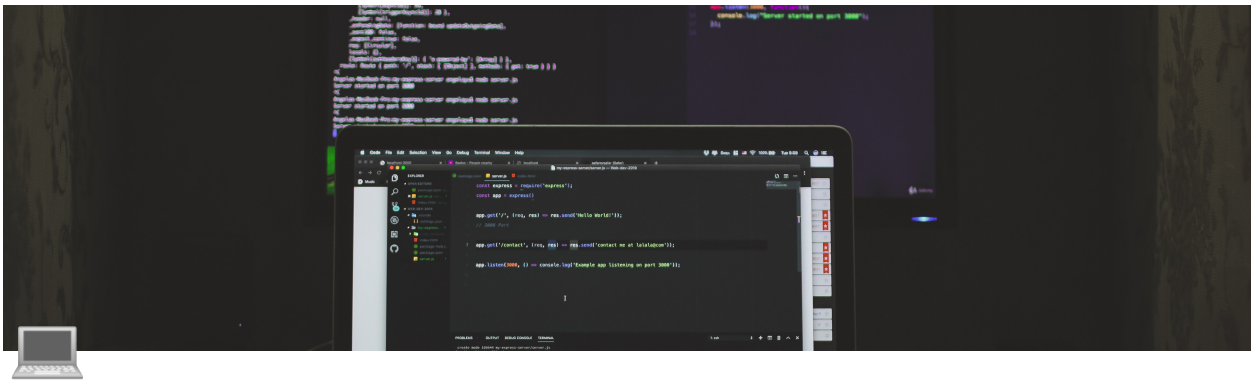


Data Structures & Algorithms

Created	@March 16, 2021 8:03 PM
Last Edited Time	@January 30, 2022 2:04 PM
Type	
Status	Done
Created By	
Last Edited By	
Stakeholders	

What is Data Structure?

A data storage format. It is the collection of values and the format they are stored in the relationships between the values in the collection as well as the operations applied on the data stored in the structure.

Abstract Data Type

An Abstract Data Type(ADT) is an abstraction of a data structure which provides only the interface to which a data structure must adhere to. The interface does not give any specific details about how something should be implemented or in what programming language.

Abstraction (ADT)	Implementation
List	Dynamin Array Linked List
Queue	Linked List based Queue Array based Queue Stack based Queue
Map	Tree Map Hash Map/ Hash Table
for example: Vehicle	Golf Cart, car, motorcycle

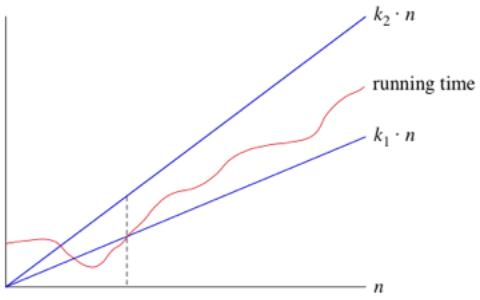
Computational Complexity Analysis

Asymptotic Notation

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—without getting mired in details that complicate our understanding. When we drop the constant coefficients and the less significant terms, we use asymptotic notation. We'll see three forms of it: big-\ThetaΘ\Theta notation, big-O notation, and big-\OmegaΩ\Omega notation.

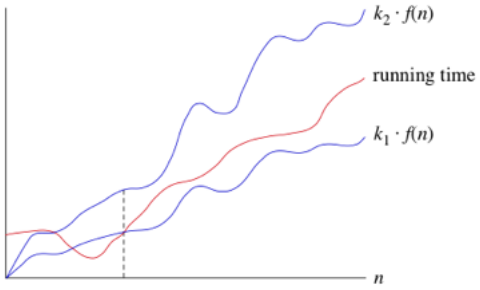
Big-Theta Notation

When we say that a particular running time is Θ(n), we're saying that once n gets large enough, the running time is at least k1.n and at most k2.n for some constants k1 and k2. Here's how to think of Θ(n):



For small values of n, we don't care how the running time compares with k1.n or k2.n. But once n gets large enough - on or to the right of the dashed line-the running time must be sandwiched between k1.n and k2.n. As long as these constants k1 and k2 exist, we say that running time is Θ(n).

We are not restricted to just n in big-Θ notation. We can use any function, such as n^2, nlogn, or any other function of n. Here's how to think of a running time that is Θ(f(n)) for some function f(n):



Once n gets large enough, the running time is between k1.f(n) and k2.f(n).

In practice, we just drop constant factors and low-order terms. Another advantage of using big-Θ notation is that we don't have to worry about which time units we're using.

When we use big-Theta notation, we're saying that we have asymptotically tight bound on the running time. "Asymptotically" because it matters for only large values of n. "Tight bound" because we've nailed the running time to within a constants factor above below.

Here's a list of functions in asymptotic notation that we often encounter when analyzing algorithms, ordered by slowest to faster growth:

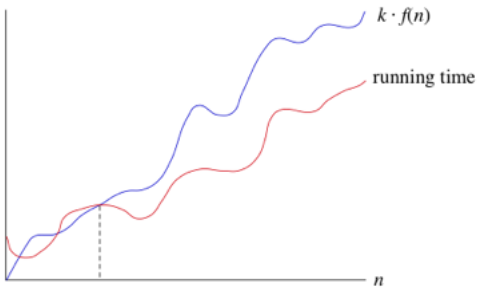
1. Theta(1)
2. Theta(logn)
3. Theta(n)
4. Theta(nlogn)
5. Theta(n^2)
6. Theta(n^2logn)

7. Theta(n^3)
8. Theta(2^n)
9. Theta(n!)

Big-O Notation

It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions.

If a running time is O(f(n)), then for large enough n, the running time is at most k.f(n) for some constant k. Here's how to think of a running time that is O(f(n)):



We say that the running time is "big-O of f(n)". We use big-O notation for asymptotic upper bounds, since it bounds the growth of the running time from above for large enough input sizes.

Because big-O notation gives only an asymptotic upper bound, and not an asymptotically tight bound, we can make statements that at first glance seem incorrect, but are technically correct. For example, it is absolutely correct to say that binary search runs in O(n) time. That's because the running time grows no faster than a constant times n. In fact, it grows slower.

Big-O Notation gives an upper bound of the complexity in the worst case, helping to qualifying performance as the input size becomes arbitrarily large.

n - the size of the input.

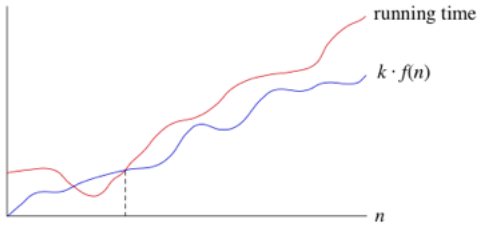
Complexities ordered in from smallest to largest

Aa Name	Tags
Constant time	O(1)
Logarithmic Time	O(log(n))
Linear Time	O(n)
Linearithmic Time	O(nlog(n))
Quadric Time	O(n^2)
Cubic Time	O(n^3)
Exponential Time	O(b^n), b>1
Factorial time	O(n!)

Big-Ω Notation:

Sometimes, we want to say that an algorithm takes at least a certain amount of time, without providing an upper bound. We use big-Ω notation.

If a running time is Ω(f(n)), then for large enough n, the running time is at least k.f(n) for some constant k. Here's how to think of a running time that is Ω(f(n)):



We say that the running time is "big-Ω of f(n)". We use big-Ω notation for asymptotic lower bounds, since it bounds the growth of the running time from below for large enough input sizes.

Static and Dynamic Arrays

When and Where is a static array is used?

1. Storing and accessing sequential data
2. Temporarily storing objects
3. Used by IO routines as buffers
4. Lookup tables and inverse lookup tables
5. Can be used to return multiple values from a function
6. Used in dynamic programming to cache answers to subproblems

Complexity

Static vs. Dynamic

Aa operations	Static Array	Dynamic Array
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	NA	O(n)
Appending	NA	O(1)
Deletion	NA	O(n)

Arrays in Java

An array is a group of like-typed variables that are referred to by a common name.

- In Java all arrays are dynamically allocated
- Since arrays are objects in java, we can find their length using the object property **length.**
- A java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.

- Java array can be also be used as a static field, a local variable or a method parameter.
- The direct superclass of an array type is Object

Array can contain primitives (int, char, etc.) as well as object (or non-primitive) references of a class depending on the definition of the array. In case of primitive data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in heap segment.

Instantiating an Array in Java

When an array is declared, only a reference of array is created. To actually create or give memory to array, you create an array like this,

```
var_name = new type[size];
"""
for example:
int intArray[] = new int[20];
"""
```

> The elements in the array allocated by new will automatically be initialized to zero(for numeric types), false (for boolean), or null (for reference type).

Multidimensional Arrays

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension. Examples:

```
int[][] intArray =  new int[10][20]; // a 2D array or matrix
int[][][] intArray = new int[10][20][15]; // a 3D array
```

```
class multiDimensional
{
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

        // printing 2D array
        for (int i=0; i< 3 ; i++)
        {
            for (int j=0; j < 3 ; j++)
                System.out.print(arr[i][j] + " ");

            System.out.println();
        }
    }
}
```

Array Members

Arrays are object of a class and direct superclass of arrays is class Object. The members of an array type are all of the following:

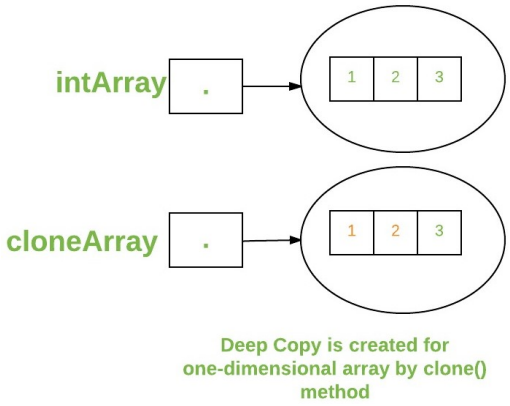
- The public final field length, which contains the number of components of the array. Length may be positive or zero.
- All the members inherited from class Object; the only method of Object that is not inherited is its **clone** method.

- The public method **clone()**, which overrides clone method in class Object and throw no checked exceptions.

Cloning of Arrays

1. When you clone a single dimensional array, such as Object[], a “deep copy” is performed with the new array containing copies of the original array’s elements as opposed to references.

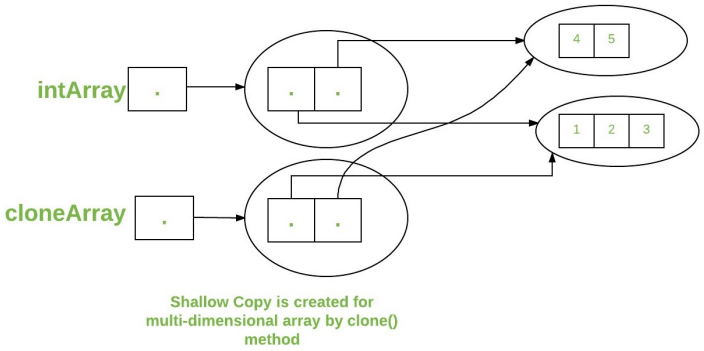
```
class Test
{
    public static void main(String args[])
    {
        int intArray[] = {1,2,3};

        int cloneArray[] = intArray.clone();

        // will print false as deep copy is created
        // for one-dimensional array
        System.out.println(intArray == cloneArray);

        for (int i = 0; i < cloneArray.length; i++) {
            System.out.print(cloneArray[i]+" ");
        }
    }
}

//output:
"""
false
1 2 3
"""
```



2. A clone of a multi-dimensional array (like Object[][]) is a “shallow copy” however, which is to say that it creates only a single new array with each element array a reference to an original element array, but subarrays are shared.

```
class Test
{
    public static void main(String args[])
    {
        int intArray[][] = {{1,2,3},{4,5}};

        int cloneArray[][] = intArray.clone();

        // will print false
        System.out.println(intArray == cloneArray);

        // will print true as shallow copy is created
        // i.e. sub-arrays are shared
        System.out.println(intArray[0] == cloneArray[0]);
        System.out.println(intArray[1] == cloneArray[1]);

    }
```

```
}

"""
output:

false
true
true
"""
```



Dynamic Array

How can we implement a dynamic Arrays?

One way is to use a static array!

1. Create a static array with an initial capacity
2. Add elements to the underlying static array, keeping track of the numbers of elements
3. If adding another element will exceed the capacity, then create a new static array with twice the capacity and copy the original elements into it.

Implementation of Dynamic Array using Static Array in Java

```
package com.Arrays;

import java.net.StandardSocketOptions;
import java.util.Iterator;
import java.util.Arrays;

import static java.util.Arrays.copyOf;

public class DynamicIntArray{

    private int[] arr;
    private int len;
    private int capacity = 0;

    public DynamicIntArray() {
        this.capacity = 1;
    }

    public DynamicIntArray(int capacity) {
        if(capacity < 0)
            throw new IllegalArgumentException("Illegal Capacity " + capacity);
        this.capacity = capacity;
        arr = new int[capacity];
    }
```

```
public DynamicIntArray(int[] array) {
    if(array == null)
        throw new IllegalArgumentException("Array cannot be null");
    arr = copyOf(array, array.length);
    capacity = len = array.length;
}

public int size() {
    return len;
}

public boolean isEmpty() {
    return len == 0;
}

public int get(int index) {
    return this.arr[index];
}

public void set(int index, int data) {
    this.arr[index] = data;
}

public void append(int data) {
    if(this.len+1>this.capacity) {
        if(this.capacity == 0)
            capacity = 1;
        else capacity *= 2;
        arr = copyOf(arr, capacity);
    }
    arr[len++] = data;
}

public void removeAt(int rmIndex) {
    if(rmIndex >= this.len || rmIndex < 0)
        throw new IndexOutOfBoundsException();

    for(int i=rmIndex; i<this.len; i++) {
        arr[i] = arr[i + 1];
    }
    this.len--;
}

public boolean remove(int elem) {
    for(int i=0; i<this.len; i++) {
        if(this.arr[i] == elem) {
            this.removeAt(i);
            return true;
        }
    }
    return false;
}

public void reverse() {
    for(int i=0; i<this.len/2; i++) {
        int temp = this.arr[i];
        this.arr[i] = this.arr[len-i-1];
        this.arr[len-i-1] = temp;
    }
}

@Override
public String toString() {
    return "DynamicIntArray{" +
            "arr=" + Arrays.toString(arr) +
            ", len=" + len +
            ", capacity=" + capacity +
            '}';
}

public static void main(String[] args){
    DynamicIntArray dynamicArray = new DynamicIntArray(2);
    dynamicArray.append(1);
    dynamicArray.append(2);
    dynamicArray.append(3);
    dynamicArray.append(4);
    dynamicArray.append(5);
    dynamicArray.append(6);
    dynamicArray.append(7);
    dynamicArray.append(8);
```

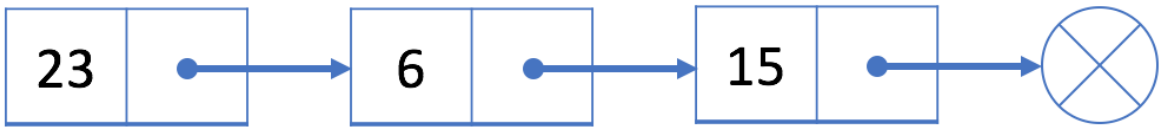
```
        dynamicArray.append(9);
//        dynamicArray.append(10);
        dynamicArray.removeAt(4);
        dynamicArray.remove(4);

        System.out.println(dynamicArray.toString());
    }
}
```

Singly and Doubly Linked List

What is Linked List ?

A linked list is a sequential list of nodes that hold data which point to other nodes also containing data. Each node in a singly-linked list contains not only the value but also `a reference field` to link to the next node. By this way, the singly-linked list organizes all the nodes in a sequence.

Here is an example of a singly-linked list:



The blue arrows show how nodes in a singly linked list are combined together.

Where are linked lists used?

- Used in many list, Queue & Stack implementations.
- Great for creating circular lists.
- Can easily model real world objects such as trains.
- Used in separate chaining, which is present certain Hashtable implementations to deal with hashing collisions.
- Often used in the implementations of adjacency lists of graphs.

Advantages of Linked List:

1. **Dynamic Data Structure:** Can Grow and Sink at the runtime by allocating and deallocating memory. There is no need to initilise
2. **No memory wastage:** Efficient memory utilization can be achieved since the size of the linked list increases and decreases at runtime so there is no memory wastage.
3. **Implementation:** Linear Data Structure like stack and queues are often easily implemented using linked list
4. **Insertion and Deletion operation:** There is no need to shift element to left or right after insertion or deletion.

Disadvantages of Linked List:

1. Memory Usage: In a Linked List a pointer is also required to store the address of the next element and it requires extra memory intself.

2. Traversal: Traversing a linked list is little bit costly in terms of time as compared to array. Direct access to an element is not possible in a linked list as in a array by index.
3. Reverse Traversing: In Singly Linked List it is not possible to traverse the list in reverse order but in doubly linked list it is possible, but doubly linked list requires more memory than singly linked list as it contains two pointer which is previous and the next.
4. Random Access: Random access to the node is not possible in the linked list due to its dynamic memory allocation.

Terminology

Head: The first node in a linked list

Tail: The last node in a linked list

Pointer: Reference to another node

Node: An object containing data and pointes

Node Structure of Singly Linked List

```
class LinkedList {
    Node head; // head of the list

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

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

In most cases, we will use the `head` node (the first node) to represent the whole list.

Designing a Linked List

Design your implementation of the linked list. You can choose to use a singly or doubly linked list. A node in a singly linked list should have two attributes: `val` and `next`. `val` is the value of the current node, and `next` is a pointer/reference to the next node. If you want to use the doubly linked list, you will need one more attribute `prev` to indicate the previous node in the linked list. Assume all nodes in the linked list are **0-indexed**.

Implement the `MyLinkedList` class:

- `MyLinkedList()` Initializes the `MyLinkedList` object.
- `int get(int index)` Get the value of the `index th` node in the linked list. If the index is invalid, return `1`.
- `void addAtHead(int val)` Add a node of value `val` before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.
- `void addAtTail(int val)` Append a node of value `val` as the last element of the linked list.
- `void addAtIndex(int index, int val)` Add a node of value `val` before the `index th` node in the linked list. If `index` equals the length of the linked list, the node will be appended to the end of the linked list. If `index` is greater than the length, the node **will not be inserted**.
- `void deleteAtIndex(int index)` Delete the `index th` node in the linked list, if the index is valid.

Example 1:

```
Input
["MyLinkedList", "addAtHead", "addAtTail", "addAtIndex", "get", "deleteAtIndex", "get"]
[[], [1], [3], [1, 2], [1], [1], [1]]
```

```
Output
[null, null, null, null, 2, null, 3]

Explanation
MyLinkedList myLinkedList = new MyLinkedList();
myLinkedList.addAtHead(1);
myLinkedList.addAtTail(3);
myLinkedList.addAtIndex(1, 2);    // linked list becomes 1->2->3
myLinkedList.get(1);              // return 2
myLinkedList.deleteAtIndex(1);    // now the linked list is 1->3
myLinkedList.get(1);              // return 3
```

Constraints:

- `0 <= index, val <= 1000`
- Please do not use the built-in LinkedList library.
- At most `2000` calls will be made to `get`, `addAtHead`, `addAtTail`, `addAtIndex` and `deleteAtIndex`.

```
class MyLinkedList {

    private SinglyLinkedListNode head;
    private SinglyLinkedListNode tail;
    private int length;

    class SinglyLinkedListNode {
        int val;
        SinglyLinkedListNode next;

        public SinglyLinkedListNode(int val) {
            this.val = val;
            this.next = null;
        }
    }
    /** Initialize your data structure here. */
    public MyLinkedList() {

    }

    /** Get the value of the index-th node in the linked list. If the index is invalid, return -1. */
    public int get(int index) {
        if(head != null){
            SinglyLinkedListNode currNode = head;
            int currIndex = 0;
            while(currNode != null) {
                if(currIndex == index)
                    return currNode.val;
                currNode = currNode.next;
                currIndex++;
            }
        }
        return -1;
    }

    /** Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of th
    public void addAtHead(int val) {

        SinglyLinkedListNode newNode = new SinglyLinkedListNode(val);
        if(isEmpty()) {
            head = tail = newNode;
        } else {
            newNode.next = head;
            head = newNode;
        }
        length++;
    }

    /** Append a node of value val to the last element of the linked list. */
    public void addAtTail(int val) {
        SinglyLinkedListNode newNode = new SinglyLinkedListNode(val);
        if(isEmpty()) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
```

```
            tail = newNode;
        }
        length++;
    }

    /** Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be
    public void addAtIndex(int index, int val) {
        if(index > length || index < 0) {
            return;
        }
        else if(index == 0) {
            addAtHead(val);
        }
        else if(index == length){
            addAtTail(val);
        }
        else {
            SinglyLinkedListNode newNode = new SinglyLinkedListNode(val);
            SinglyLinkedListNode curr = head;
            int currIndex = 0;
            while(currIndex != index-1) {
                curr = curr.next;
                currIndex++;
            }
            newNode.next = curr.next;
            curr.next = newNode;
            length++;
        }

    }

    /** Delete the index-th node in the linked list, if the index is valid. */
    public void deleteAtIndex(int index) {
        if(index >= length || index < 0) {
            return;
        }
        else if(index == 0) {
            deleteFirst();
        }
        else if(index == length-1){
            deleteLast();
        }
        else {
            SinglyLinkedListNode curr = head;
            int currIndex = 0;
            while(currIndex < index-1) {
                curr = curr.next;
                currIndex++;
            }
            curr.next = curr.next.next;
            length--;
        }
    }

    private boolean isEmpty() {
        return head == null;
    }

    private void deleteFirst() {
        if(isEmpty()) return;

        else if(head == tail) {
            head = tail = null;
            return;
        }
        SinglyLinkedListNode curr = head;
        head = curr.next;
        curr = null;
        length--;
    }

    private void deleteLast(){

        if(isEmpty()) return;

        else if(head == tail) {
            head = tail = null;
            return;
        }

        SinglyLinkedListNode curr = head;
```

```
        while(curr.next.next != null) {
            curr = curr.next;
        }
        curr.next = null;
        tail = curr;
        length--;
    }
}

/**
 * Your MyLinkedList object will be instantiated and called as such:
 * MyLinkedList obj = new MyLinkedList();
 * int param_1 = obj.get(index);
 * obj.addAtHead(val);
 * obj.addAtTail(val);
 * obj.addAtIndex(index,val);
 * obj.deleteAtIndex(index);
 */
```

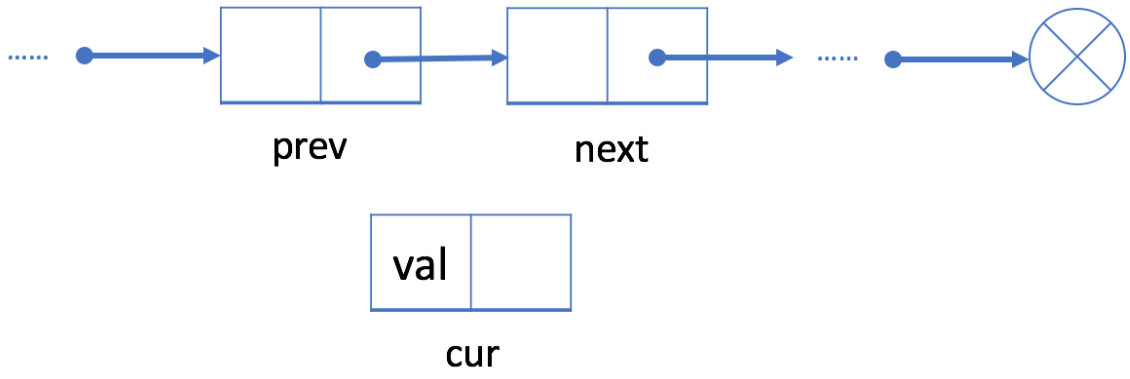
Operations

Unlike the array, we are not able to access a random element in a singly-linked list in constant time. If we want to get the ith element, we have to traverse from the head node one by one. It takes us `O(N)` time on average to `visit an element by index`, where N is the length of the linked list.

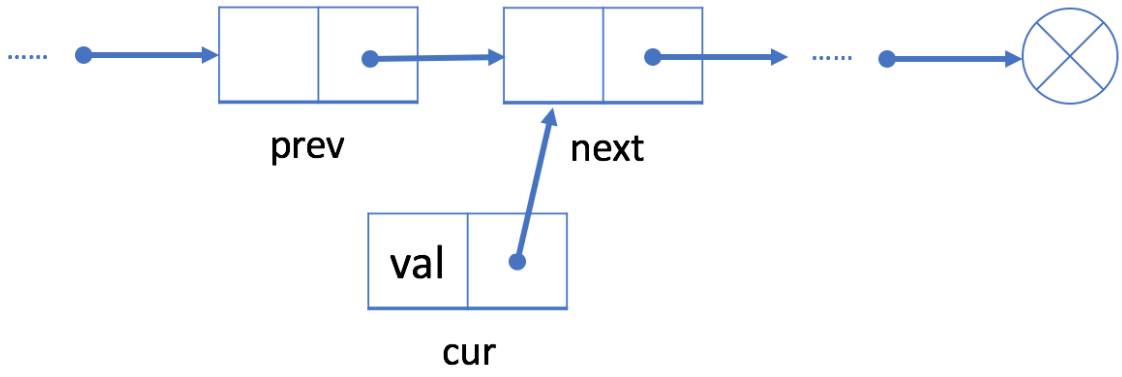
1. Add Operation

If we want to add a new value after a given node `prev`, we should:

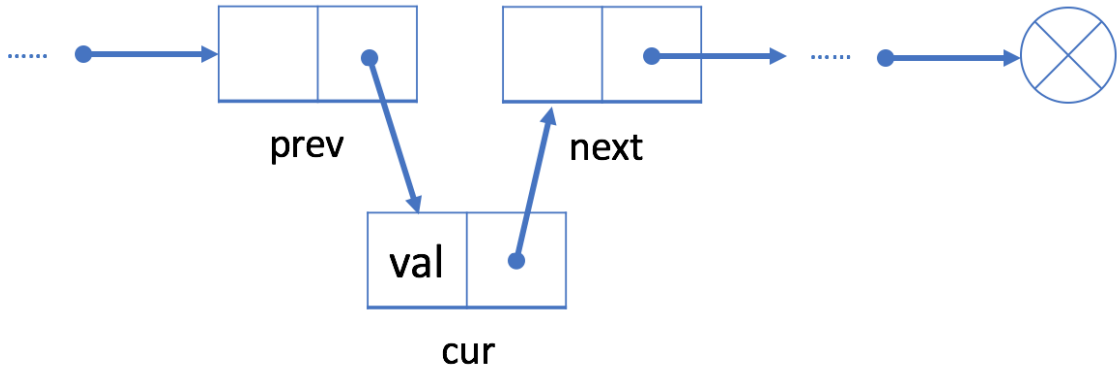
1. Initialize a new node `cur` with the given value;



2. Link the "next" field of `cur` to prev's next node `next`;



3. Link the "next" field in `prev` to `cur`.



Unlike an array, we don't need to move all elements past the inserted element. Therefore, you can insert a new node into a linked list in `O(1)` time complexity, which is very efficient.

Add Node at the beginning of the Linked List

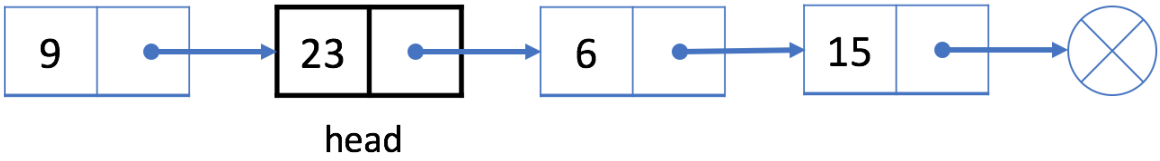
As we know, we use the head node `head` to represent the whole list.

So it is essential to update `head` when adding a new node at the beginning of the list.

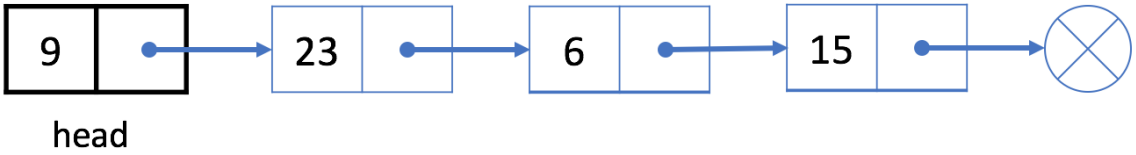
1. Initialize a new node `cur`;
2. Link the new node to our original head node `head`.
3. Assign `cur` to `head`.

For example, let's add a new node 9 at the beginning of the list.

1. We initialize a new node 9 and link node 9 to current head node 23.



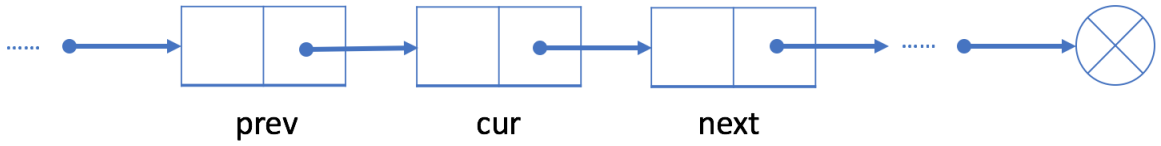
2. Assign node 9 to be our new head.



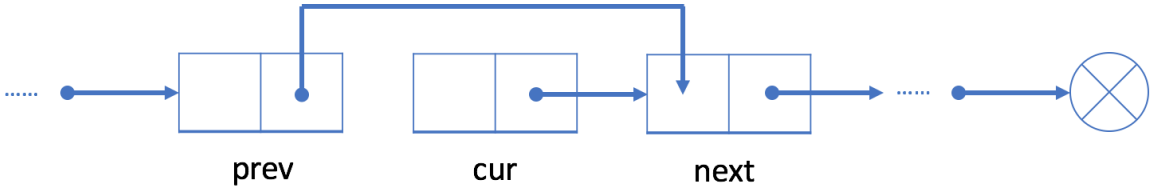
2. Delete operation

If we want to delete an existing node `cur` from the singly linked list, we can do it in two steps:

1. Find cur's previous node `prev` and its next node `next`;



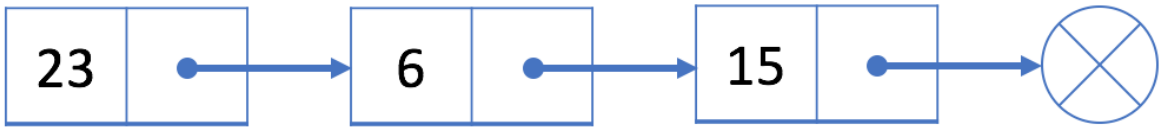
2. Link `prev` to cur's next node `next`.



In our first step, we need to find out `prev` and `next`. It is easy to find out `next` using the reference field of `cur`. However, we have to traverse the linked list from the head node to find out `prev` which will take `O(N)` time on average, where N is the length of the linked list. So the time complexity of deleting a node will be `O(N)`.

The space complexity is `O(1)` because we only need constant space to store our pointers.

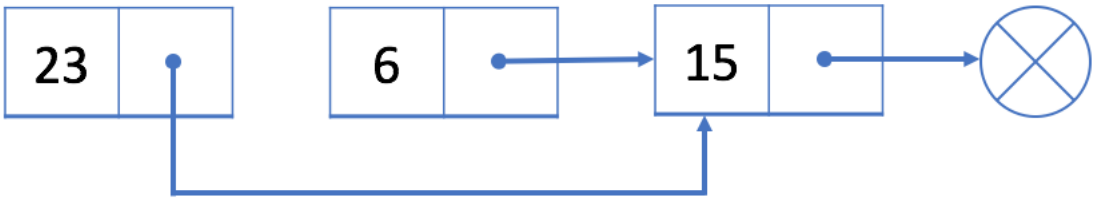
An Example



Let's try to delete node 6 from the singly linked list above.

1. Traverse the linked list from the head until we find the previous node `prev` which is node 23

2. Link `prev` (node 23) with `next` (node 15)

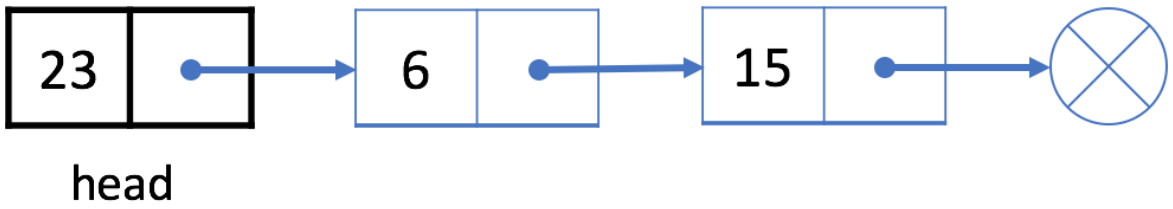


Node 6 is not in our singly linked list now.

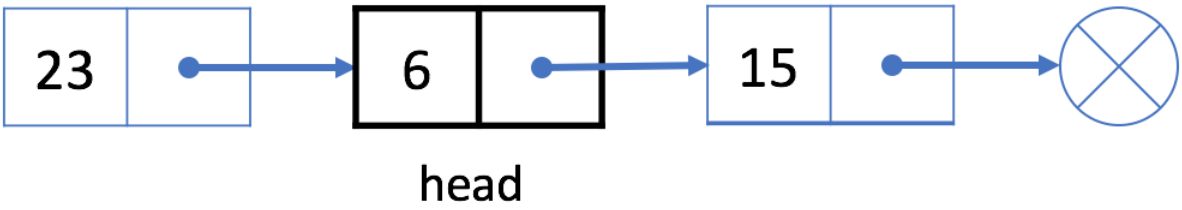
Delete the First Node

If we want to delete the first node, the strategy will be a little different.

As we mentioned before, we use the head node `head` to represent a linked list. Our head is the black node 23 in the example below.



If we want to delete the first node, we can simply `assign the next node to head`. That is to say, our head will be node 6 after deletion.



The linked list begins at the head node, so node 23 is no longer in our linked list.

Doubly Linked List

A Doubly Linked List contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.

Advantages of Doubly Linked List over singly Linked List

1. A DLL can be traversed in both forward and backward direction
2. The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3. We can quickly insert a new node before a given node.

Disadvantages of Doubly Linked List over singly Linked List

1. Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though
2. All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

   Insertion:

   a. At the front of the DLL

   b. After a given node

   c. At the end of the DLL

   d. Before a given node

Doubly Linked List Implementation

```
class MyLinkedList {
    Node head, tail;
    int size;

    public MyLinkedList() {
        head = new Node(0); // head tail dummy nodes
        tail = new Node(0);
        head.next = tail;
        tail.prev = head;
    }

    public int get(int index) {
        if (index >= size || index < 0) return -1;

        Node t = head; // head is the dummy node so '-1'th index

    for (int i = 0; i < index; i++) t = t.next;  // move t to index-1-th position

        return t.next.val; // t.next is index because of head as dummy
    }

    public void addAtHead(int val) {
        Node next = head.next;
        Node n = new Node(val);
        head.next = n;
        n.next = next;
        if (size == 0) tail.prev = n;
        size++;
    }

    public void addAtTail(int val) {
        Node tailNode = tail.prev;
        Node n = new Node(val);
        tailNode.next = n;
        n.next = tail;
        tail.prev = n;
        size++;
    }

    public void addAtIndex(int index, int val) {
        if (index > size || index < 0) return; // now index can be size and it's okay :D


        Node t = head;

        for (int i = 0; i < index; i++) t = t.next;

        // now the new node needs to be placed between t and t.next
        Node next = t.next;
        Node n = new Node(val);
        t.next = n;
        n.next = next;

        if (size == index) tail.prev = n; // if index == size we should reassign tail

        size++;

    }

    public void deleteAtIndex(int index) {
        if (index >= size || index < 0) return;


        Node t = head;

        for (int i = 0; i < index; i++) t = t.next;

        // node t.next should be removed
        Node next = t.next.next;
        t.next = next;

        if (index == size-1) tail.prev = t; // if we removed the last node tail should be reassigned
        size--;
    }
```

```
    class Node {
        Node next,prev;
        int val;
        Node (int v){
            val = v;
        }
    }
}
```

Reversing a Doubly Linked List

```
void Reverse() {
  Node temp = null;
  Node curr = head;
  while(curr!=null) {
    temp = curr.next;
    curr.next = curr.prev;
    curr.prev = temp;
    curr = curr.prev;
  }
  if(temp!=null) {
    head=temp.prev;
  }
}
```

Singly Circular Linked List

Circular linked list *is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*

Advantages of Circular Linked Lists:

1. Any Node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation for queue.
3. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
4. Circular Doubly Linked Lists are used for implementation of advanced data structures like **Fibonacci Heap**.

Disadvantages of Circular LinkedList:

1. Depending on implementation, inserting at start of list would require doing a search for the last node which could be expensive.
2. Finding end of list and loop control is harder(no Null's to mark beginning and end)

Implementation:

```
package com.SinglyCircularLinkedList;

public class SinglyCircularLinkedList {

    public class Node {
        int data;
        Node next;

        Node() {
```

```
            data = 0;
            next = null;
        }
        Node(int x) {
            this.data = x;
            this.next = null;
        }
    }
    private Node head;
    private Node tail;
    private int size;

    public SinglyCircularLinkedList() {
        this.head = null;
        this.tail = null;
    }
    public boolean isEmpty(){
        return head == null;
    }

    public void addFirst(int data) {
        Node newNode = new Node(data);
        if(isEmpty()) {
            head = newNode;
            tail = newNode;
            newNode.next = head;
        } else {
            Node temp = head;
            newNode.next = temp;
            head = newNode;
            tail.next = head;
        }
        this.size++;
    }
    public void addAtIndex(int data, int index){
        if(isEmpty() || size < index-1) {
            throw new IllegalArgumentException();
        }
        else if(index == 0) {
            addFirst(data);
        }
        else if(size == index) {
            addAtEnd(data);
        }
        else {
            Node newNode = new Node(data);
            Node curr = head;
//            for(int i=1; i<index-1; i++) {
//                curr = curr.next;
//            }
            while(index-1 > 0) {
                curr = curr.next;
                index--;
            }
            newNode.next = curr.next;
            curr.next = newNode;
            size++;
        }
    }
    public void addAtEnd(int data) {
        Node newNode = new Node(data);
        if(isEmpty()) {
            head = newNode;
            tail = newNode;
            tail.next = head;
        } else {
            tail.next = newNode;
            tail = newNode;
            newNode.next = head;
        }
        this.size++;
    }

    public void deleteFirst() throws Exception {
        if(isEmpty()) {
            throw new Exception();
        } else {
            if(head != tail) {
                head = head.next;
                tail.next = head;
            } else {
```

```
                head = tail = null;
            }
        }
        this.size--;
    }

    public void deleteAtIndex(int index) throws Exception{
        if(isEmpty()) {
            throw new IndexOutOfBoundsException();
        }
        else if(size < index-1 || size <= index) throw new IllegalArgumentException();
        // use below else if, if you want to delete on circular manner
//        else if(size <= index) {
//            index = index%size;
//            deleteAtIndex(index);
//        }
        else if(index == 0) {
            deleteFirst();
        }
        else if(index+1 == size){
            deleteEnd();
        }
        // [1, 2, 3, 4]
        //     p
        else {
            Node curr = head;
            while(index-1 > 0) {
                curr = curr.next;
                index--;
            }
            Node temp = curr.next.next;
            curr.next = temp;
        }
        this.size--;
    }

    public void deleteEnd() throws Exception {
        if(isEmpty()) {
            throw new Exception();
        } else {
            if(head!=null) {
                Node curr = head;
                while(curr.next != tail) {
                    curr = curr.next;
                }
                tail = curr;
                tail.next = head;
            }
            else {
                head = tail = null;
            }
        }
        this.size--;
    }

    public int getAtIndex(int index) {
        if(isEmpty() || size < index-1) {
            throw new IllegalArgumentException();
        } else {
            Node curr = head;
            while(index>0) {
                curr = curr.next;
                index--;
            }
            return curr.data;
        }
    }

    public void display() {
        Node curr = head;
        if(head!=null) {
            do {
                System.out.printf("%d ", curr.data);
                curr = curr.next;
            } while(curr != head);
        }
        System.out.printf("\n");
    }

}
```

Stacks

Stack is a linear data structure which follows a particular order in which the operation are performed. The order may be LIFO(Last In First Out).

Application of Stacks:

1. Implement the undo feature
2. Build compilers(eg syntax checking)
3. Evaluate expressions (eg 1+2*3)
4. Build navigation (eg forward/back)

Operations: (all in O(1) Runtime)

1. push(item)
2. pop()
3. peek()
4. isEmpty()

Stack Implementation Using Arrays

```
package com.Stack;

import java.util.Arrays;

public class StackImplement {
    private int count = 0;
    private int[] array = new int[5];

    public void push(int item) {

        if(count == array.length) {
            throw new StackOverflowError();
        }

        array[count++] = item;
    }

    public int pop() {
        if(count == 0) {
            throw new IllegalStateException();
        }
        return array[--count];
    }

    public int peek() {
        if(count == 0) {
            throw new IllegalStateException();
        }

        return array[count-1];
    }

    public boolean isEmpty() {
        return count == 0;
    }
    @Override
    public String toString() {
        var content = Arrays.copyOfRange(array, 0, count);
        return Arrays.toString(content);
```

```
    }
}
```

Stack Implementation using LinkedList

```
package com.Stack;

import java.util.ArrayList;
import java.util.List;

public class StackListImplementation<T> {

    private List<T> stack;
    public StackListImplementation() {
        this.stack = new ArrayList<T>();
    }

    public boolean isEmpty() {
        return stack.size()==0;
    }

    public void push(T obj) {
        stack.add(obj);
    }

    public T pop() {
        if(stack.size() == 0) {
            throw new IllegalStateException();
        }
        return stack.remove(stack.size()-1);
    }

    public T peek() {
        if(stack.size() == 0) {
            throw new IllegalStateException();
        }
        return stack.get(stack.size()-1);
    }

    @Override
    public String toString() {
        StringBuilder str = new StringBuilder();
        for(T obj: stack) {
            str.append(obj);
            str.append(", ");
        }
        return str.toString();
    }
}
```

Stack Implementation using Queues

A stack can be implemented using two queues. Let stack to be implemented be ‘s’ and queues used to implement be ‘q1’ and ‘q2’. Stack ‘s’ can be implemented in two ways:**Method 1 (By making push operation costly)** This method makes sure that newly entered element is always at the front of ‘q1’, so that pop operation just dequeues from ‘q1’. ‘q2’ is used to put every new element at front of ‘q1’.

1. **push(s, x)** operation’s step are described below:
    - Enqueue x to q2
    - One by one dequeue everything from q1 and enqueue to q2.
    - Swap the names of q1 and q2
2. **pop(s)** operation’s function are described below:

- Dequeue an item from q1 and return it.

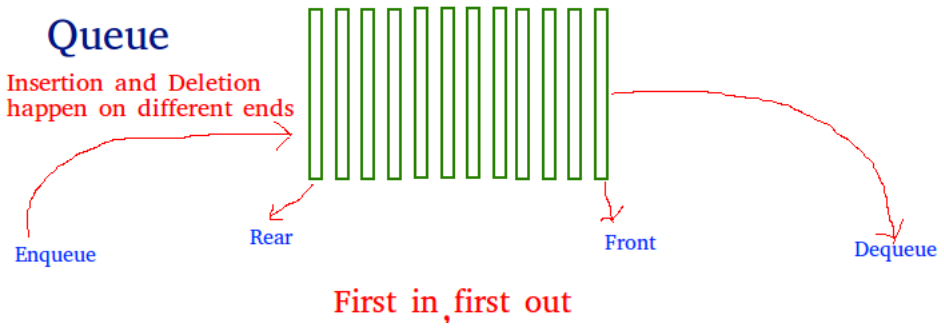
Below is the implementation of the above approa

```
package com.Stack;

import java.util.LinkedList;
import java.util.Queue;

public class StackQueueImplementation {

    Queue<Integer> q1 = new LinkedList<>();
    Queue<Integer> q2 = new LinkedList<>();
    int curr_size;

    public StackQueueImplementation() {
        curr_size = 0;
    }

    public void push(int x) {
        curr_size++;
        q2.add(x);

        while(!q1.isEmpty()) {
            q2.add(q1.peek());
            q1.remove();
        }

        Queue<Integer> q = q1;
        q1 = q2;
        q2 = q;
    }

    public void pop() {
        if(q1.isEmpty()) {
            return;
        }

        q1.remove();
        curr_size--;
    }

    public int top() {
        if(q1.isEmpty()) {
            return -1;
        }
        return q1.peek();
    }

    public int size() {
        return curr_size;
    }

}
```

QUEUES

What are Queues ?

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Application of Queues:

- Printers
- Operation Systems
- Web Servers
- Live Support System

Operations Performs Using a Queue:

operations of Queues

Operations	Time Complexity
Enqueue	O(1)
Dequeue	O(1)
peek	O(1)
isEmpty	O(1)
isFull	O(1)

Queues in Java:

In java Queue is implemented using Interface, so we can't instantiate it. for implementing the Queue we have to use the implementing classes, like, ArrayDequeue, LinkedList, PriorityQueue, etc.

You can find more about queue in the java documentation of Queue.

```
package com.Queue;
import java.util.ArrayDeque;
import java.util.Queue;

public class Main {

    public static void main(String[] args) {
        Queue<Integer> queue = new ArrayDeque<>();
        queue.add(10);
        queue.add(20);
        queue.add(30);
        var front = queue.poll();
        System.out.println(queue);
        System.out.println(front);
```

```
    }
}
```

Reversing the Queue:

In this method, I have implemented the reversal of queue using Stack. In this approach a stack is used to reverse the the queue. For every item removed from the front of the Queue, pushed to stack and when the queue gets empty and stack got all the elements of the queue then in the last, stack will have the last of element of the queue as a top and when we pop item from the stack and add it into the queue, then, it gets reversed.

```
package com.Queue;
import java.util.ArrayDeque;
import java.util.Queue;
import java.util.Stack;

public class Main {

    public static void main(String[] args) {
        Queue<Integer> queue = new ArrayDeque<>();
        queue.add(10);
        queue.add(20);
        queue.add(30);
        var front = queue.poll();
        System.out.println(queue);
        System.out.println(front);
        reverse(queue);
        System.out.println(queue);


    public static void reverse(Queue<Integer> queue) {

        if(queue.isEmpty()) {
            System.out.println("Empty Queue");

        }
        Stack<Integer> temp1 = new Stack<>();
        while(!(queue.isEmpty())) {
            var front = queue.remove();
            temp1.add(front);
        }
        while(!(temp1.isEmpty())) {
            var top = temp1.pop();
            queue.add(top);
        }

    }
}
```

Implementation of Queue:

Queue can be implemented using 3 different ways:

1. Using Array
2. Using Linked List
3. Using Stack

1. Using Array

In this topic, I have implemented Queue using the array which is also the from scratch implementation of a Queue. For this I have created one class name ArrayQueue.java for which code is given below. In this, I have implemented different queue methods like enqueue(), dequeue(), peek(), isEmpty(), isFull().

```
package com.Queue;

import java.util.Arrays;

public class ArrayQueue {
    private int[] items;
    private int rear;
    private int front;
    private int count;
    public ArrayQueue(int capacity) {
        items = new int[capacity];
    }

    public void enqueue(int item) {
        if(count == items.length) {
            throw new IllegalStateException();
        }
        items[rear++] = item;
        count++;
    }

    public int dequeue() {
        var item = items[front];
        items[front++] = 0;
        count--;
        return item;
    }

    public int peek() {
        if(count == 0) {
            return 0;
        }
        return items[front];
    }

    public boolean isEmpty() {
        return count == 0;
    }

    public boolean isFull() {
        return count == items.length;
    }

    @Override
    public String toString() {
        return Arrays.toString((items));
    }
}
```

But this above implementation has one problem, when we remove an item from the front of the queue then that space becomes empty, but while adding the item at the rear of the queue the empty space will not be used but the element is added in the end of the array. For example, here is an queue[10, 20, 30, 0, 0] and now when we remove the front item which is in this case is 10 then the queue will look like this [0, 20, 30, 0, 0] now let's suppose if we add 40 in the queue and then 50, then queue will look like this [0, 20, 30, 40, 50] but now when we add another item suppose 60 in the queue, it will give the error named ArrayIndexOutofBoundError, as we see even we have one space left in our array(which is used to implement the queue) it is giving this error. so to resolve this error now we can make an "Circular Queue" through which we can add the item in the empty space of the queue without getting any error.

To implement circular queue for the above problem, we have to come up with an approach which can solve this problem, you can understand the approach give below,

```
// suppose there is an array of
// R -> Rear, F-> Front
// [0, 0, 30, 40, 50]
//                R
// next time we want to add an item we will increment R by 1
// now, 5 -> 0 (Left % Length)
// 6 -> 1
// 7 -> 2
// 8 -> 3
// 9 -> 4
// 10 -> 0
// (rear + 1) % length of the array

// so now the new item will go like
// [60, 70, 30, 40, 50]
//       R   F
```

now in java it can be written like this,

```
package com.Queue;

import java.util.Arrays;

public class ArrayQueue {
    private int[] items;
    private int rear;
    private int front;
    private int count;
    public ArrayQueue(int capacity) {
        items = new int[capacity];
    }

    public void enqueue(int item) {
        if(isFull()) {
            throw new IllegalStateException();
        }
        items[rear] = item;
        rear = (rear + 1) % items.length;
        count++;
    }

    public int dequeue() {
        var item = items[front];
        items[front] = 0;
        front = (front + 1) % items.length;
        count--;
        return item;
    }

    public int peek() {
        if(isEmpty()) {
            return 0;
        }
        return items[front];
    }

    public boolean isEmpty() {
        return count == 0;
    }

    public boolean isFull() {
        return count == items.length;
    }

    @Override
    public String toString() {
        return Arrays.toString((items));
    }
}
```

2. Using Stack

In this topic, I have implemented Queue using the Stack which is also the from scratch implementation of a Queue. For this I have created one class name StackQueue.java for which code is given below. In this, I have implemented different queue methods like enqueue(), dequeue(), peek(), isEmpty().

```
package com.Queue;
import java.util.Arrays;
import java.util.Stack;

public class StackQueue {

    private Stack<Integer> s1 = new Stack<>();
    private Stack<Integer> s2 = new Stack<>();


    public void enqueue(int item) {
     s1.push(item);

    }
    public int dequeue() {

        if(isEmpty()) {
            throw new IllegalStateException();
        }
        moveStack1ToStack2();

        return s2.pop();
    }

    private void moveStack1ToStack2() {
        if (s2.isEmpty()) {
            while (!s1.isEmpty()) {
                s2.push(s1.pop());
            }
        }
    }

    public boolean isEmpty() {
        return s1.isEmpty() && s2.isEmpty();
    }

    public int peek() {

        if(isEmpty()) {
            throw new IllegalStateException();
        }
        moveStack1ToStack2();

        return s2.peek();
    }

    @Override
    public String toString() {
        return "StackQueue{" +
                "s1=" + s1 +
                ", s2=" + s2 +
                '}';
    }
}
```

3. Using Linked List

In this topic, I have implemented Queue using the Linked List which is also the from scratch implementation of a Queue. For this I have created one class name LinkedListQueue.java for which code is given below. In this, I have implemented different queue methods like enqueue(), dequeue().

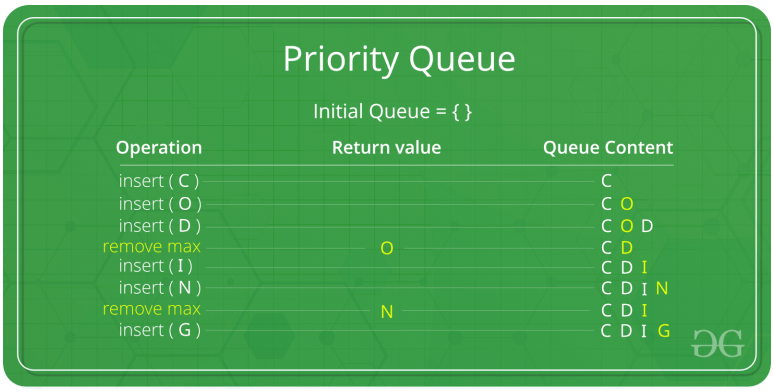
```
package com.Queue;
```

```
class QNode {
    int key;
    QNode next;

    public QNode(int item) {
        this.key = item;
        this.next = null;
    }
}

public class LinkedListQueue {

    QNode front, rear;

    public LinkedListQueue() {
        this.front = this.rear = null;
    }

    public void enqueue(int item) {

        QNode temp = new QNode(item);

        if(this.rear == null) {
            this.front = this.rear = temp;
            return;
        }

        this.rear.next = temp;
        this.rear = temp;
    }

    public void dequeue() {
        if(this.front == null) {
            throw new IllegalStateException();
        }
        QNode temp = this.front;
        this.front = this.front.next;

        if(this.front == null) {
            this.rear = null;
        }
    }

    @Override
    public String toString() {
        return "LinkedListQueue{" +
                "front=" + front +
                ", rear=" + rear +
                '}';
    }
}
```

Priority Queue:

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.



Priority Queue in Java:

```
public class Main{
  public static void main(String[] args) {
    PriorityQueue<E> queue = new PriorityQueue<>();
  }
}
```

Implementation of Priority Queue using Array:

Suppose we have to insert x in the sorted manner in the array then we have to think like this:

[1, 3, 5, 7] → let this be an array, now add value 2 in this array in sorted manner.

To do this we have to start comparing from the back and check if the current element is greater than the element we are inserting, if this condition is true then copy the current item to the right

→ [1, 3, 5, 7]

→ [1, 3, 5, 7, 7]

→ [1, 3, 5, 5, 7]

→ [1, 3, 3, 5, 7]

→ [1, 2, 3, 5, 7]

Implementation:

```
package com.Queue;

import java.util.Arrays;

public class PriorityQueueArray {
    private int[] items = new int[5];
    private int count;

    public void add(int val) {
        if(count == items.length) {
            int[] temp = new int[2*count];
            for(int i=0; i<count; i++)
                temp[i] = items[i];
            items = temp;
        }
        // shifting items
        int i;
        for(i=count-1; i>=0; i--){
            if(items[i] >= val) {
                items[i+1] = items[i];
            }
            else
                break;
        }
```

```
        items[i+1] = val;
        count++;
    }

    public int remove() {
        if (isEmpty()) {
            throw new IllegalStateException();
        }
        return items[--count];
    }

    public boolean isEmpty() {
        return count==0;
    }
    @Override
    public String toString() {
        return Arrays.toString((items));
    }

}
```

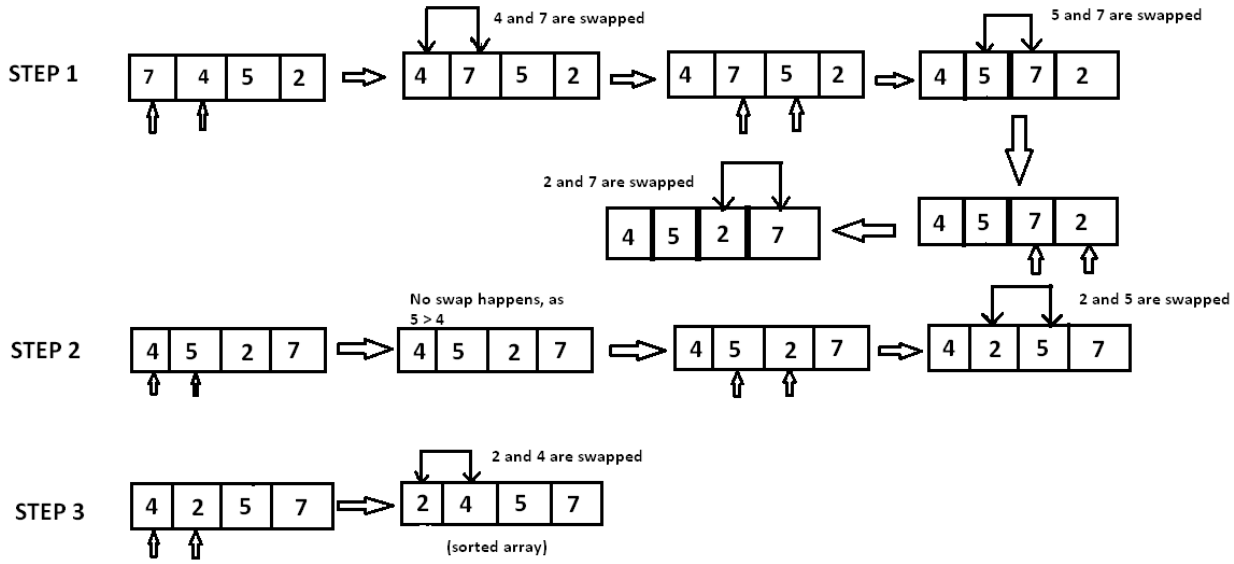
Time complexity of Priority Queue for insertion in O(1) as just it add items in the end of the array, for getting the highest priority in it, it took O(n) time as it have to linearly search for the item.

Sorting Algorithms:

1. Bubble Sort:

Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

For an unsorted array A[] of n elements, the array needs to be sorted in ascending order.



The Time complexity of bubble sort is O(n^2) in both worst and average cases.

```
package com.Sorting;

public class BubbleSort {
    public void sort(int[] array) {
        boolean isSorted;
        for(var i=0; i<array.length; i++) {
```

```
            isSorted = true;
            for (int j = 1; j < array.length - i; j++) {
                if (array[j] < array[j - 1]) {
                    swap(array, j, j - 1);
                    isSorted = false;
                }
            }
            if(isSorted) {
                return;
            }
        }
    }

    public void swap(int[] arr, int index1, int index2) {
        int temp = arr[index1];
        arr[index1] = arr[index2];
        arr[index2] = temp;
    }
}
```

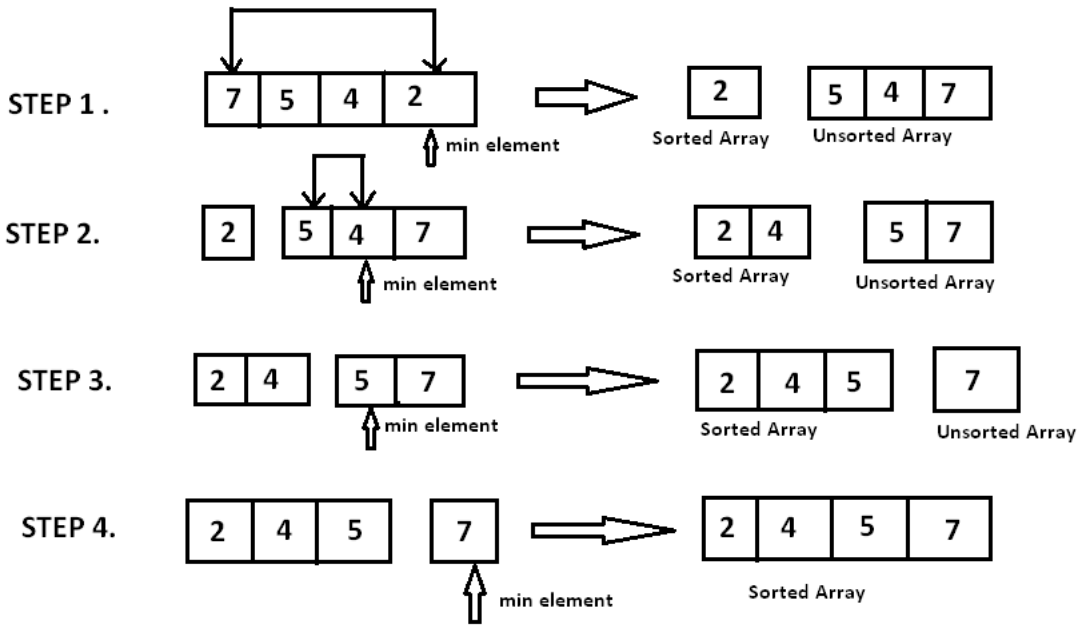
2. Selection Sort

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

Assume that the array A=[7,5,4,2] needs to be sorted in ascending order.

The minimum element in the array i.e. 2 is searched for and then swapped with the element that is currently located at the first position, i.e. 7. Now the minimum element in the remaining unsorted array is searched for and put in the second position, and so on.

Let's take a look at the implementation.




```
void selectionSort(int a[], n) {
  Scanner s = new Scanner(System.in);
          int n = s.nextInt();

          int[] a = new int[n];
          for(int i=0; i<n; i++) {
              a[i] = s.nextInt();
          }
```

```
        for(int i=0; i<n; i++) {
            int min = i;
            for(int j=i+1; j<n; j++) {
                if(a[j] < a[min]) {
                    min = j;
                }
            }
            int temp = a[i];
            a[i] = a[min];
            a[min] = temp;
        }

        for(int i=0; i<n; i++) {
            System.out.print(a[i] + " ");
        }
    }
```

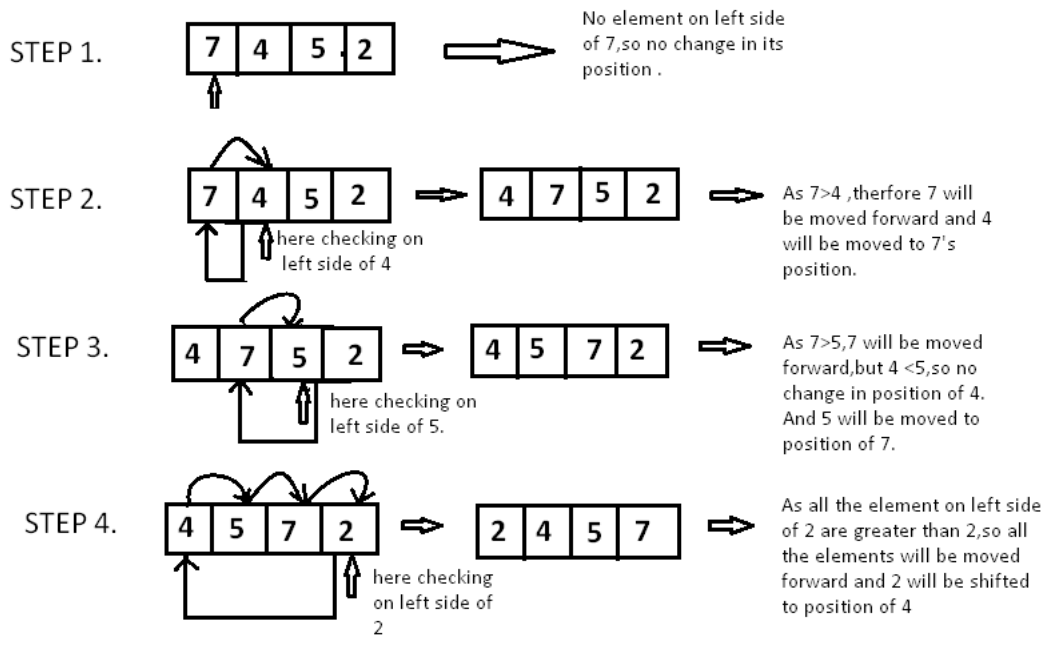
Time Complexity of Selection Sort is O(n^2) Quadratic.

After putting the minimum element in its proper position, the size of an unsorted array reduces to N−1 and then N−2 comparisons are required to find the minimum in the unsorted array.

3. Insertion Sort

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead.




```
package com.Sorting;

public class InsertionSort {

    public void sort(int[] array) {
        for(var i=1; i<array.length; i++) {
```

```
            var current = array[i];
            var j = i-1;
            while(j >= 0 && array[j] > current) {
                array[j+1] = array[j];
                j--;
            }
            array[j+1] = current;
        }
    }
}
```

Time complexity of insertion sort is O(n^2) however it best time complexity is O(n) as we don't have to perform any shifts if the array is already sorted.

4. Merge Sort

It is a Divide and conquer based algorithm which firsts divide the array into two halves until there is only one element left in the array, then merge the array by comparing which makes it a sorted array.

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Idea:

- Divide the unsorted list into sublists, each containing element.

  N

  1

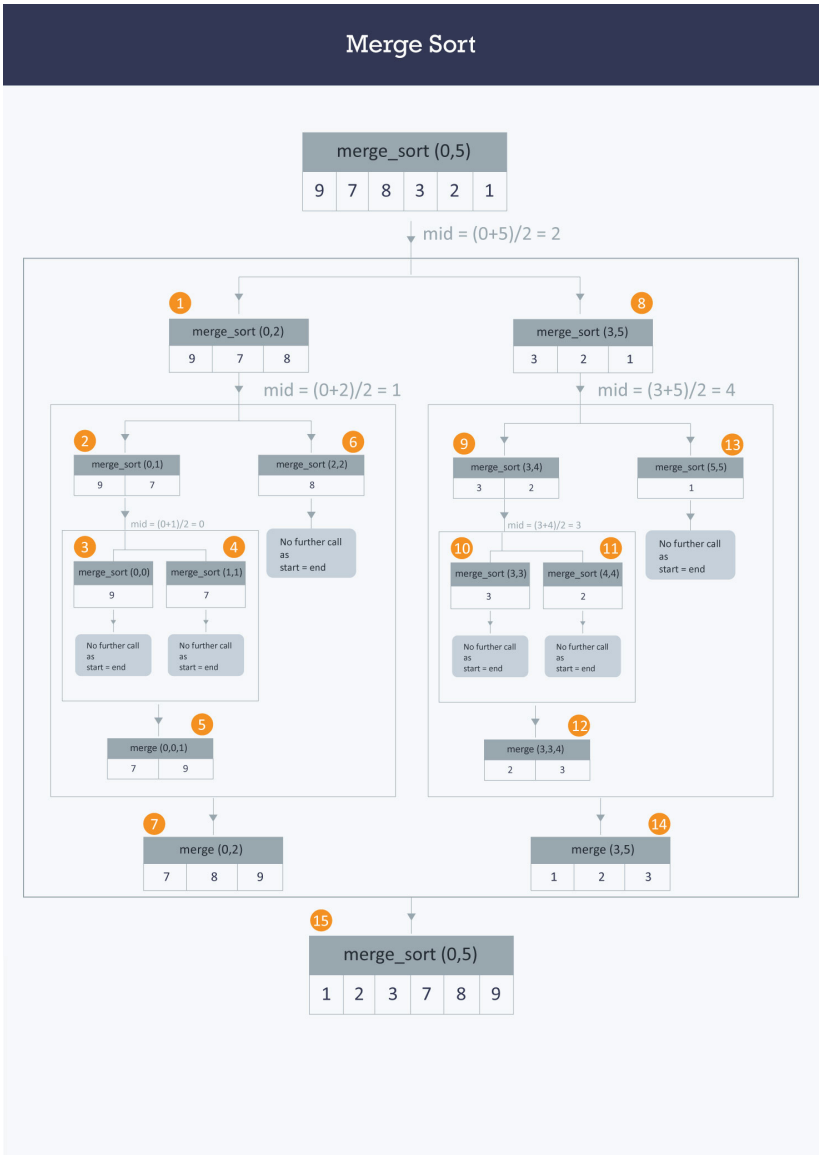
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. will now convert into lists of size 2.

  N

  N/2

- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.



Implementation:

```
package com.Sorting;

public class MergeSort {

    public void sort(int[] array) {
        if(array.length < 2) {
            return;
        }

        int mid = array.length/2;
        int[] left = new int[mid];
        for(int i=0; i<mid; i++) {
            left[i] = array[i];
        }

        int[] right = new int[array.length-mid];
        for(int i=mid; i<array.length; i++) {
            right[i-mid] = array[i];
        }

        sort(left);
```

```
        sort(right);
        merge(left, right, array);

    }

    private void merge(int[] left, int[] right, int[] result) {
         int i=0, j=0, k=0;
         while(i<left.length && j<right.length) {
             if(left[i] <= right[j])
                 result[k++] = left[i++];
             else
                 result[k++] = right[j++];
         }

         while(i < left.length) {
             result[k++] = left[i++];
         }
         while(j < right.length) {
             result[k++] = right[j++];
         }
    }
}
```

The time complexity of merge sort is based on two conditons which is first dividing the array and then merging the array. So for dividing the array it will take log(n) as it is dividing it into the half, every time, then for merging it will

5. QuickSort

It is also a Divide and conquer algorithm like Merge Sort but more efficient and compatible than it. It picks an element as pivot and partition the given array around the chosen pivot. There can be different ways to pick the pivot like, always

1. Pick first element as a pivot
2. pick last element as a pivot
3. pick random element as a pivot
4. pick middle element as a pivot

The important or the key process in quicksort in the partition. Given an array and an element x of array as pivot, put x in its correct position in sorted array and put all the elements greater than the x, after the x and smaller elements before the x and this all should happen in linear time.

Below is the implementation of Quick Sort(picking last element as a pivot)

```
package com.Sorting;

public class QuickSort {

    public void sort(int[] array, int low, int high) {
        if(low >= high)
            return;

        // partition
        int pi = partition(array, low, high);
        // left sort
        sort(array, low, pi-1);
        // right sort
        sort(array, pi+1, high);

    }

    private int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int boundary = low-1;

        for(int i=low; i<=high; i++) {
            if(array[i] <= pivot){
                swap(array, i, ++boundary);
            }
```

```
        }
        return boundary;
    }

    private void swap(int[] arr, int index1, int index2) {
        int temp = arr[index1];
        arr[index1] = arr[index2];
        arr[index2] = temp;
    }
}
```

Analysis of QuickSort Time taken by QuickSort, in general, can be written as following.

```
 T(n) = T(k) + T(n-k-1) +Theta(n)
```

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot. The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

```
 T(n) = T(0) + T(n-1) +theta(n)
which is equivalent to
T(n) = T(n-1) +theta(n)
```

The solution of above recurrence is O(n^2)

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

```
 T(n) = 2T(n/2) +theta(n)
```

The solution of above recurrence is

(nLogn). It can be solved using case 2 of

Master Theorem

.

Average Case: To do average case analysis, we need to **consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy**. We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

```
 T(n) = T(n/9) + T(9n/10) +(n)
```

Solution of above recurrence is also O(nLogn)Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like **Merge Sort** and **Heap Sort**, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

Is QuickSort stable? The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

Is QuickSort In-place? As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.

What is 3-Way QuickSort? In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot. Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts: a) arr[l..i] elements less than pivot. b) arr[i+1..j-1] elements equal to pivot. c) arr[j..r] elements greater than pivot. See **this** for implementation.

How to implement QuickSort for Linked Lists? QuickSort on Singly Linked List QuickSort on Doubly Linked List

Can we implement QuickSort Iteratively? Yes, please refer **Iterative Quick Sort**.

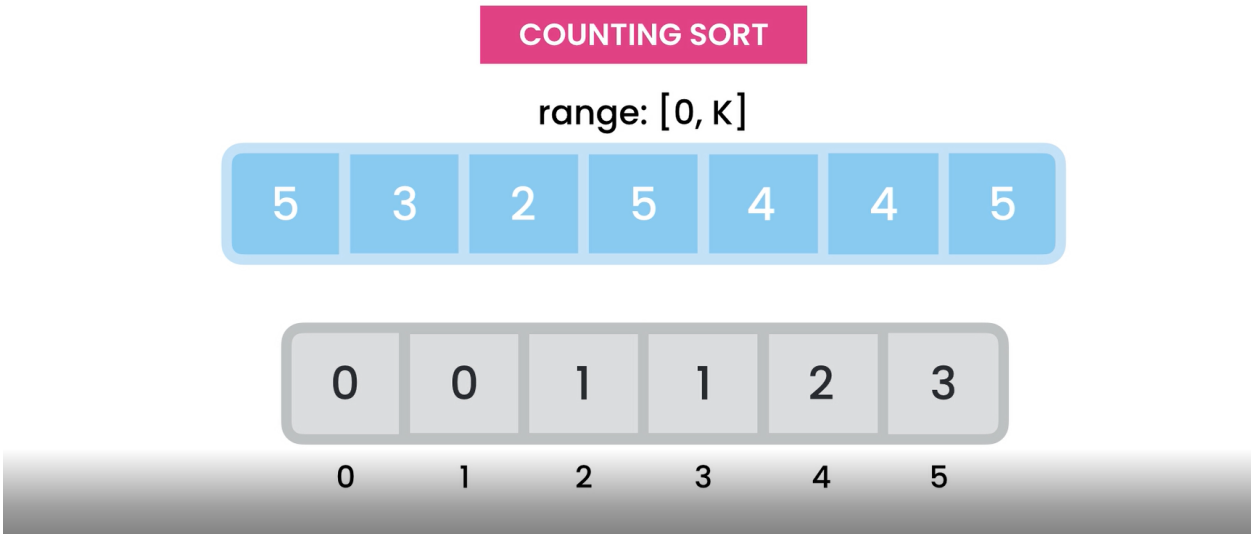
Why Quick Sort is preferred over MergeSort for sorting Arrays Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space. Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice. Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays. Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why MergeSort is preferred over QuickSort for Linked Lists? In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists. In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

6. Counting Sort

Counting sort is a sorting technique based non-comparison method, in which according to the maximum element in the input array creates another array named counting array which is used to store the number of times a single element in the input array appeared. Then replacing the elements in the input array according to the elements occurred in the counting array, in sorted manner.

-



When to use counting sort:

1. Allocating extra space is not an issue
2. values are positive integers
3. Most of the values in the range are present

Points to be noted:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
2. It is not a comparison based sorting. It running time complexity is O(n) with space proportional to the range of data.
3. It is often used as a sub-routine to another sorting algorithm like radix sort.
4. Counting sort uses a partial hashing to count the occurrence of the data object in O(1).
5. Counting sort can be extended to work for negative inputs also.

Implementation:

```
package com.Sorting;
// non-comparision based sorting algorithm
public class CountingSort {
    public void sort(int[] array, int max) {
        int[] counts = new int[max+1];
        int x = 0;

        for(var item: array)
            counts[item]++;

        for(int i=0; i<counts.length; i++)
            if(counts[i] != 0)
                for(int j=0; j<counts[i]; j++)
                    array[x++] = i;
    }
}
```

Time Complexity:

The input array is traversed in O(n) time and the resulting sorted array is also computed in O(n) time. Counting array is traversed in O(k) time. Therefore, the overall time complexity of counting sort algorithm is O(n + k).

7. Bucket Sort

It is another non-comparison based algorithm, and is mainly useful when input is uniformly distributed over a range. For example, Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range.

Implementation:

```
package com.Sorting;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class BucketSort {

    public void sort(float[] array, int numOfBuckets) {
        int i=0;
        for(var bucket: createBuckets(array, numOfBuckets)) {
            Collections.sort(bucket);
            for (var item : bucket)
                array[i++] = item;
        }
    }

    private List<List<Float>> createBuckets(float[] array, int numOfBuckets) {
        List<List<Float>> buckets = new ArrayList<>();
        for(var i=0; i<numOfBuckets; i++)
```

```
            buckets.add(new ArrayList<>());

        for(float item: array) {
            var i = (int)(item / numOfBuckets) / numOfBuckets;
            buckets.get(i).add(item);
        }
        return buckets;
    }
}

public class Main {
      public static void main(String[] args) {
        BucketSort sorter = new BucketSort();
//        int[] arr = {2, 5, 8, 6, 7, 5, 10, 5};
        float[] arr = {(float)0.897, (float)0.565,
                (float)0.656, (float)0.1234,
                (float)0.665, (float)0.3434};
        sorter.sort(arr, 3);
        System.out.println(Arrays.toString(arr));
    }
}
```

Time Complexity:

Distribution in bucket sort will take O(n) time as there are n no. of items in the array. Iterating over the buckets will take O(k) time as there is k no. of buckets and for sorting it will take O(1) for best case and O(n^2) for the worst case scenario. So the overall time complexity of bucket sort is O(n+k) for best case and O(n^2) for the worst case.

Space Complexity:

It's space complexity is O(n+k) as it will only have k no. of buckets and n no. of items in the array.

Hash Tables

What are Hash Tables?

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. They are also know for a key value pair, like for every there would be some unique value assigned to it, because of which look up becomes which easier.

Application of Hash Tables:

- Spell Checkers
- Dictionaries
- Compilers
- Code editor

> Note: In Hash Map we cannot have duplicate keys

Operations:

- Insert O(1)
- Lookup O(1)
- Delete O(1)

Java Implementation:

```
public static void main(String[] args) {
  Map<Integer, String> map = new HashMap<>();
  map.put(1, "Sanskar");
  map.put(2, "Aman");
  map.put(3, "Kuldeep");

  map.contains(3); // O(1)
// O(n) because in this case hashmap cannot rely in its objects, it have to iterate over object to find this value
  map.containsValue("Sanskar");
  for(var item: map.keySet())
    System.out.println(item);
}
```

Sets

- HashSet class implements the Set Interface, backed by Hash table which is actually a HashMap. There is no key value pair in this.
- There is no constant order of iteration
- This class permits the null element
- As it implements the set interface, duplicate values are not allowed.

Operations:

- add
- remove
- contains
- size

Java Implementation:

```
public static void main(String[] args) {
  Set<E> set = new HashSet<>();
}
```

Interview Question: First Repeated Character

Return the first repeated character from the sentence. Ignore the letter casing for now.

Input: green apple

Output: e

```
public static char firstRepeatedChar(String str) {
    Set<Character> set = new HashSet<>();
    for(var char: str.toCharArray()) {
      if(set.contains(ch))
        return ch;
      else
        set.add(ch);
    }
    return Character.MIN_VALUE // in case there is no repeated character
```

Hash Functions:

Hash Function is a function that gets a value and maps it to a different kind of value which we call a Hash Value, a Hash Code or Hash. A hash function maps a key value to a index value.

Dummy Representation of Hash Function in Java:

```
public static void main(String[] args) {
  Map<String, String> map = new HashMap<>();
  map.put("1234567-S", "John");
  // here hash function is used find the index value to store this string value;
  System.out.println(hash("1234567-S"));
}
public static int hash(String key) {
  int hash=0;
  for(var ch: key.toCharArray()) {
  /* Even though we are working with characters here, because we have declared hash as an
  integer, so when we apply agumented assignment operator with a character, this will
  automatically will be converted into an integer, this is known as Implicit Casting.
  */
    hash += ch;
  }
  return hash%100; // here we are assuming that are array size is 100.
}
```

Collisions:

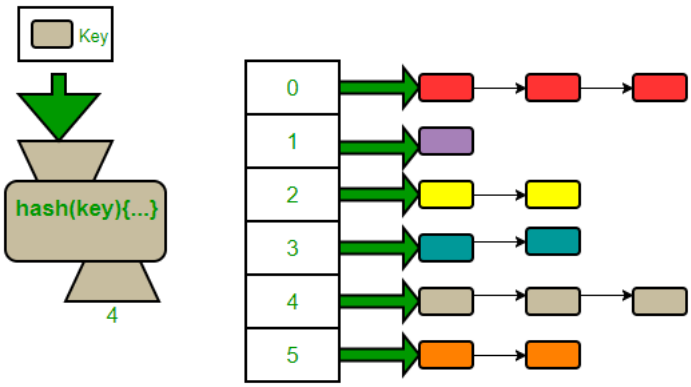
Collisions happens when two different keys are assigned for the same index by the hash function.

There are two ways to handle collisions:

1. Using Linked-List (By Chaining)
2. Another solution is to find a different slot for storing the second value, this is called open addressing.

Chaining:

Assume we have an array of size 5, and we want to store the values, initially all our buckets(slots) are null or empty. If two different key gets assigned to the same index then that value will be added to the end of the linked list.

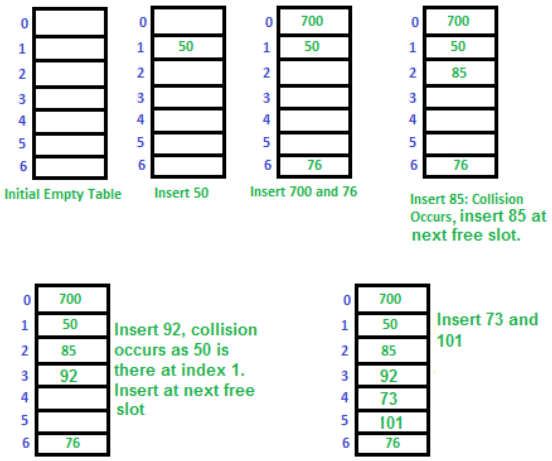


Open Addressing:

There are 3 type of Open Addressing approach:

1. **Linear Probing**: With this approach we don't store values in the linked list but directly in array cells or slots. Formula to find the index in Linear Probing is:

> Linear Probing : (hash(key) + i) % table_size



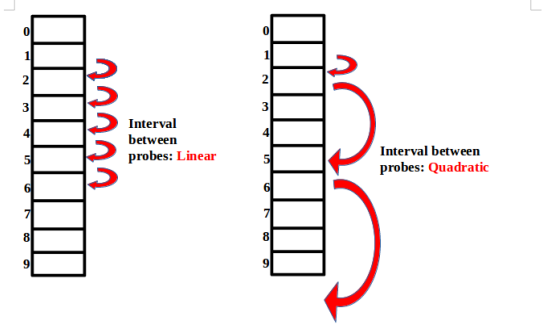
here i denotes the loop variable. When collisions happens in this case then, the colliding value traverse the array and find the empty slot, for value to get stored. This is called Probing which means searching. It is called open addressing because the address of the key value pair does not determined by the hash function.

One drawback of this is approach is, if the array is full then their is no space to store the items.

2. **Quadratic Probing:** To solve the problem of forming clusters(it means every time we have to add item then we have to traverse through filled slots) in linear probing we use Quadratic Probing. Through this approach we will spread the values out, more widely in the array. Formula to find the index in Quadratic Probing is:

> Linear Probing : (hash(key) + i^2) % table_size

Now here i's value is squared so the values will spread more widely.



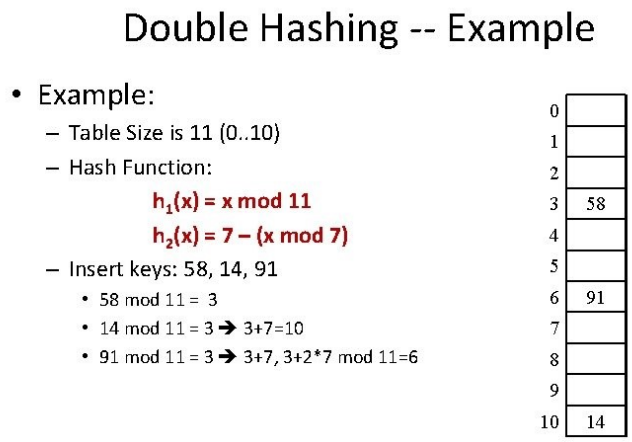
Now, this Quadratic Probing approach also has one problem. As we make big jumps because of i^2, due to which we can again come back to beginning of the array and end up doing the same traversing which can end up into infinite loop. To solve this problem we use Double Hashing.

3. **Double Hashing:** With this approach, instead of i or i^2 there is separate independent hash function to calculate the number of steps.

> hash2(key) = prime - (key % prime)

Prime should be a prime number less than a size of table.

> index = (hash1(key) + i*hash2(key)) % table_size



Interview Question: Build a Hash Table

Build a hash function using chaining approach. Operation to be implemented:

1. put(k, v)
2. get(k): v
3. remove(k)
4. key: integer
5. value: string
6. collisions: chaining

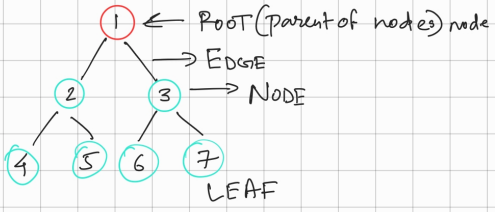
```
// code for hashing will come here
package com.Hashing;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class HashTable {

    private class Entry {
        private int key;
        private String value;

        public Entry(int key, String value) {
            this.key = key;
            this.value = value;
        }

    }

    private LinkedList<Entry>[] entries;

    public HashTable(int size) {
        this.entries = new LinkedList[size];
    }

    public void put(int key, String value) {
```

```
        var entry = getEntry(key);
        if(entry!=null) {
            entry.value = value;
            return;
        }
        var bucket = getOrCreateBucket(key);
        bucket.addLast(new Entry(key, value ));
    }


    public String get(int key) {
        var entry = getEntry(key);
        return(entry==null) ? null : entry.value;
    }


    public void remove(int key) {
        var entry = getEntry(key);
        if(entry == null)
            throw new IllegalStateException();
        getBucket(key).remove(entry);
    }

    private int hash(int key){
        return key % entries.length;
    }

    private LinkedList<Entry> getBucket(int key) {
        return entries[hash(key)];
    }

    private LinkedList<Entry> getOrCreateBucket(int key) {
        var index = hash(key);
        if(entries[index] == null) {
            entries[index] = new LinkedList<>();
        }
        var bucket = entries[index];
        return bucket;
    }

    private Entry getEntry(int key) {
        var bucket = getBucket(key);
        if(bucket != null) {
            for(var entry: bucket) {
                if(entry.key == key) {
                    return entry;
                }
            }
        }
        return null;
    }
}
```

Binary Tree

What are Trees?

A tree is a non-linear Data Structure that stores elements in the form of hierarchy, that consists of nodes connected by edges.



This is a special type of tree called Binary Tree, which Consists of only two child nodes.

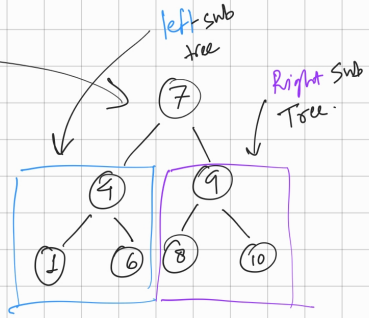
Application of Trees

- Represent hierarchical data
- Databases
- Autocompletion.
- Compilers
- Compression (JPEG, MP3)

Binary Search Tree:

left < node < right

This means in Binary Search Tree every left node is less than the parent node and every right node is greater than the parent node.



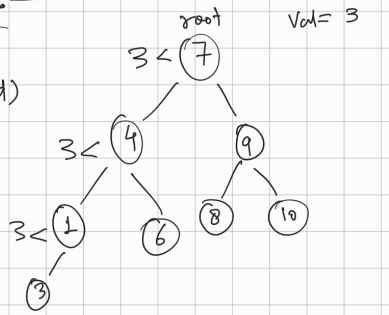
Note:- Whenever we throw out half our items and narrow down our search we have Logarithmic Time Complexity

Operations in Binary Search Tree:

1) Lookup: O(LogN)
2) Inserting: O(LogN)
3) Deletion: O(LogN)

Building a Binary Search Tree :—

1. Tree (root)
2. Node (value, leftchild, rightchild)
3. insert (value)
4. find (value)




```
insert(int val){
 Var node = new Node(val);
 if(root == null) {
    root = node;
    return;
 }
 Var curr = root;
 while(true {
    if(val > curr.val){
        if( curr.rightchild == null) {
            curr.rightchild = node;
            break;
        }
        curr = curr.rightchild;
    }
    else{
```

```
find(val) {
 if(root != null){
    var curr = root;
    while (curr != null) {
      if(curr.val == val)
          return true;
      if (val > curr.value)
          curr = curr.rightCh
```

```
            if (curr.leftchild == null) {
                curr.leftchild = node;
                break;
            }
            curr = curr.leftchild;
        }
    }
}
```

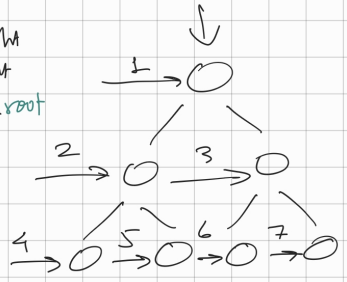
```
else
    curr = curr.leftchild;
}
}
return False;
}
```

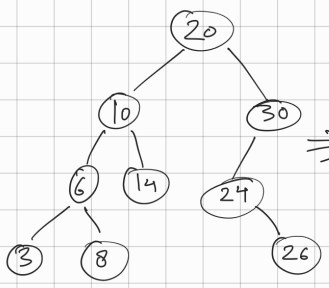
TREE TRAVERSAL:

There are two type of Tree Traversal

(1) BFS (Breath First Traversal) → This is also called level order traversal, means first visit all the node at the same level before visiting the node at another level

(2) DFS (Depth First Traversal)
- Preorder → Root, left, right
- Inorder → left, root, right
- PostOrder → left, right, root





Preorder ⇒ 20, 10, 6, 3, 8, 14, 30, 24, 26

Inorder ⇒ 3, 6, 8, 10, 14, 20, 24, 26, 30

Postorder ⇒ 3, 8, 6, 14, 10, 26, 24, 30, 20

BFS (Level) ⇒ 20, 10, 30, 6, 14, 24, 3, 8, 26

Recursion :- ← Part for Traversing the trees.

Function calling itself. Expor ° a problem into smaller problem.

For eg. factorial of 4 = 4x3!

We can generalise it as n = n x (n-1)!

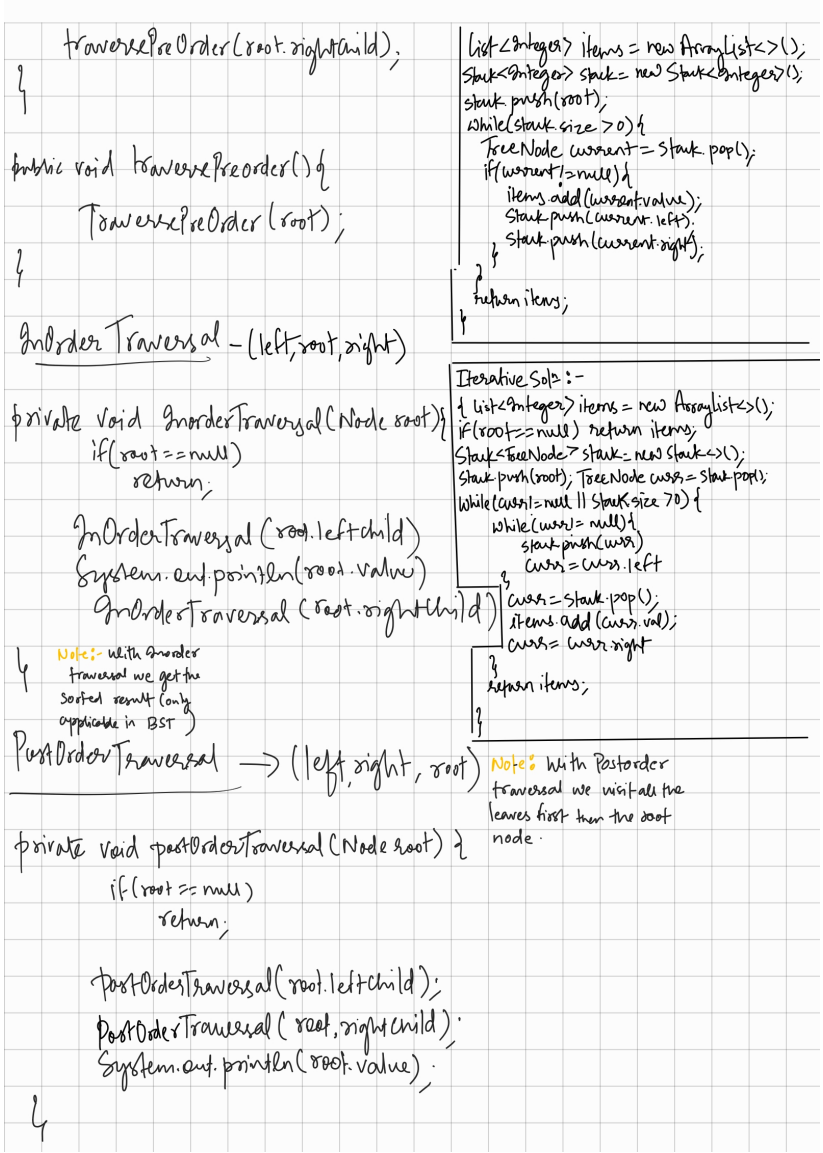
```
factorial(int n) {
    if (n==0) // base condition.
      return 1;
    return n * factorial(n-1);
}
```

Preorder Traversal: — (root, left, right) ← This is the implementation detail.

```
private void traversePreOrder(Node root) {
    if(root==null)
        return.
    System.out.println(root.value);
    traversePreOrder(root.leftChild);
```

Iterative Soln

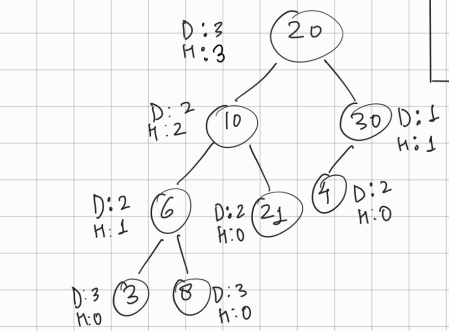
```
public List<Integer> PreorderTraversal
(TreeNode root){
    if(root==null)
      return null;
```


traversePreOrder(root.rightChild);
}
public void traversePreorder(){
TraversePreOrder(root);
}
InOrder Traversal - (left, root, right)
private void InorderTraversal(Node root){
if(root==null)
return;
InOrderTraversal(root.leftchild)
System.out.println(root.value)
InOrderTraversal(root.rightchild)
}
Note:- With Inorder traversal we get the sorted result (only applicable in BST)
PostOrderTraversal → (left, right, root)
Note: With Postorder traversal we visit all the leaves first then the root node.
private void postOrderTraversal(Node root) {
if(root == null)
return;
postOrderTraversal(root.leftchild);
postOrderTraversal(root.rightchild);
System.out.println(root.value);
}
List<Integer> items = new ArrayList<>();
Stack<Integer> stack = new Stack<Integer>();
stack.push(root);
while(stack.size>0){
TreeNode current = stack.pop();
if(current!=null){
items.add(current.value);
stack.push(current.left);
stack.push(current.right);
}
return items;
}
Iterative Sol:-
{ List<Integer> items = new Arraylist<>();
if(root==null) return items;
Stack<TreeNode> stack = new stack<>();
stack.push(root); TreeNode curr = stack.pop();
while(curr!=null || stack.size>0){
while(curr!=null){
stack.push(curr)
curr = curr.left
}
curr = stack.pop();
items.add(curr.val);
curr = curr.right
}
return items;
}

Depth And Height

There are two properties of tree.

> Note:- The height of the root node is also known as the height of the tree.



D = Depth
H = Height.

Formula for calculating the height of the node:

1 + max(height(leftSubtree), height(rightSubtree))

height is the method which is calling itself recursively.

We use postorder traversal for calculating the height of the tree.

Program to find the height of the tree:

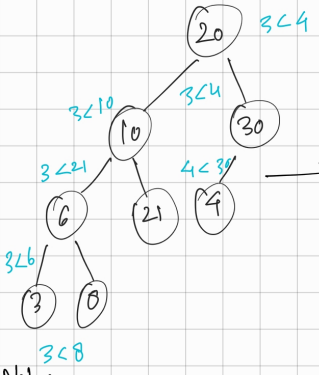
```
private int height(Node root) {
    if(root==null)
        return -1;
    if(root.leftChild==null && root.rightChild==null)
            return 0;
    return (1 + Math.max(height(root.leftChild), height(root.rightChild)))
```

}

The minimum value in the tree: (PostOrder)

So the logic behind finding the min value in a Binary Search tree is pretty easy. To find the min value just go to left part of tree as in BST left node's value is always less than the root node.

To find minimum value in other type of tree, we have to compare the both the child nodes with each other and then compare the min value from them, to root node. For eg.



⇒ So the min value here is 3.

→ Compare every left node with the right one then the result with root node.

Note:-
Here we use PostOrder traversal to compare the nodes.

Implementation: (For other tree)

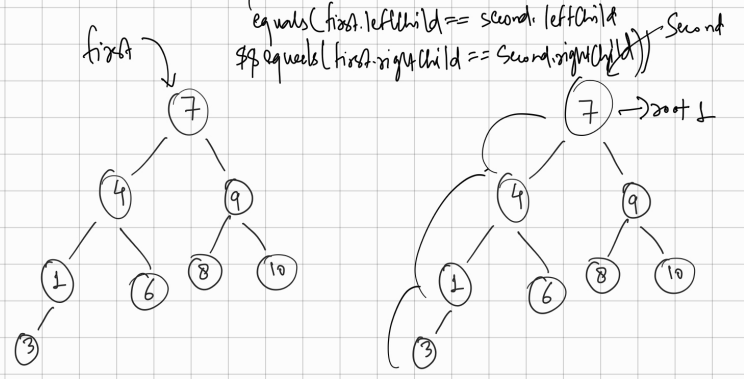
```
private int min (Node root) {
  if (isLeaf(root)) return root.value
  var left = min(root.leftChild)
  var right = min(root.rightChild)
  return Math.min(Math.min(
              left, right), root.value)
} // O(n)
```

Implementation: (for BST)

```
private int min(Node root) {    // O(logn)
 if(root== null) throw gn IllegalStateException();
    var current= root;
     var last = current;
    while ( current != null) {
          last = current;
       current =current. leftChild
    }
   return last.value.
}
```

Interview Question: Check if two trees are equal. (PreOrder)

(first.value == second.value &&
equals(first.leftChild == second. leftChild &&
&& equals(first.rightChild == second.rightChild))



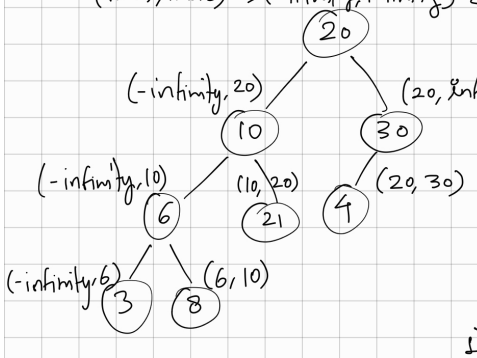
Implementation: —

```
public boolean equals (Tree other) {
   return equals(root, other.root);
}
```

```
private boolean equals(Node first, Node second) {
    if(first == null && second == null)
        return true;
    else if(first != null && second != null)
        return (first.value == second.value &&
                equals(first.leftChild, second.leftChild) &&
                equals(first.rightChild, second.rightChild))
    return false;
}
```

Interview Question: Validating Binary Search tree (Preorder)

(min, max) → (-infinity, infinity) ← This represent the range of values which can be stored.



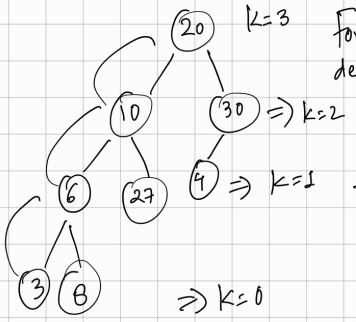
So to solve this problem:
We need a recursive method that have three parameters the node we are visiting, max value and min value.

1) traverse tree using preorder traversal.
2) validate the range of every node. i.e. if they are under the (min, max) range.

```
public boolean isBST(){
    return isBST(root, Integer.MIN_VALUE,
                 Integer.MAX_VALUE);
}
```

```
private boolean isBST(Node root, int min, int max) {
    if (root.leftChild == null || root.rightChild == null)
        return true;

    if(root.value < min || root.value > max)
        return false

    return (isBST(root.leftChild, min, root.value-1)
        && isBST(root.rightChild, root.value+1, max));
}
```

Interview Question: Nodes at K distance from the root. (PostOrder)

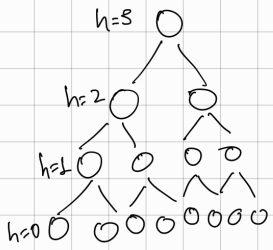


For every level down in the tree K will decrease by 1

```
private void nodesAtKDistance(Node root, K)
    if(K==0)
      System.out.println(root.value);
      return;
    if(root==null && K>0)
      throw new IllegalStateException();
    nodesAtKDistance(root.leftChild, k-1);
    nodesAtKDistance(root.rightChild, k-1);
}
```

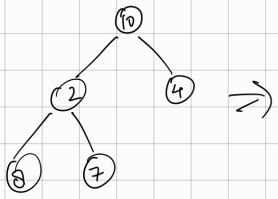
Level Order Traversal (BFS):-

To get every node, level wise then for this we traverse the list based on its height and at every level store the nodes in the list.




```
public void levelOrderTraversal() {
    for(var i=0; i<=height(); i++) {
        for(var item: getNodesAtDistance(i))
            System.out.print(item + " ");
        System.out.println();
    }
}
```

Question: Count No. of leaves in Binary Tree (PostOrder)

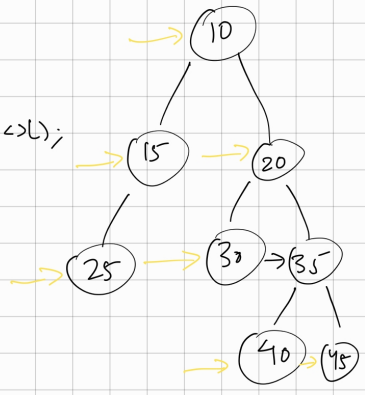


⇒ leaves = 3

```
private int getLeavesCount(Node root){
    if(root==null){
    } return 0
    if(isLeaf(root)) return 1;
    var count = ( getLeavesCount
        (root.leftChild) +
        getLeavesCount(
        root.rightChild)
}
```

BFS iterative implementation:

```
public List<List<Integer>> levelOrderTraversal(TreeNode root) {
    List<List<Integer>> list = new ArrayList<>();
    if(root == null) return list;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    TreeNode curr = root;
    while(!queue.isEmpty()) {
        List<Integer> sublist = new ArrayList<>();
        for(int i=0; i<queue.size(); i++) {
            TreeNode node = queue.remove();
            sublist.add(node.val);
            if(node.left!=null)
                queue.add(node.left);
            if(node.right!=null)
                queue.add(node.right);
        }
        list.add(sublist);
    }
    return list;
}
```



[[10], [15, 20], [25, 30, 35], [40, 45]]

Algorithm:

1) Create a queue
2) add a root in the queue
3) loop through the queue until it is empty
4) iterate the number of nodes present in the queue.
5) If node in the queue have left or right node then add them in the queue.