its pointer
 points towards random addresses*/

  head->data = 5; // data set for head node
  head->next = node2; // next pointer assigned to address of
node2

  node2->data = 10;
  node2->next = node3;

  node3->data = 12;
  node3->next = node4;

  node4->data = 3;
  node4->next = head;
  //last node assigned to head node as linked list ends here
  //this completes Circular linked list

  /* Finally Circular linked list looks like

    head          node2          node3           node4
     |             |             |              |
     |             |             |              |
  +---+-----+    +----+----+    +----+----+    +----+----+
  | 5 | next|---->| 10 | next|--->|12 | next|---->| 3 | next| ---> |
  +---+-----+    +----+----+    +----+----+    +----+----+      |
     ^                                                          |
     |                                                          |
     ------------------------------------------------------------
  */

  display(head);

  return 0;
}
```

Chapter 14- Linked List in C

Linked Lists in C Programming Language
Linked List in C is a linear type of data structures, which has some major advantages over arrays and other linear data structures. Linked Lists in C are used for storing the data in a contiguous manner. It is not necessary on the concept of linked lists that the address to the next node is provided in continuous manner.Linked is constructed of two parts-:

Node – It contains the data
Pointer – It contains the address of the next node

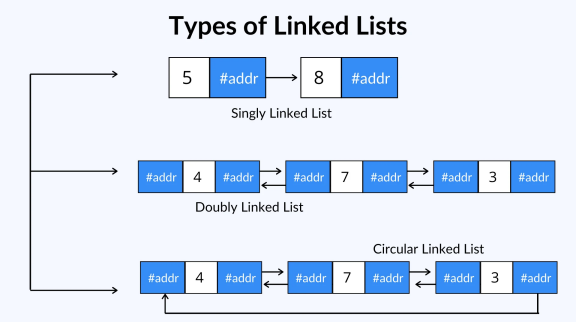
Why do we prefer Linked Lists ?
Linked Lists are a more preferable data structure to use than any other data structure if we are willing to store large amount of data, as insertion and deletion process in linked lists are fairly easy and less complicated in comparison with other data structures.

Linked List
Linked List is like an array is a linear data structure. However, the elements or data storage of linked list elements are not in a contiguous manner and we use pointers to create linearity in linked lists.×Dismiss alert
How to construct a Linked List in C ?
For constructing a Linked List in C we use user defined data type, i.e. we make a structure in C for using Linked List. We design a user defined struct data type, that contains a datatype, for storing the desired data and a pointer variable for storing the address of the next node in the Linked List.

Syntax for creating a node in Linked List in C
struct Node {
int data;
struct Node* next;
};
This code will create a new node in a Linked List which will be storing integer type of data

Advantages of using Linked List in C
It does not have a restriction of size, i.e; we do not have to declare the size of linked list before creating it.
There is no memory wastage, as we connect new nodes depending upon how much data do we have to store, and on deleting a data its node is also deleted.

Linked List does not need contiguous memory, i.e; if one node has an address of 1000 then the next node may have the address 2000.
Insertion and Deletion Operation in Linked Lists are pretty easy and less complicated.
Disadvantages of using Linked List in C
Although insertion and deletion are not a difficult task in Linked List, searching in Linked List is very difficult.
We cannot use efficient searches like Binary Search.
It takes more spaces, as it stores the address also.
Linked lists are not cache friendly as arrays are stored in contigious format they can be cached easily
Loss of data threat is there, if we lose one pointer location the rest of linked list can't be accessed.



Code for Implementing Linked List in C

```
//Linked List Implementation code
#include <stdio.h>
#include <stdlib.h>
//stdlib is included because we will be using  'malloc'

// creating a node in linked list
struct Node {
   int data;
   struct Node* next;
   // Pointer pointing towards next node
};

//function to print the linked list
void display(struct Node* node)
{
   while (node != NULL) {
      printf(" %d ", node->data);
      node = node->next;
   }
}

// main function
int main()
{
  //creating 4 pointers of type struct Node
```

```
    //So these can point to address of struct type variable
    struct Node* head = NULL;
    struct Node* node2 = NULL;
    struct Node* node3 = NULL;
    struct Node* node4 = NULL;

    // allocate 3 nodes in the heap
    head =  (struct Node*)malloc(sizeof(struct Node));
    node2 = (struct Node*)malloc(sizeof(struct Node));
    node3 = (struct Node*)malloc(sizeof(struct Node));
    node4 = (struct Node*)malloc(sizeof(struct Node));


    head->data = 10; // data set for head node
    head->next = node2; // next pointer assigned to address
of node2

    node2->data = 20;
    node2->next = node3;

    node3->data = 30;
    node3->next = node4;

    node4->data = 40;
    node4->next = NULL;


    display(head);

   return 0;
}
```

Output

```
10   20   30   40
```

Insertion operations on linked list
Insertion at beginning
Insertion at end
Insertion in between of nodes.
Deletion operations on linked list
Deletion at beginning
Deletion at end
Deletion of a specific node.
Other operations on linked list
Reverse a linked list
Search an element in linked list
Find middle of the linked list
Fold a linked list

Insertion at Beginning in Singly Linked List in C++

June 17, 2020

Insertion at begin in Singly linked list in C++ programming
How to perform insertion at beginning in Singly Linked List in C++?
Insertion at Beginning in Singly Linked List in C++, is one of the operations that we can perform on Linked List. Singly linked list in C++ are part of linked list and is a type of linear data structure. Linked is made up of two parts node and pointer where node contains the data and pointer contains the address of the next node.

To add a new node add beginning we need to allocate space for the new node first, then we have to store the address of previous head in the pointed of newly created node and make the new node as head of the linked list. In this way insertion at beginning is done in linked list.

Singly linked list definition in C++
Nodes of singly linked list is created by using the code mentioned besides.
This set of code will construct linked list by creating each node of the list.

```
class Node
{
   int data;
   Node *next;
};
```

Steps to insert an element at beginning of singly linked list
1.) Allocate space for a new node.

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

2.) In the space created for new node put the data in.

```
new_node->data = new_data
```

3.) Point the pointer of new node to the head of singly linked list

```
new_node->ptr = (*head);
```

4.) Make new node as head.

```
(*head) = new_node
```

Singly Linked List
We can traverse in only one direction in singly linked list as singly linked list does not contain previous pointer, it only has next pointer.×Dismiss alert
Insertin at beginning in singly linked list in C++
Algorithm to insert an element at beginning of singly linked list
IF PTR = NULL
EXIT
ELSE SET NEW_NODE = PTR
SET PTR = PTR → NEXT

SET NEW_NODE → DATA = VALUE
SET NEW_NODE → NEXT = HEAD
SET HEAD = NEW_NODE
EXIT

C++ program to insert an element at beginning of singly linked list

```
#include <iostream>
using namespace std;

struct node
{
   int num;
   node *nextptr;
}*stnode; //node constructed

void createList(int n);
void insertatBegin(int num);
void display();

int main()
{
   int n,num;

   cout<<"Enter the number of nodes: ";
   cin>>n;
   createList(n);
   cout<<"\nLinked list data: \n";
   display();
   cout<<"\nEnter data you want to insert at the beginning: ";
   cin>>num;
   insertatBegin(num);
   cout<<"\nLinked list after insertion: \n";
   display();

   return 0;
}
void createList(int n) //function to create linked list.
{
   struct node *frntNode, *tmp;
   int num, i;

   stnode = (struct node *)malloc(sizeof(struct node));
   if(stnode == NULL)
   {
      cout<<" Memory can not be allocated";
   }
   else
   {

      cout<<"Enter the data for node 1: ";
      cin>>num;
      stnode-> num = num;
      stnode-> nextptr = NULL; //Links the address field to NULL
      tmp = stnode;

      for(i=2; i<=n; i++)
      {
         frntNode = (struct node *)malloc(sizeof(struct node));


         if(frntNode == NULL) //If frntnode is null no memory cannot be alloted
         {
            cout<<" Memory can not be allocated";
            break;
         }
         else
         {
            cout<<"Enter the data for node "<<i<<": "; // Entering data in nodes.
            cin>>num;
            frntNode->num = num;
            frntNode->nextptr = NULL;
            tmp->nextptr = frntNode;
            tmp = tmp->nextptr;
         }
      }
   }
}

void insertatBegin(int num) //function to insert element at beginning
{
   struct node *frntNode;
   frntNode = (struct node*)malloc(sizeof(struct node));
   if(frntNode == NULL)
   {
      cout<<" Memory can not be allocated";
   }
   else
   {
      frntNode->num = num;        //Linkinking data
      frntNode->nextptr = stnode; //Linking address
      stnode = frntNode;
   }
}

void display() //function to print linked list
{
   struct node *tmp;
   if(stnode == NULL)
   {
      cout<<" No data found in the list";
   }
```

```
   else
   {
      tmp = stnode;
      cout<<"Linked List: ";
      while(tmp != NULL)
      {
         cout<<"\t"<<tmp->num;
         tmp = tmp->nextptr;
      }
   }
}
```

Output:

```
Enter the number of nodes: 4
Enter the data for node 1: 11
Enter the data for node 2: 22
Enter the data for node 3: 33
Enter the data for node 4: 44

Linked list data:
Linked List:          11          22          33          44
Enter data you want to insert at the beginning: 55

Linked list after insertion:
Linked List:          55          11          22          33
44
```

Insertion at End in Singly Linked List in C++

Algorithm for insertion at end in singly linked list in C++
How to insert element at End in Singly Linked List in C++?
To add a new node at end we need to allocate space for the new node first, then we have to store null in the pointer of newly created node and make the new node as tail of the linked list. In this way insertion at end is done in linked list.Insertion at end in singly linked list in C++, is one of the insertion operations that we can perform on Linked List. Singly linked list in C++ are part of linked list and is a type of linear data structure. Linked is made up of two parts node and pointer where node contains the data and pointer contains the address of the next node.

Singly linked list definition in C++
Nodes of singly linked list is created by using the code mentioned besides.
This set of code will construct linked list by creating each node of the list.

```
class Node
{
   int data;
   Node *next;
};
```

Steps to insert an element at end of singly linked list
1.) Allocate space for a new node.

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

2.) In the space created for new node put the data in.

```
new_node->data = new_data
```

3.) Point the pointer of new node to null

```
new_node->ptr = NULL;
```

4.) If the Linked List is empty, then make the new node as head

```
 if (*head == NULL)
     *head = new_node;
```

5.) Traverse till last node and point the pointer of this node to new node.

```
while (last->next != NULL)
       last = last->next;
       last->ptr = new_node;
```

Did you know!
We can traverse in only one direction in singly linked list as singly linked list does not contain previous pointer, it only has next pointer.×Dismiss alert

Algorithm for Insertion at end in singly linked list in c++

```
IF PTR = NULL
EXIT
SET NEW_NODE = PTR
SET PTR = PTR – > NEXT
SET NEW_NODE – > DATA = VAL
SET NEW_NODE – > NEXT = NULL
SET PTR = HEAD
WHILE PTR – > NEXT != NULL
SET PTR = PTR – > NEXT
SET PTR – > NEXT = NEW_NODE
EXIT
```

C++ program to insert an element at end of singly linked list

```
#include <iostream>
using namespace std;
//creating the structure of a new node
struct node
{
   int num;
   node *nextptr;
```

```
}*stnode;

void createList(int n) //function to create a linked list.
{
   struct node *frntNode, *tmp;
   int num, i;

   stnode = (struct node *)malloc(sizeof(struct node));
   if(stnode == NULL)
   {
      cout<<"Memory can not be allocated";
   }
   else
   {

      cout<<"Enter the data for node 1: ";
      cin>>num;
      stnode-> num = num;
      stnode-> nextptr = NULL; //Links the address field to
NULL
      tmp = stnode;

      for(i=2; i<=n; i++)
      {
         frntNode = (struct node *)malloc(sizeof(struct
node));


         if(frntNode == NULL) //If frntnode is null no
memory cannot be allotted
         {
            cout<<" Memory can not be allocated";
            break;
         }
         else
         {
            cout<<"Enter the data for node "<<i<<": "; //
Entering data in nodes.
            cin>>num;
            frntNode->num = num;
            frntNode->nextptr = NULL;
            tmp->nextptr = frntNode;
            tmp = tmp->nextptr;
         }
      }
   }
}

void display() //function to print linked list
{
   struct node *tmp;
   if(stnode == NULL)
   {
      cout<<" No data found in the list";
   }
   else
   {
      tmp = stnode;
      cout<<"Linked List: ";
      while(tmp != NULL)
      {
         cout<<"\t"<<tmp->num;
         tmp = tmp->nextptr;
      }
   }
}

void insertatEnd(int num)//function to add element at the
end
{
   struct node *frntNode, *tmp;
   frntNode = (struct node*)malloc(sizeof(struct node));
   if(frntNode == NULL)
   {
      cout<<"Memory can not be allocated.";
   }
   else
   {
      frntNode->num = num;     //Linking the data part
      frntNode->nextptr = NULL;
      tmp = stnode;
      while(tmp->nextptr != NULL)
         tmp = tmp->nextptr;
      tmp->nextptr = frntNode;  //Linking the address part
   }
}


int main()
{
   int n,num;

   cout<<"Enter the number of nodes: ";
   cin>>n;
   createList(n);
   cout<<"\nLinked list data: \n";
   display();
   cout<<"\nEnter data you want to insert at the end: ";
   cin>>num;
   insertatEnd(num);
   cout<<"\nLinked list after insertion: \n";
   display();

   return 0;
}
```

Insertion in between of nodes in singly linked list in C++

Insertion in between the nodes in singly linked list in C++
How to perform insertion in between nodes in Singly Linked List in C++?
Insertion in between of nodes in singly linked lists in C++ is a multiple step process. In this article we will learn these steps and algorithms for data insertion at desired positions.

We can also perform insertion at beginning that is inserting an element in the beginning of the linked list and insertion at end of the linked list. You can learn about this by clicking the button below

Insertion at beginning
Steps to insert an element in between of nodes in singly linked list
Following steps are followed for insertion of an element at nth position in singly linked list.

1.)If there is no data in the head then exit

```
If(head==NULL)
return
```

2.) Allocate space for a new node.

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

3.) In the space created for the new node, put the data in.

```
new_node->data = new_data
```

4.)Traverse till the desired position and make the pointer of the new node where the previous node is pointing.

```
new_node->ptr= prev_node->ptr
```

5.) Now point the pointer of the previous node to the new node.

```
prev_node->ptr = new_node
```

Defining a singly linked list in C++
Nodes of singly linked lists are created by using the code mentioned besides.
This set of code will construct a linked list by creating each node of the list.

```
class Node
{
   int data;
   Node *next;
};
```

Algorithm for insertion in between the nodes in singly linked list in c++
Algorithm to insert an element at nth position of singly linked list

```
IF PTR = NULL
EXIT
SET NEW_NODE = PTR
NEW_NODE → DATA = VAL
SET TEMP = HEAD
SET I = 0
REPEAT
TEMP = TEMP → NEXT
IF TEMP = NULL
EXIT
PTR → NEXT = TEMP → NEXT
TEMP → NEXT = PTR
SET PTR = NEW_NODE
EXIT
```

Program for insertion in between of nodes in singly linked list in C++

```
#include <iostream>
using namespace std;

struct node
{
   int num;
   node *nextptr;
}*stnode; //node constructed

void createList(int n);
void insertNode(int num, int pos);
void display();

int main()
{
   int n,num,pos;

   cout<<"Enter the number of nodes: ";
   cin>>n;
   createList(n);
   cout<<"\nLinked list data: \n";
   display();
   cout<<"\nEnter data you want to insert at the nth
position: ";
   cin>>num;
   cout<<"Enter the position to insert new node : ";
   cin>>pos;
   insertNode( num, pos);
   cout<<"\nLinked list after insertion: \n";
   display();

  return 0;
}
void createList(int n) //function to create linked list.
{
   struct node *frntNode, *tmp;
   int num, i;

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

```
   if(stnode == NULL)
   {
      cout<<"Memory can not be allocated";
   }
   else
   {

      cout<<"Enter the data for node 1: ";
      cin>>num;
      stnode-> num = num;
      stnode-> nextptr = NULL; //Linking the address field
to NULL
      tmp = stnode;

      for(i=2; i<=n; i++)
      {
         frntNode = (struct node *)malloc(sizeof(struct
node));


         if(frntNode == NULL) //If frntnode is null no
memory cannot be allotted
         {
            cout<<"Memory can not be allocated";
            break;
         }
         else
         {
            cout<<"Enter the data for node "<<i<<": "; //
Entering data in nodes.
            cin>>num;
            frntNode->num = num;
            frntNode->nextptr = NULL;
            tmp->nextptr = frntNode;
            tmp = tmp->nextptr;
         }
      }
   }
}

void insertNode(int num, int pos)//fuction to add node in
desired position
{
   int i;
   struct node *frntNode, *tmp;
   frntNode = (struct node*)malloc(sizeof(struct node));
   if(frntNode == NULL)
   {
     cout<<"Memory can not be allocated.";
   }
   else
   {
      frntNode->num = num; //Linking  the data
      frntNode->nextptr = NULL;
      tmp = stnode;
      for(i=2; i<=pos-1; i++)
      {
         tmp = tmp->nextptr;

         if(tmp == NULL)
            break;
      }
      if(tmp != NULL)
      {
         frntNode->nextptr = tmp->nextptr;  //Linking the
address part of new node
         tmp->nextptr = frntNode;
      }
      else
      {
         cout<<"Data cannot be entered in that particular
position\n";
      }
   }
}

void display() //function to print linked list
{
   struct node *tmp;
   if(stnode == NULL)
   {
      cout<<" No data found in the list";
   }
   else
   {
      tmp = stnode;
      cout<<"Linked List: ";
      while(tmp != NULL)
      {
         cout<<"\t"<<tmp->num;
         tmp = tmp->nextptr;
      }
   }
}
```

Output:

```
Enter the number of nodes: 4
Enter the data for node 1: 11
Enter the data for node 2: 22
Enter the data for node 3: 33
Enter the data for node 4: 44

Linked list data:
Linked List:          11          22          33          44
Enter data you want to insert at the nth position: 55
Enter the position to insert new node : 4

Linked list after insertion: Linked List:  11          22
33          55          44
```