**Introduction to Data Structures**

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

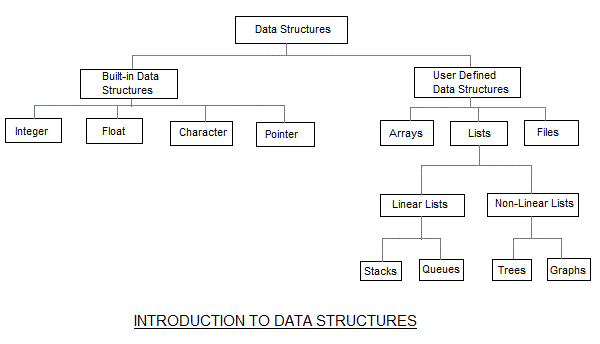
**Basic types of Data Structures**

As we discussed above, anything that can store data can be called as a data strucure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

* Linked List
* Tree
* Graph
* Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



**What is Algorithm ?**

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

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

1. Time Complexity
2. Space Complexity

**Space Complexity**

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

* **Instruction Space :** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
* **Data Space :** Its the space required to store all the constants and variables value.
* **Environment Space :** Its the space required to store the environment information needed to resume the suspended function.

**Time Complexity**

Time Complexity is a way to represent the amount of time needed by the program to run to completion. We will study this in details in our section.

**NOTE:** Before going deep into data structure, you should have a good knowledge of programming either in C or in C++ or Java.

**Time Complexity of Algorithms**

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the **big O notation**.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

**Calculating Time Complexity**

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

statement;

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

for(i=0; i < N; i++)

{

statement;

}

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

for(i=0; i < N; i++)

{

for(j=0; j < N;j++)

{

statement;

}

}

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N \* N.

while(low <= high)

{

mid = (low + high) / 2;

if (target < list[mid])

high = mid - 1;

else if (target > list[mid])

low = mid + 1;

else break;

}

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

void quicksort(int list[], int left, int right)

{

int pivot = partition(list, left, right);

quicksort(list, left, pivot - 1);

quicksort(list, pivot + 1, right);

}

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N\*log( N )**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

**NOTE :** In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

**Types of Notations for Time Complexity**

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

**Understanding Notations of Time Complexity with Example**

**O(expression)** is the set of functions that grow slower than or at the same rate as expression.

**Omega(expression)** is the set of functions that grow faster than or at the same rate as expression.

**Theta(expression)** consist of all the functions that lie in both O(expression) and Omega(expression).

Suppose you've calculated that an algorithm takes f(n) operations, where,

f(n) = 3\*n^2 + 2\*n + 4. // n^2 means square of n

Since this polynomial grows at the same rate as **n^2**, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of **n^2**, the time complexity can be best represented as **Theta(n^2)**.

**Introduction to Sorting**

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

**Sorting** arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

* Roll No.
* Name
* Age
* Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

**Sorting Efficiency**

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters. First parameter is the execution time of program, which means time taken for execution of program. Second is the space, which means space taken by the program.

**Types of Sorting Techniques**

There are many types of Sorting techniques, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next sections.

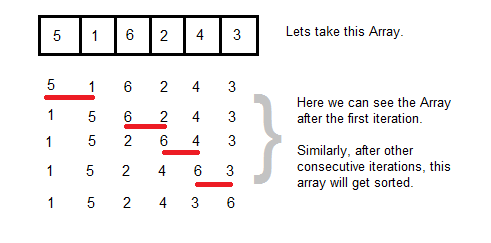
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

**Bubble Sorting**

**Bubble Sort** is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with **N** number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.



**Sorting using Bubble Sort Algorithm**

Let's consider an array with values {5, 1, 6, 2, 4, 3}

int a[6] = {5, 1, 6, 2, 4, 3};

int i, j, temp;

for(i=0; i<6; i++)

{

for(j=0; j<6-i-1; j++)

{

if( a[j] > a[j+1])

{

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

//now you can print the sorted array after this

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.

Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and wew can jump out of the for loop.

int a[6] = {5, 1, 6, 2, 4, 3};

int i, j, temp;

for(i=0; i<6; i++)

{

int **flag** = 0; *//taking a flag variable*

for(j=0; j<6-i-1; j++)

{

if( a[j] > a[j+1])

{

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

**flag** = 1; *//setting flag as 1, if swapping occurs*

}

}

if(!**flag**) *//breaking out of for loop if no swapping takes place*

{

break;

}

}

In the above code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

**Complexity Analysis of Bubble Sorting**

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

(n-1)+(n-2)+(n-3)+.....+3+2+1

Sum = n(n-1)/2

i.e O(n2)

Hence the complexity of Bubble Sort is **O(n2)**.

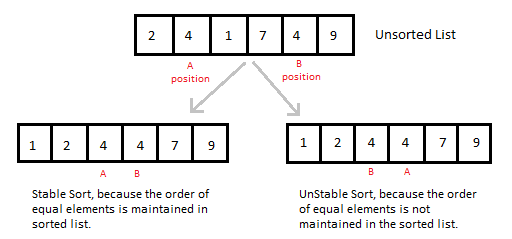
The main advantage of Bubble Sort is the simplicity of the algorithm.Space complexity for Bubble Sort is **O(1)**, because only single additional memory space is required for **temp** variable

**Best-case** Time Complexity will be **O(n)**, it is when the list is already sorted.

**Insertion Sorting**

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, inerstion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys



**How Insertion Sorting Works**



**Sorting using Insertion Sort Algorithm**

int a[6] = {5, 1, 6, 2, 4, 3};

int i, j, key;

for(i=1; i<6; i++)

{

key = a[i];

j = i-1;

while(j>=0 && key < a[j])

{

a[j+1] = a[j];

j--;

}

a[j+1] = key;

}

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable **key**, in which we put each element of the array, in each pass, starting from the second element, that is **a[1]**.

Then using the while loop, we iterate, until **j** becomes equal to zero or we find an element which is greater than **key**, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

**Insertion Sorting in C++**

#include <stdlib.h>

#include <iostream.h>

using namespace std;

*//member functions declaration*

void **insertionSort**(int arr[], int length);

void **printArray**(int array[],int size);

int **main**() {

int array[5]= {5,4,3,2,1};

**insertionSort**(*array*,5);

return 0;

}

void **insertionSort**(int arr[], int length) {

int i, j ,tmp;

for (i = 1; i < length; i++) {

j = i;

while (j > 0 && arr[j - 1] > arr[j]) {

tmp = arr[j];

arr[j] = arr[j - 1];

arr[j - 1] = tmp;

j--;

}

printArray(arr,5);

}

}

void **printArray**(int array[], int size){

cout<< "Sorting tha array using Insertion sort... ";

int j;

for (j=0; j < size;j++)

for (j=0; j < size;j++)

cout <<" "<< array[j];

cout << endl;

}

**Complexity Analysis of Insertion Sorting**

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n)

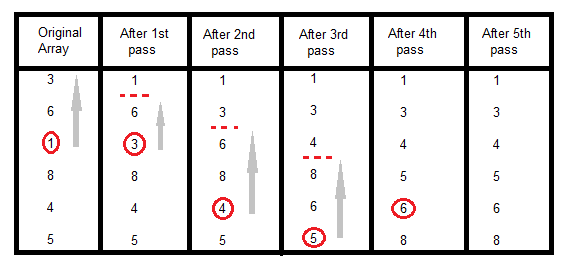
**Average Time Complexity :** O(n2)

**Space Complexity :** O(1)

**Selection Sorting**

Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

**How Selection Sorting Works**



In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 nad 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.

**Sorting using Selection Sort Algorithm**

void selectionSort(int a[], int size)

{

int i, j, min, temp;

for(i=0; i < size-1; i++ )

{

min = i; //setting min as i

for(j=i+1; j < size; j++)

{

if(a[j] < a[min]) //if element at j is less than element at min position

{

min = j; //then set min as j

}

}

temp = a[i];

a[i] = a[min];

a[min] = temp;

}

}

**Complexity Analysis of Selection Sorting**

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n2)

**Average Time Complexity :** O(n2)

**Space Complexity :** O(1)

**Quick Sort Algorithm**

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less aditional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

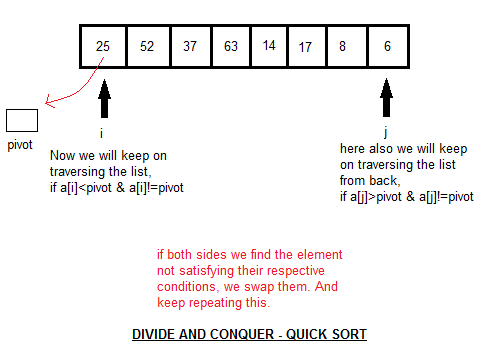
1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hnece after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

**How Quick Sorting Works**



**Sorting using Quick Sort Algorithm**

/\* a[] is the array, p is starting index, that is 0,

and r is the last index of array. \*/

void **quicksort**(int a[], int p, int r)

{

if(p < r)

{

int q;

q = partition(a, p, r);

quicksort(a, p, q);

quicksort(a, q+1, r);

}

}

int **partition**(int a[], int p, int r)

{

int i, j, pivot, temp;

pivot = a[p];

i = p;

j = r;

while(1)

{

while(a[i] < pivot && a[i] != pivot)

i++;

while(a[j] > pivot && a[j] != pivot)

j--;

if(i < j)

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

else

{

return j;

}

}

}

**Complexity Analysis of Quick Sort**

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(n log n)

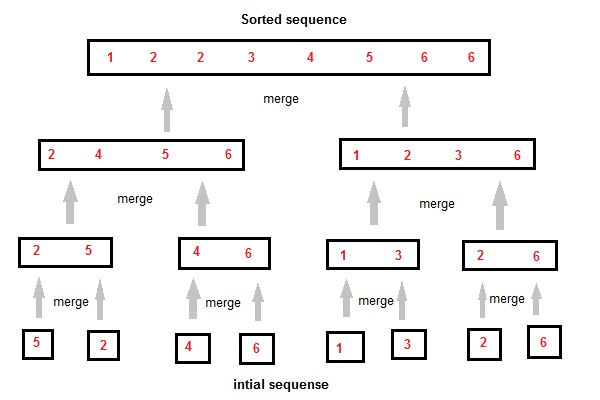
* Space required by quick sort is very less, only O(n log n) additional space is required.
* Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

**Merge Sort Algorithm**

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divides the list into two halves. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

**How Merge Sort Works**



Like we can see in the above example, merge sort first breaks the unsorted list into sorted sublists, and then keep merging these sublists, to finlly get the complete sorted list.

**Sorting using Merge Sort Algorithm**

/\* a[] is the array, p is starting index, that is 0,

and r is the last index of array. \*/

Lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

void **mergesort**(int a[], int p, int r)

{

int q;

if(p < r)

{

q = floor( (p+r) / 2);

mergesort(a, p, q);

mergesort(a, q+1, r);

merge(a, p, q, r);

}

}

void **merge**(int a[], int p, int q, int r)

{

int b[5]; //same size of a[]

int i, j, k;

k = 0;

i = p;

j = q+1;

while(i <= q && j <= r)

{

if(a[i] < a[j])

{

b[k++] = a[i++]; // same as b[k]=a[i]; k++; i++;

}

else

{

b[k++] = a[j++];

}

}

while(i <= q)

{

b[k++] = a[i++];

}

while(j <= r)

{

b[k++] = a[j++];

}

for(i=r; i >= p; i--)

{

a[i] = b[--k]; // copying back the sorted list to a[]

}

}

**Complexity Analysis of Merge Sort**

**Worst Case Time Complexity :** O(n log n)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(n)

* Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
* It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.
* It is the best Sorting technique for sorting **Linked Lists**.

**Heap Sort Algorithm**

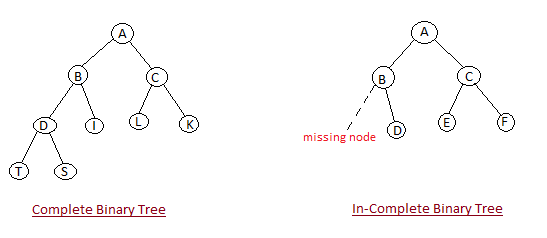
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

* Creating a Heap of the unsorted list.
* Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

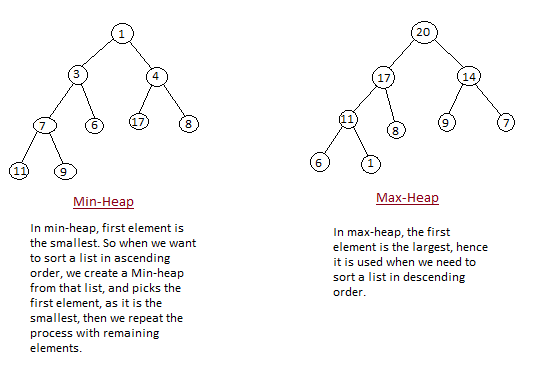
**What is a Heap ?**

Heap is a special tree-based data structure, that satisfies the following special heap properties :

1. **Shape Property :** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.



1. **Heap Property :** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smalled than their child nodes, heap is called **Min-Heap**.



**How Heap Sort Works**

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

In the below algorithm, initially **heapsort()** function is called, which calls **buildheap()** to build heap, which inturn uses **satisfyheap()** to build the heap.

**Sorting using Heap Sort Algorithm**

/\* Below program is written in C++ language \*/

void heapsort(int[], int);

void buildheap(int [], int);

void satisfyheap(int [], int, int);

void main()

{

int a[10], i, size;

cout << "Enter size of list"; // less than 10, because max size of array is 10

cin >> size;

cout << "Enter" << size << "elements";

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

{

cin >> a[i];

}

heapsort(a, size);

getch();

}

void **heapsort**(int a[], int length)

{

buildheap(a, length);

int heapsize, i, temp;

heapsize = length - 1;

for( i=heapsize; i >= 0; i--)

{

temp = a[0];

a[0] = a[heapsize];

a[heapsize] = temp;

heapsize--;

satisfyheap(a, 0, heapsize);

}

for( i=0; i < length; i++)

{

cout << "\t" << a[i];

}

}

void **buildheap**(int a[], int length)

{

int i, heapsize;

heapsize = length - 1;

for( i=(length/2); i >= 0; i--)

{

satisfyheap(a, i, heapsize);

}

}

void **satisfyheap**(int a[], int i, int heapsize)

{

int l, r, largest, temp;

l = 2\*i;

r = 2\*i + 1;

if(l <= heapsize && a[l] > a[i])

{

largest = l;

}

else

{

largest = i;

}

if( r <= heapsize && a[r] > a[largest])

{

largest = r;

}

if(largest != i)

{

temp = a[i];

a[i] = a[largest];

a[largest] = temp;

satisfyheap(a, largest, heapsize);

}

}

**Complexity Analysis of Heap Sort**

**Worst Case Time Complexity :** O(n log n)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(n)

* Heap sort is not a Stable sort, and requires a constant space for sorting a list.
* Heap Sort is very fast and is widely used for sorting.

**Searching Algorithms on Array**

Before studying searching algorithms on array we should know what is an algorithm?

An **algorithm** is a step-by-step procedure or method for solving a problem by a computer in a given number of steps. The steps of an algorithm may include repetition depending upon the problem for which the algorithm is being developed. The algorithm is written in human readable and understandable form. To search an element in a given array, it can be done in two ways Linear search and Binary search.

**Linear Search**

A linear search is the basic and simple search algorithm. A linear search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.

**Example with Implementation**

To search the element 5 it will go step by step in a sequence order.



function **findIndex**(values, target)

{

for(var i = 0; i < values.length; ++i)

{

if (values[i] == target)

{

return i;

}

}

return -1;

}

//*call the function findIndex with array and number to be searched*

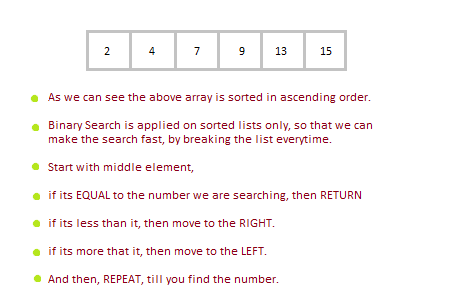
findIndex([ 8 , 2 , 6 , 3 , 5 ] , 5) ;

**Binary Search**

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

**Example with Implementation**

To search an element 13 from the sorted array or list.



function findIndex(values, target)

{

return **binarySearch**(values, target, 0, values.length - 1);

};

function binarySearch(values, target, start, end) {

if (start > end) { return -1; } *//does not exist*

var middle = Math.floor((start + end) / 2);

var value = values[middle];

if (value > target) { return **binarySearch**(values, target, start, middle-1); }

if (value < target) { return **binarySearch**(values, target, middle+1, end); }

return middle; *//found!*

}

findIndex([2, 4, 7, 9, 13, 15], 13);

In the above program logic, we are first comparing the middle number of the list, with the target, if it matches we return. If it doesn't, we see whether the middle number is greater than or smaller than the target.

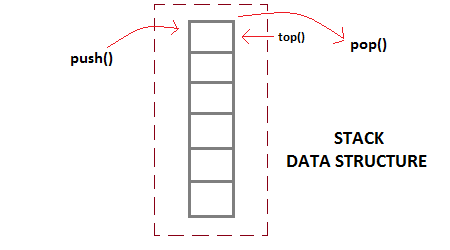
If the Middle number is greater than the Target, we start the binary search again, but this time on the left half of the list, that is from the start of the list to the middle, not beyond that.

If the Middle number is smaller than the Target, we start the binary search again, but on the right half of the list, that is from the middle of the list to the end of the list.

# ======Data Structures======

**Stacks**

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.



**Basic features of Stack**

1. Stack is an ordered list of similar data type.
2. Stack is a **LIFO** structure. (Last in First out).
3. **push()** function is used to insert new elements into the Stack and **pop()** is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

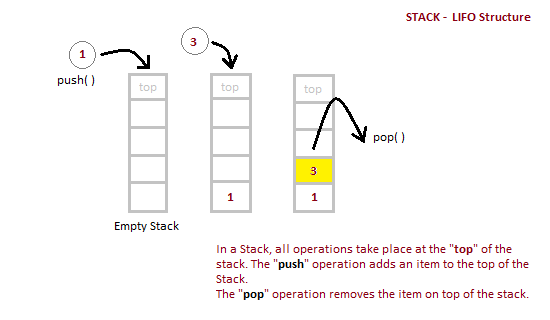
**Applications of Stack**

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like : **Parsing**, **Expression Conversion**(Infix to Postfix, Postfix to Prefix etc) and many more.

**Implementation of Stack**

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



/\* Below program is written in C++ language \*/

Class **Stack**

{

int top;

public:

int a[10]; //Maximum size of Stack

Stack()

{

top = -1;

}

};

void Stack::**push(int x)**

{

if( top >= 10)

{

cout << "Stack Overflow";

}

else

{

a[++top] = x;

cout << "Element Inserted";

}

}

int Stack::**pop()**

{

if(top < 0)

{

cout << "Stack Underflow";

return 0;

}

else

{

int d = a[top--];

return d;

}

}

void Stack::**isEmpty()**

{

if(top < 0)

{

cout << "Stack is empty";

}

else

{

cout << "Stack is not empty";

}

}

|  |  |
| --- | --- |
| **Position of Top** | **Status of Stack** |
| -1 | Stack is Empty |
| 0 | Only one element in Stack |
| N-1 | Stack is Full |
| N | Overflow state of Stack |

**Analysis of Stacks**

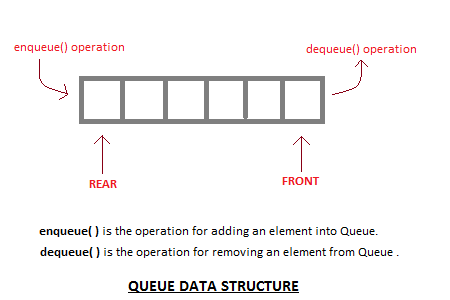
Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

* **Push Operation** : O(1)
* **Pop Operation** : O(1)
* **Top Operation** : O(1)
* **Search Operation** : O(n)

**Queue Data Structures**

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of exisiting element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

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



**Basic features of Queue**

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO( First in First Out ) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek( )** function is oftenly used to return the value of first element without dequeuing it.

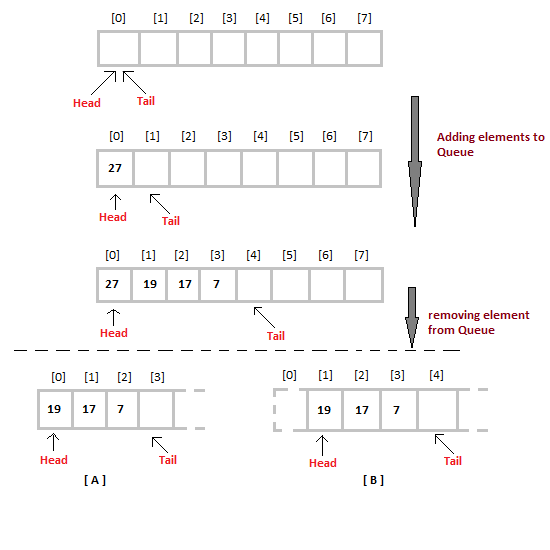
**Applications of Queue**

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

**Implementation of Queue**

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array. Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



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

In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element. In approach [B] there is no such overhead, but whener we move head one position ahead, after removal of first element, the size on Queue is reduced by one space each time.

/\* Below program is wtitten in C++ language \*/

#define SIZE 100

class **Queue**

{

int a[100];

int rear; //same as tail

int front; //same as head

public:

Queue()

{

rear = front = -1;

}

void enqueue(int x); //declaring enqueue, dequeue and display functions

int dequeue();

void display();

}

void Queue :: **enqueue**(int x)

{

if( rear = SIZE-1)

{

cout << "Queue is full";

}

else

{

a[++rear] = x;

}

}

int queue :: **dequeue**()

{

return a[++front]; //following approach [B], explained above

}

void queue :: **display**()

{

int i;

for( i = front; i <= rear; i++)

{

cout << a[i];

}

}

To implement approach [A], you simply need to change the dequeue method, and include a for loop which will shift all the remaining elements one position.

return a[0]; //returning first element

for (i = 0; i < tail-1; i++) //shifting all other elements

{

a[i]= a[i+1];

tail--;

}

**Analysis of Queue**

* Enqueue : **O(1)**
* Dequeue : **O(1)**
* Size : **O(1)**

**Queue Data Structure using Stack**

A Queue is defined by its property of FIFO, which means First in First Out, i.e the element which is added first is taken out first. Hence we can implement a Queue using Stack for storage instead of array.

For performing **enqueue** we require only one stack as we can directly **push** data into stack, but to perform **dequeue** we will require two Stacks, because we need to follow queue's FIFO property and if we directly **pop** any data element out of Stack, it will follow LIFO approach(Last in First Out).

**Implementation of Queue using Stacks**

In all we will require two Stacks, we will call them InStack and OutStack.

class Queue {

public:

Stack S1, S2;

*//defining methods*

void **enqueue**(int x);

int **dequeue**();

}

We know that, Stack is a data structure, in which data can be added using **push()** method and data can be deleted using **pop()** method. To learn about Stack, follow the link : [Stack Data Structure](http://www.studytonight.com/data-structures/stack-data-structure)

**Adding Data to Queue**

As our Queue has Stack for data storage in place of arrays, hence we will be adding data to Stack, which can be done using the push() method, hence :

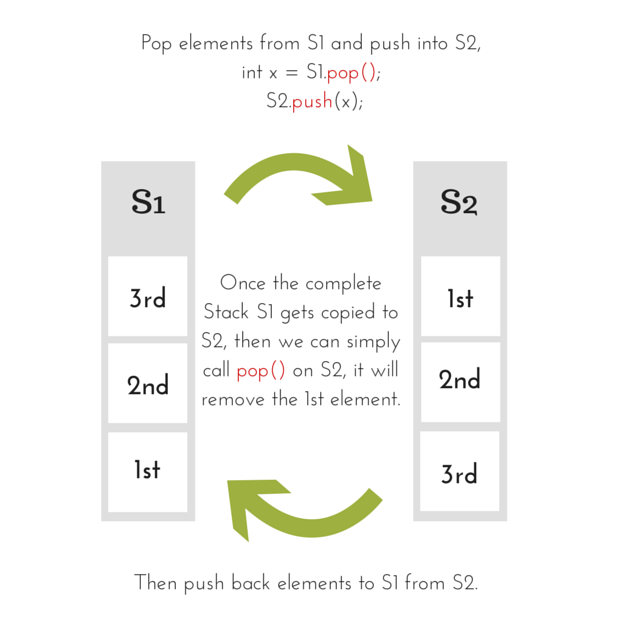
void Queue :: **enqueue**(int x) {

S1.**push**(x);

}

**Removing Data from Queue**

When we say remove data from Queue, it always means taking out the First element first and so on, as we have to follow the FIFO approach. But if we simply perform S1.pop() in our **dequeue** method, then it will remove the Last element first. So what to do now?



int Queue :: dequeue() {

while(S1.isEmpty()) {

x = S1.**pop**();

S2.**push**();

}

*//removing the element*

x = S2.pop();

while(!S2.isEmpty()) {

x = S2.**pop**();

S1.**push**(x);

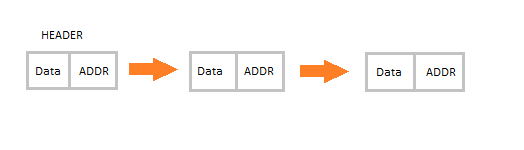
}

return x;

}

**Introduction to Linked Lists**

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.



**Advantages of Linked Lists**

* They are a dynamic in nature which allocates the memory when required.
* Insertion and deletion operations can be easily implemented.
* Stacks and queues can be easily executed.
* Linked List reduces the access time.

**Disadvantages of Linked Lists**

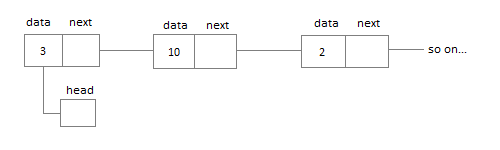
* The memory is wasted as pointers require extra memory for storage.
* No element can be accessed randomly; it has to access each node sequentially.
* Reverse Traversing is difficult in linked list.

**Applications of Linked Lists**

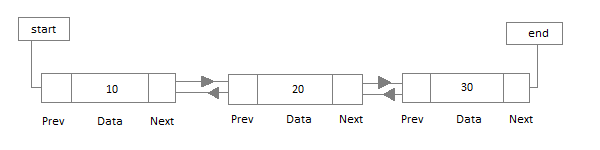
* Linked lists are used to implement stacks, queues, graphs, etc.
* Linked lists let you insert elements at the beginning and end of the list.
* In Linked Lists we don’t need to know the size in advance.

**Types of Linked Lists**

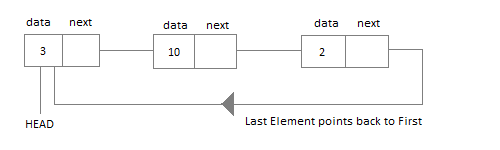
* **Singly Linked List :** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



* **Doubly Linked List :** In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



* **Circular Linked List :** In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



**Linear Linked List**

The element can be inserted in linked list in 2 ways :

* Insertion at beginning of the list.
* Insertion at the end of the list.

We will also be adding some more useful methods like :

* Checking whether Linked List is empty or not.
* Searching any element in the Linked List
* Deleting a particular Node from the List

Before inserting the node in the list we will create a class **Node**. Like shown below :

class **Node** {

public:

int data;

*//pointer to the next node*

node\* next;

**node**() {

data = 0;

next = NULL;

}

**node**(int x) {

data = x;

next = NULL;

}

}

We can also make the properties data and next as private, in that case we will need to add the getter and setter methods to access them. You can add the getters and setter like this :

int **getData**() {

return data;

}

void **setData**(int x) {

this.data = x;

}

node\* **getNext**() {

return next;

}

void **setNext**(node \*n) {

this.next = n;

}

Node class basically creates a node for the data which you enter to be included into Linked List. Once the node is created, we use various functions to fit in that node into the Linked List.

**Linked List class**

As we are following the complete OOPS methodology, hence we will create a separate class for **Linked List**, which will have all its methods. Following will be the Linked List class :

class **LinkedList** {

public:

node \*head;

*//declaring the functions*

*//function to add Node at front*

int addAtFront(node \*n);

*//function to check whether Linked list is empty*

int isEmpty();

*//function to add Node at the End of list*

int addAtEnd(node \*n);

*//function to search a value*

node\* search(int k);

*//function to delete any Node*

node\* deleteNode(int x);

**LinkedList**() {

head = NULL;

}

}

**Insertion at the Beginning**

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

int LinkedList :: **addAtFront**(node \*n) {

int i = 0;

*//making the next of the new Node point to Head*

n**->**next = head;

*//making the new Node as Head*

head = n;

i++;

*//returning the position where Node is added*

return i;

}

**Inserting at the End**

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

int LinkedList :: **addAtEnd**(node \*n) {

*//If list is empty*

if(head == NULL) {

*//making the new Node as Head*

head = n;

*//making the next pointe of the new Node as Null*

n**->**next = NULL;

}

else {

*//getting the last node*

node \*n2 = **getLastNode**();

n2**->**next = n;

}

}

node\* LinkedList :: **getLastNode**() {

*//creating a pointer pointing to Head*

node\* ptr = head;

*//Iterating over the list till the node whose Next pointer points to null*

*//Return that node, because that will be the last node.*

while(ptr**->**next!=NULL) {

*//if Next is not Null, take the pointer one step forward*

ptr = ptr**->**next;

}

return ptr;

}

**Searching for an Element in the List**

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

node\* LinkedList :: **search**(int x) {

node \*ptr = head;

while(ptr != NULL && ptr**->**data != x) {

*//until we reach the end or we find a Node with data x, we keep moving*

ptr = ptr**->**next;

}

return ptr;

}

**Deleting a Node from the List**

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

* If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
* If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

node\* LinkedList :: **deleteNode**(int x) {

*//searching the Node with data x*

node \*n = **search**(x);

node \*ptr = head;

if(ptr == n) {

ptr**->**next = n**->**next;

return n;

}

else {

while(ptr**->**next != n) {

ptr = ptr**->**next;

}

ptr**->**next = n**->**next;

return n;

}

}

**Checking whether the List is empty or not**

We just need to check whether the **Head** of the List is NULL or not.

int LinkedList :: **isEmpty**() {

if(head == NULL) {

return **1**;

}

else { return **0**; }

}

Now you know a lot about how to handle List, how to traverse it, how to search an element. You can yourself try to write new methods around the List.

If you are still figuring out, how to call all these methods, then below is how your main() method will look like. As we have followed OOP standards, we will create the objects of **LinkedList** class to initialize our List and then we will create objects of **Node** class whenever we want to add any new node to the List.

int main() {

LinkedList L;

*//We will ask value from user, read the value and add the value to our Node*

int x;

cout << "Please enter an integer value : ";

cin >> x;

Node \*n1;

*//Creating a new node with data as x*

n1 = new Node(x);

*//Adding the node to the list*

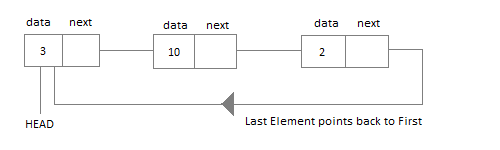
L.addAtFront(n1);

}

Similarly you can call any of the functions of the LinkedList class, add as many Nodes you want to your List.

## Circular Linked List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



#### Application of Circular Linked List

* The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
* Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
* Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

### Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last **Node** will have it's **next** point to the **Head** of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.

So this will be oue Node class, as we have already studied in the lesson, it will be used to form the List.

class **Node** {

public:

int data;

*//pointer to the next node*

node\* next;

**node**() {

data = 0;

next = NULL;

}

**node**(int x) {

data = x;

next = NULL;

}

}

#### Circular Linked List

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods.

class **CircularLinkedList** {

public:

node \*head;

*//declaring the functions*

*//function to add Node at front*

int addAtFront(node \*n);

*//function to check whether Linked list is empty*

int isEmpty();

*//function to add Node at the End of list*

int addAtEnd(node \*n);

*//function to search a value*

node\* search(int k);

*//function to delete any Node*

node\* deleteNode(int x);

**CircularLinkedList**() {

head = NULL;

}

}

#### Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the fisrt Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

int CircularLinkedList :: **addAtFront**(node \*n) {

int i = 0;

*/\* If the list is empty \*/*

if(head == NULL) {

n**->**next = head;

*//making the new Node as Head*

head = n;

i++;

}

else {

n->next = head;

*//get the Last Node and make its next point to new Node*

Node\* last = **getLastNode**();

last->next = n;

*//also make the head point to the new first Node*

head = n;

i++;

}

*//returning the position where Node is added*

return i;

}

#### Insertion at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

int CircularLinkedList :: **addAtEnd**(node \*n) {

*//If list is empty*

if(head == NULL) {

*//making the new Node as Head*

head = n;

*//making the next pointer of the new Node as Null*

n**->**next = NULL;

}

else {

*//getting the last node*

node \*last = **getLastNode**();

last**->**next = n;

*//making the next pointer of new node point to head*

n**->**next = head;

}

}

#### Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

node\* CircularLinkedList :: **search**(int x) {

node \*ptr = head;

while(ptr != NULL && ptr**->**data != x) {

*//until we reach the end or we find a Node with data x, we keep moving*

ptr = ptr**->**next;

}

return ptr;

}

#### Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

* If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
* If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
* If the Node is at the end, then remove it and make the new last node point to the head.

node\* CircularLinkedList :: **deleteNode**(int x) {

*//searching the Node with data x*

node \*n = **search**(x);

node \*ptr = head;

**if**(ptr == NULL) {

cout << "List is empty";

return NULL;

}

**else if**(ptr == n) {

ptr**->**next = n**->**next;

return n;

}

**else** {

while(ptr**->**next != n) {

ptr = ptr**->**next;

}

ptr**->**next = n**->**next;

return n;

}

}