**Whatsup Design**

**Features:**

*This is the first part of any system design interview, coming up with the features which the system should support. As an interviewee, you should try to list down all the features you can think of which our system should support. Try to spend around 2 minutes for this section in the interview. You can use the notes section alongside to remember what you wrote.*

* **Q:**What is the scale that we are looking at?   
  **A:**Let's assume the scale of Facebook Messages. Let's say we need to hand around 10B message sends a day and around 300M users.
* **Q:**Do we only need to support 1:1 conversations or group conversations as well?   
  **A:**Let's assume we are building things just for 1:1 conversations. We will extend it to group conversations if need be.
* **Q:**Do we need to support attachments?   
  **A:**For now, let's assume that we don’t. We will only look at building plain-text messaging system.
* **Q:**What is a reasonable limit to the size of a message?   
  **A:**Let's assume that we are building a chat messaging system. As such, we would expect every message to be shorter in length. We can impose a limit here on the maximum size of such a message. Let's say we will only handle messages less than 64Kb in size and reject the others.
* **Q:**What about the notification system for new messages received?   
  **A:**Considering the size of the discussion here ( 45 mins in the interview ), we will not delve into the notification system for messages.

**Estimation:**

*This is usually the second part of a design interview, coming up with the estimated numbers of how scalable our system should be. Important parameters to remember for this section is the number of queries per second and the data which the system will be required to handle.   
Try to spend around 5 minutes for this section in the interview.*

Let's estimate the volume of each. Assume that our system would be the one of the most popular messaging service.

**Q:**Given the number of messages being sent, what is the amount of message sent data size we are generating everyday?

**A:**Number of message sends : 10B   
Assuming each message on average has 160 characters , that results in 10B \* 160 = 1.6TB assuming no message metadata.

**Q:**What is the expected storage size?

**A:**From the previous section, we know that we generate 1.6TB data everyday if we only store one copy of the message. If we were to provision for 10 years, we are looking at 1.6 \* 365 \* 10 TB which is approximately 6 Petabytes.

**Design Goals:**

* **Latency** - Is this problem very latency sensitive (Or in other words, Are requests with high latency and a failing request, equally bad?). For example, search typeahead suggestions are useless if they take more than a second.
* **Consistency** - Does this problem require tight consistency? Or is it okay if things are eventually consistent?
* **Availability** - Does this problem require 100% availability?

*There could be more goals depending on the problem. It's possible that all parameters might be important, and some of them might conflict. In that case, you’d need to prioritize one over the other.*

**Q:**Is Latency a very important metric for us?

**A:**Yes. Chat is supposed to be realtime, and hence the end to end time actually matters.

**Q:**How important is Consistency for us?

**A:**Definitely, yes. Its not okay if someone sends me a sequence of message and I don’t see some of them. That could lead to huge confusion. Think of cases when you miss an emergency message or missed messages cause misunderstanding between individuals.

**Q:**How important is Availability for us?

**A:**Availability is good to have. If we had to choose between consistency and availability, consistency wins.

**Skeleton of the design:**

*The next step in most cases is to come up with the barebone design of your system, both in terms of API and the overall workflow of a read and write request. Workflow of read/write request here refers to specifying the important components and how they interact. Try to spend around 5 minutes for this section in the interview.****Important****: Try to gather feedback from the interviewer here to indicate if you are headed in the right direction.*

**Q:**What are the operations that we need to support?

**A:**

* Send a message to another person
* For a user, fetch the most recent conversations
* For every conversation, fetch the most recent messages

**Q:**What would the API look like for the client?

**Q:**How would the *sendMessage* API look like?

**A:**Send Message : Things to take care of in this API

* sendMessage should be idempotent. If a client retries the message, the message should not be added twice. We can resolve this by generating a random timestamp based ID on the client which can be used to de-duplicate the same message being sent repeatedly.
* Ordering of messages should be maintained. If I send message A, and then send message B, then A should always appear before B. However, it is possible that due to delays, if two messages are sent quickly one after another, then the requests reach the DB out of order. How do we solve such a case? Obviously, we need to resolve based on the timestamp when they were sent at.   
  Timestamp on the client is always unreliable. So, we would need to record the timestamp the first time the request hits the servers ( Need not be a part of the API to the client )

sendMessage(senderId, recepientId, messageContent, clientMessageId)

**Q:**How would the API for fetching user's latest conversation look like?

**A:**This API would be called if I need to show a page of conversations/threads ( Think of the view you see when you open the Whatsapp / Messenger app ).   
At a time, only a certain number of conversations will be in the viewport ( let's say 20 ). Let's call it a page of conversations. For a user, we would only want to fetch a page of conversations at a time.   
***Gotcha:***Would the page size remain constant across different situations?   
Probably not. The page size would be different across clients based on screen size and resolution. For example, a mobile’s page size might be lower than that of a web browser’s.

* **Delta fetch**: In most cases, our API calls will be made by users who are active on the site. As such, they already have a view of conversations till a certain timestamp and are only looking for updates after the timestamp ( which would typically be 0-2 more conversations ). For clients which are data sensitive (like mobile), fetching the whole page every time even when I have all of the conversations can be draining. So, we need to support a way of fetching only the updates when the lastFetchedTimestamp is closer to currentTimestamp.

Keeping the above 2 facts in mind, following is how a hybrid API might look like :

ConversationResult fetchConversation(userId, pageNumber, pageSize, lastUpdatedTimestamp)

where ConversationResult has the following fields :

ConversationResult {

List(Conversation) conversations,   
boolean isDeltaUpdate

}   
Conversation {

conversationId,   
participants,   
snippet,   
lastUpdatedTimestamp

}

**Q:**How would the API for fetching most recent messages in a conversation look like?

**A:**Fetch most recent message in a conversation :  
This API is almost identical to the fetchConversation API.

MessageResult fetchMessages(userId, pageNumber, pageSize, lastUpdatedTimestamp)

where MessageResult has the following fields :

MessageResult {

List(Message) messages,   
boolean isDeltaUpdate

}   
Message {

messageId,   
senderId,   
participants,   
messageContent,   
sentTimestamp

}

**A:**The first and last operation ends up doing a write to the database. The other operations are purely read operations. Following is how API's may look like:

* Send Message:

sendMessage(senderId, recepientId, messageContent, clientMessageId)

* Conversations of a user :

ConversationResult fetchConversation(userId, pageNumber, pageSize, lastUpdatedTimestamp)

where ConversationResult has the following fields :

ConversationResult {

List(Conversation) conversations,   
boolean isDeltaUpdate

}   
Conversation {

conversationId,   
participants,   
snippet,   
lastUpdatedTimestamp

}

* Fetch most recent message in a conversation : This API is almost identical to the fetchConversation API.

MessageResult fetchMessages(userId, pageNumber, pageSize, lastUpdatedTimestamp)

where MessageResult has the following fields :

MessageResult {

List(Message) messages,   
boolean isDeltaUpdate

}   
Message {

messageId,   
senderId,   
participants,   
messageContent,   
sentTimestamp

}

**Q:**How would a typical write query look like?

**A:**Components: 

* Client ( Mobile app / Browser, etc ) which calls sendMessage(senderId, recepientId, messageContent, clientMessageId)
* Application server which interprets the API call and calls DB to do the following:
  + Puts in the serverTimestamp
  + Figures out the conversation to which the message should be appended based on the other participant
  + Figures out if a recent message exists with the clientMessageId
  + Store the message
* Database server which stores the message.

**Q:**How would a typical read query look like?

**A:**Components: 

* Client (Mobile app/Browser, etc ) which calls fetchConversation
* Application server which interprets the API call and queries the database for the top conversation.
* Database server which looks up the user’s conversations.

**Deep Dive:**

*Lets dig deeper into every component one by one. Discussion for this section will take majority of the interview time(20-30 minutes).*

Let's dig deeper into every component one by one.

**Application layer:**

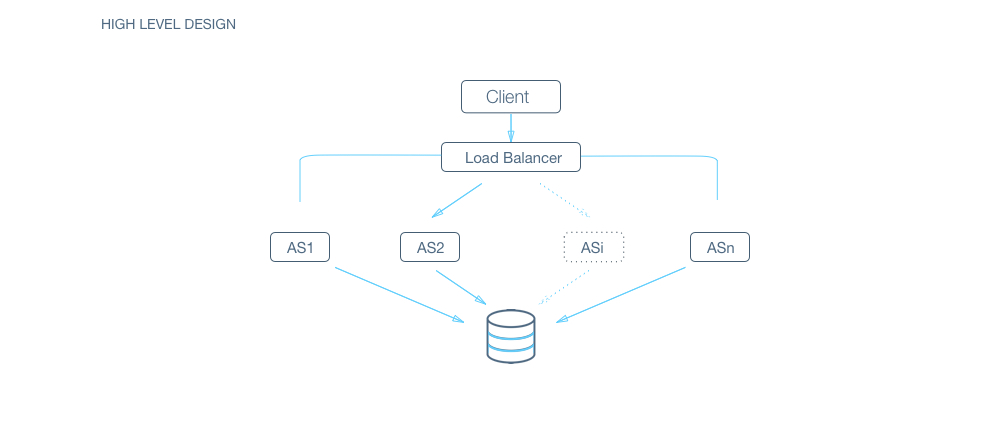
*Think about all details/gotchas yourself before beginning.* 

**Q:**How would you take care of application layer fault tolerance?

**Q:**How do we handle the case where our application server dies?

**A:**The simplest thing that could be done here is to have multiple application server. They do not store any data (stateless) and all of them behave the exact same way when up. So, if one of them goes down, we still have other application servers who would keep the site running.

**Q:**How does our client know which application servers to talk to. How does it know which application servers have gone down and which ones are still working?

**A:**We introduce load balancers. Load balancers are a set of machines (an order of magnitude lower in number) which track the set of application servers which are active ( not gone down ). Client can send request to any of the load balancers who then forward the request to one of the working application servers randomly. 

**A:**If we have only one application server machine, our whole service would become unavailable. Machines will fail and so will network. So, we need to plan for those events. Multiple application server machines along with load balancer is the way to go.

**Database layer**

This is the heart of the question. In the skeleton design, we assumed that the database is a black box which can magically store or retrieve anything efficiently. Let's dig into how we will build that magic black box.

**Q:**RDBMS or NoSQL?

**Q:**Are joins required?

**A:**NoSQL databases are inefficient for joins or handling relations. As such, NoSQL databases store everything in a denormalized fashion. In this case, we do have relations like 

* user -> messages
* user -> conversations
* conversation -> messages

SQL seems to win on this parameter on ease of use.

**Q:**How much data would we have to store?

**A:**If the size of the data is so small that it fits on a single machine’s main memory, SQL is a clear winner. SQL on a single machine has next to zero maintenance overhead and has great performance with right index built. If your index can fit into RAM, its best to go with a SQL solution. In our earlier estimations, we had already established that we will need to provision petabytes of data which most definitely does not fit on a single machine.   
So, a SQL solution will have a sharding overhead. Most NoSQL solutions however are built with the assumption that the data does not fit on a single machine and hence have sharding builtin. NoSQL wins on this parameter.

**Q:**What is the read-write pattern?

**A:**Messaging is going to be very write heavy. Unlike photos or tweets or posts which are written once and then consumed a lot of times by a lot of people, messaging is written once and consumed by the other participant once.   
For a write heavy system with a lot of data, RDBMS usually don’t perform well. Every write is not just an append to a table but also an update to multiple index which might require locking and hence might interfere with reads and other writes.   
However, there are NoSQL DBs like HBase where writes are cheaper. NoSQL seems to win here.

**A:**Things to consider : 

* Are joins required?
* Size of the DB
* Technology Maturity

From the looks of it, NoSQL seems like a better fit. Let's proceed with NoSQL for now.

**Q:**How would we store the data? Discuss schema

**A:**As discussed before, with NoSQL, we need to store the data in denormalized form.   
First thing first, this means that every user would have his/her own copy of the mailbox. That means that we will store 2 copies of the message, one for each participant for every message send.   
Let's delve into how the schema would look. We’ll assume that we are using HBase for this problem.   
If this is your first time designing schema, we strongly recommend you go through a primer [here](https://www.mapr.com/blog/guidelines-hbase-schema-design).   
For schema design, its good to recognize our access patterns. To achieve that, let's look at our major operations :

* For a user, append a message to a conversation
* Fetch timestamp ordered conversations for a user ( Most recent )
* Fetch most recent messages in a conversation for a user ( Most recent )

As you can see, the first lookup for all three operations is for the user. In NoSQL context, it hence makes sense to have userId as the row ID ( Data is sharded based on users ).   
Now, within the user, we will need to lookup conversations, recent conversations and recent messages.   
One naive approach is to fetch the whole list of conversations or all the messages in a conversation and then filter the data we need. This however is really slow ( Remember that one of our design goals was to have low latency ). Even more so, in cases when some popular users get a lot of message and have a huge mailbox (in GBs).   
  
Let's see, how we would solve each read request one by one.

* Recent conversations : We can separately store conversationId : timestamp mapping in the same row ( In case of HBase, in a separate column family). This will not be a lot of data and its okay to read it completely (ofcourse, with caching).
* Recent messages in a conversation : Loading all the messages or loading all the messages in a conversation would make this really expensive. Doing the same thing with messageIds is still an improvement in terms of the amount of data that we have to load.   
  As an improvement, if key in the same index is conversationID\_timestamp, we can use a prefix search of conversationID and use the most recent messages based on timestamp in the key ( assuming, the data is stored sorted with the key ).

**Q:**How would we do sharding?

**A:**HBase inherently use consistent hashing(in this case on user\_id). We explain it in detail at <https://www.interviewbit.com/problems/sharding-a-database/>

**Q:**How would we handle a DB machine going down?

**A:**We explain in detail how to build a reliable and consistent database system in <https://www.interviewbit.com/problems/highly-consistent-database/>

**Q:**What are some other things we can do to increase efficiency of the system?

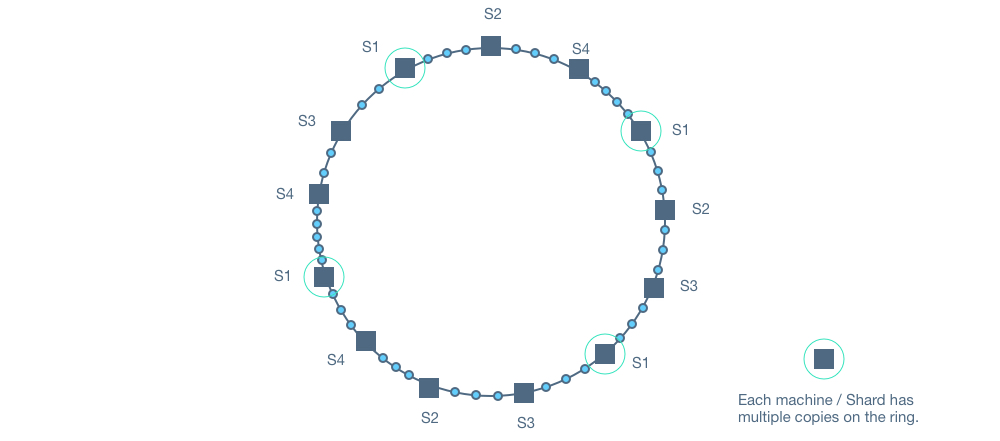
**A:**Caching ( Its the answer in most cases, isn’t it? :) ).   
This one however is not that easy. If you remember, one of our design goals was to ensure tight consistency. Most distributed caching system are good with availability and they become eventually consistent. But they are not tightly consistent.   
For example, let's say I have a distributed cache. If a user’s messages or conversations are spread across machines, then it starts causing trouble for us because :

* The changes are no more atomic. Consider the case when messages for a user are one machine and conversations on another. When a message is added, the update request is sent to the server with messages and server with conversations for this user. There could potentially be a period when one server has processed the update and the other has not.
  + If changes are not atomic, the system is not tightly consistent anymore. I might see different views based on when I query the system.

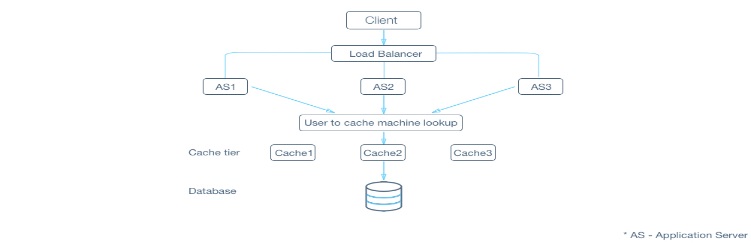
One way to resolve this is to make sure that caching for a user completely resides on one server. The same server can also have other users as well, but users are assigned to exactly one server for caching. To further ensure consistency, all the writes for that user should be directed through this server and this server updates its cache when the write is successful on DB before confirming success.   
  
There are some issues with this system :

* A single point of failure for the user : My reads and writes are routed through this caching server. If this caching server suddenly dies, then my reads and writes suddenly start failing.   
  To resolve this, we should be able to quickly detect the failed machine, mark it as dead and start from scratch ( as cache, reading from DB) on a separate machine. If we have servers for backup, we can just have a [heartbeat mechanism](http://searchenterpriselinux.techtarget.com/definition/Heartbeat)to detect the server going down and we can activate the backup server.
* Need for a reliable routing service : Obviously, if a user is assigned to a server, then there has to be a service which should be able to track that. All request would be routed through this service ( kind of like a load balancer with a lookup based on user\_id ). The machine would need to track the IP of the active server for the user ( If the server goes down, then the routing service should know instantly to route to the new server which starts with a cold cache).   
  Distribution of user into different servers is another problem in itself. If we do static allotment, how do we handle the following cases :
  + More servers are added to the caching pool. How do we re-distribute the users without causing a cold cache for the whole userbase?
  + More users would keep registering. How would they be assigned ensuring uniform load?

What if we did not have a backup server and I had to re-distribute this user’s load into remaining servers?

A classic solution to this problem is consistent hashing with multiple tokens for each server ( See diagram attached ). For more details, read <https://www.interviewbit.com/problems/sharding-a-database/>   


* A minor problem - Multiple concurrent writes : The caching server will also multiple indices corresponding to the mailbox ( the ones for recent conversations / recent messages ). A single write would affect multiple columns. While a NoSQL DB might guarantee atomicity on a row level, in the caching layer, we will have to guarantee it artificially. One simple way of solving it would be to have a user level lock in the caching server for the user which allows only one write operation to go through at a time.

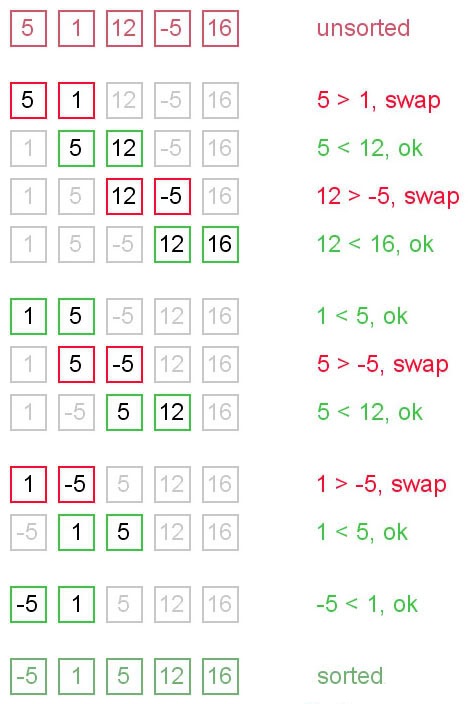


**Sorting Algo:**

**1. Bubble Sorting**

Bubble sort, also referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists.

Bubble sort has worst-case and average complexity both О(n2), where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of O(n log n). Even other О(n2) sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.Performance of bubble sort over an already-sorted list (best-case) is O(n).



public class MyBubbleSort {

    // logic to sort the elements

    public static void bubble\_srt(int array[]) {

        int n = array.length;

        int k;

        for (int m = n; m >= 0; m--) {

            for (int i = 0; i < n - 1; i++) {

                k = i + 1;

                if (array[i] > array[k]) {

                    swapNumbers(i, k, array);

                }

            }

            printNumbers(array);

        }

    }

    private static void swapNumbers(int i, int j, int[] array) {

        int temp;

        temp = array[i];

        array[i] = array[j];

        array[j] = temp;

    }

    private static void printNumbers(int[] input) {

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

            System.out.print(input[i] + ", ");

        }

        System.out.println("\n");

    }

    public static void main(String[] args) {

        int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };

        bubble\_srt(input);

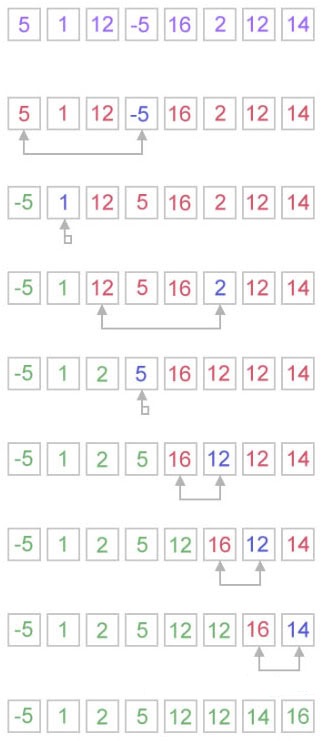
    }

}

**Selection Sort**

The selection sort is a combination of searching and sorting. During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array. The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takesn − 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n − 1 elements and so on, for (n − 1) + (n − 2) + ... + 2 + 1 = n(n − 1) / 2 ∈ Θ(n2) comparisons. Each of these scans requires one swap for n − 1 elements.



public class MySelectionSort {

    public static int[] doSelectionSort(int[] arr){

        for (int i = 0; i < arr.length - 1; i++)

        {

            int index = i;

            for (int j = i + 1; j < arr.length; j++)

                if (arr[j] < arr[index])

                    index = j;

            int smallerNumber = arr[index];

            arr[index] = arr[i];

            arr[i] = smallerNumber;

        }

        return arr;

    }

    public static void main(String a[]){

        int[] arr1 = {10,34,2,56,7,67,88,42};

        int[] arr2 = doSelectionSort(arr1);

        for(int i:arr2){

            System.out.print(i);

            System.out.print(", ");

        }

    }

}

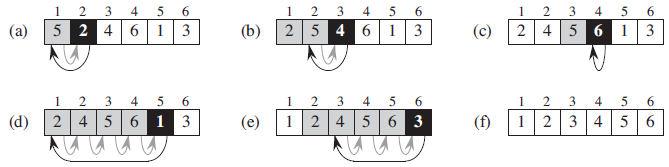
**Insertion sort:**

Insertion sort is a simple sorting algorithm, it builds the final sorted array one item at a time. It is much less efficient on large lists than other sort algorithms.

Advantages of Insertion Sort:   
  
1) It is very simple.  
2) It is very efficient for small data sets.  
3) It is stable; i.e., it does not change the relative order of elements with equal keys.  
4) In-place; i.e., only requires a constant amount O(1) of additional memory space.

Insertion sort iterates through the list by consuming one input element at each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., Θ(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array. The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n2)). The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

****

package com.java2novice.algos;

public class MyInsertionSort {

    public static void main(String a[]){

        int[] arr1 = {10,34,2,56,7,67,88,42};

        int[] arr2 = doInsertionSort(arr1);

        for(int i:arr2){

            System.out.print(i);

            System.out.print(", ");

        }

    }

    public static int[] doInsertionSort(int[] input){

        int temp;

        for (int i = 1; i < input.length; i++) {

            for(int j = i ; j > 0 ; j--){

                if(input[j] < input[j-1]){

                    temp = input[j];

                    input[j] = input[j-1];

                    input[j-1] = temp;

                }

            }

        }

        return input;

    }

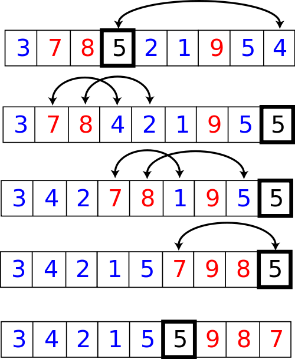
}

**Quick Sort:-**

Quicksort or partition-exchange sort, is a fast sorting algorithm, which is using divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

**Steps to implement Quick sort:**  
  
1) Choose an element, called pivot, from the list. Generally pivot can be the middle index element.  
2) Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.  
3) Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The complexity of quick sort in the average case is Θ(n log(n)) and in the worst case is Θ(n2).



public class MyQuickSort {

    private int array[];

    private int length;

    public void sort(int[] inputArr) {

        if (inputArr == null || inputArr.length == 0) {

            return;

        }

        this.array = inputArr;

        length = inputArr.length;

        quickSort(0, length - 1);

    }

    private void quickSort(int lowerIndex, int higherIndex) {

        int i = lowerIndex;

        int j = higherIndex;

        // calculate pivot number, I am taking pivot as middle index number

        int pivot = array[lowerIndex+(higherIndex-lowerIndex)/2];

        // Divide into two arrays

        while (i <= j) {

            /\*\*

             \* In each iteration, we will identify a number from left side which

             \* is greater then the pivot value, and also we will identify a number

             \* from right side which is less then the pivot value. Once the search

             \* is done, then we exchange both numbers.

             \*/

            while (array[i] < pivot) {

                i++;

            }

            while (array[j] > pivot) {

                j--;

            }

            if (i <= j) {

                exchangeNumbers(i, j);

                //move index to next position on both sides

                i++;

                j--;

            }

        }

        // call quickSort() method recursively

        if (lowerIndex < j)

            quickSort(lowerIndex, j);

        if (i < higherIndex)

            quickSort(i, higherIndex);

    }

    private void exchangeNumbers(int i, int j) {

        int temp = array[i];

        array[i] = array[j];

        array[j] = temp;

    }

    public static void main(String a[]){

        MyQuickSort sorter = new MyQuickSort();

        int[] input = {24,2,45,20,56,75,2,56,99,53,12};

        sorter.sort(input);

        for(int i:input){

            System.out.print(i);

            System.out.print(" ");

        }

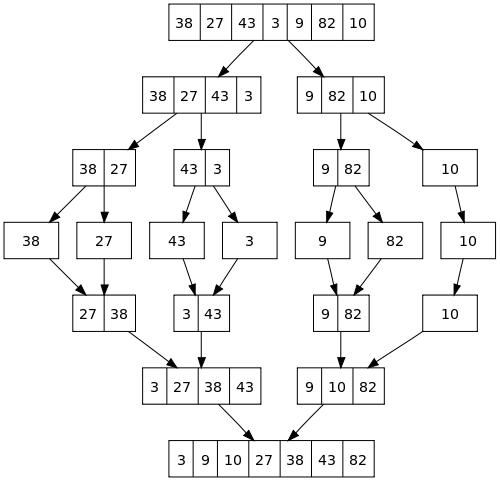
    }

}

**Merge Sort:-**

Merge sort is a divide and conquer algorithm.Steps to implement Merge Sort:  
  
1) Divide the unsorted array into n partitions, each partition contains 1 element. Here the one element is considered as sorted.  
2) Repeatedly merge partitioned units to produce new sublists until there is only 1 sublist remaining. This will be the sorted list at the end.

Merge sort is a fast, stable sorting routine with guaranteed O(n\*log(n)) efficiency. When sorting arrays, merge sort requires additional scratch space proportional to the size of the input array. Merge sort is relatively simple to code and offers performance typically only slightly below that of quicksort.



public class MyMergeSort {

    private int[] array;

    private int[] tempMergArr;

    private int length;

    public static void main(String a[]){

        int[] inputArr = {45,23,11,89,77,98,4,28,65,43};

        MyMergeSort mms = new MyMergeSort();

        mms.sort(inputArr);

        for(int i:inputArr){

            System.out.print(i);

            System.out.print(" ");

        }

    }

    public void sort(int inputArr[]) {

        this.array = inputArr;

        this.length = inputArr.length;

        this.tempMergArr = new int[length];

        doMergeSort(0, length - 1);

    }

    private void doMergeSort(int lowerIndex, int higherIndex) {

        if (lowerIndex < higherIndex) {

            int middle = lowerIndex + (higherIndex - lowerIndex) / 2;

            // Below step sorts the left side of the array

            doMergeSort(lowerIndex, middle);

            // Below step sorts the right side of the array

            doMergeSort(middle + 1, higherIndex);

            // Now merge both sides

            mergeParts(lowerIndex, middle, higherIndex);

        }

    }

    private void mergeParts(int lowerIndex, int middle, int higherIndex) {

        for (int i = lowerIndex; i <= higherIndex; i++) {

            tempMergArr[i] = array[i];

        }

        int i = lowerIndex;

        int j = middle + 1;

        int k = lowerIndex;

        while (i <= middle && j <= higherIndex) {

            if (tempMergArr[i] <= tempMergArr[j]) {

                array[k] = tempMergArr[i];

                i++;

            } else {

                array[k] = tempMergArr[j];

                j++;

            }

            k++;

        }

        while (i <= middle) {

            array[k] = tempMergArr[i];

            k++;

            i++;

        }

    }

}

**Heap Sort:-**

In this post, we will see how to implement heap sort in java.  
I will divide heap sort in multiple parts to make it more understandable.

* What is heap?
* Understanding complete binary tree
* Binary heaps
* Types of heaps
* Heapifying an element
* Steps for heap Sort

#### What is heap?

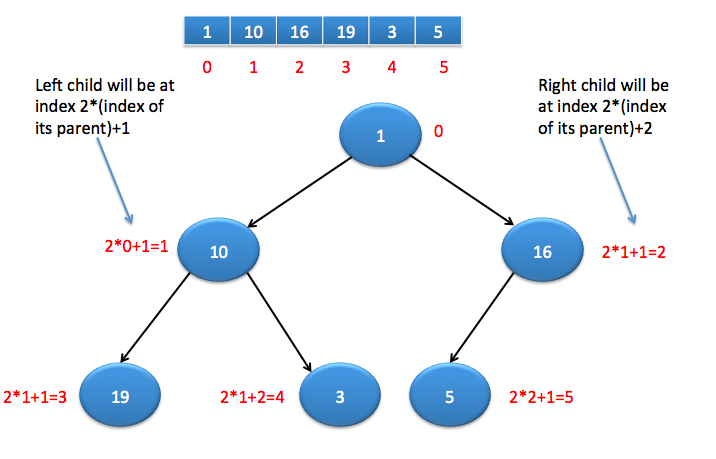
A heap is a tree with some special properties, so value of node should be greater than or equal to(less than or equal to in case of min heap) children of the node and tree should be complete binary tree.

#### Binary heaps

Binary heaps are those heaps which can have up to 2 children. We will use binary heaps in our next few sections.

#### Understanding complete binary tree:

Complete binary tree is a binary tree whose leaves are at h or h-1 level where h is height of the tree.  
Index of left child= 2\*(index of its parent)+1  
Index of right child= 2\*(index of its parent)+2



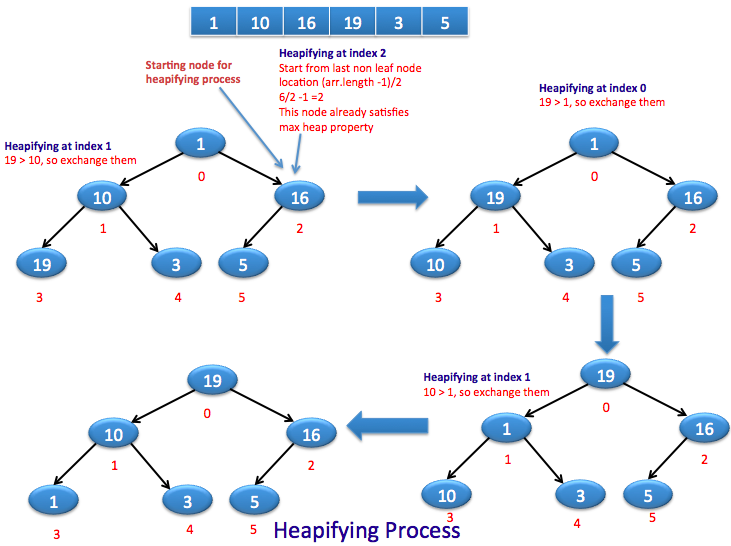
#### Types of heaps

Max heap : It is binary heap where value of node is greater than left and right child of the node.

Min heap : It is binary heap where value of node is lesser than left and right child of the node.

#### Heapifying an element:

Once we create a heap , it may not satisfy heap property. In order to make it heap again, we need to adjust locations of the heap and this process is known as heapifying the elements.  
In order to create a max heap, we will compare current element with its children and find the maximum, if current element is not maximum then exchange it with maximum of left or right child.



#### Steps for heap sort:

* Represent array as complete binary tree.
  + Left child will be at 2\*i+1 th location
  + Right child will be at 2\*i+2 th location.
* Build a heap.
  + All the leaf nodes already satisfy heap property, so we don’t need to heapify them.
  + Last leaf node will be present at (n-1)th location, so parent of it will be at (n-1)/2 th location, hence (n-1)/2 will be location of last non leaf node.
  + Iterate over non leaf nodes and heapify the elements.
* After building a heap, max element will be at root of the heap. We will exchange it with (n-1)th location, so largest element will be at proper place and remove it from the heap by reducing size of n.
* When you exchange largest element, it may disturb max heap property, so you need to again heapify it.
* Once you do above steps until no elements left in heap, you will get sorted array in the end.

public class HeapSortMain {

   public static void buildheap(int []arr) {

    /\*

     \* As last non leaf node will be at (arr.length-1)/2

     \* so we will start from this location for heapifying the elements

     \* \*/

    for(int i=(arr.length-1)/2; i>=0; i--){

     heapify(arr,i,arr.length-1);

      }

   }

   public static void heapify(int[] arr, int i,int size) {

      int left = 2\*i+1;

      int right = 2\*i+2;

      int max;

      if(left <= size && arr[left] > arr[i]){

       max=left;

      } else {

       max=i;

      }

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

       max=right;

      }

      // If max is not current node, exchange it with max of left and right child

      if(max!=i) {

         exchange(arr,i, max);

         heapify(arr, max,size);

      }

   }

   public static void exchange(int[] arr,int i, int j) {

        int t = arr[i];

        arr[i] = arr[j];

        arr[j] = t;

   }

   public static int[] heapSort(int[] arr) {

      buildheap(arr);

      int sizeOfHeap=arr.length-1;

      for(int i=sizeOfHeap; i>0; i--) {

         exchange(arr,0, i);

         sizeOfHeap=sizeOfHeap-1;

         heapify(arr, 0,sizeOfHeap);

      }

      return arr;

   }

   public static void main(String[] args) {

      int[] arr={1,10,16,19,3,5};

      System.out.println("Before Heap Sort : ");

      System.out.println(Arrays.toString(arr));

      arr=heapSort(arr);

      System.out.println("=====================");

      System.out.println("After Heap Sort : ");

      System.out.println(Arrays.toString(arr));

   }

}

**Bucket Sort**

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.   
*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. How do we sort the numbers efficiently?*

A simple way is to apply a comparison based sorting algorithm. The [lower bound for Comparison based sorting algorithm](https://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/)(Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.  
Can we sort the array in linear time? [Counting sort](https://www.geeksforgeeks.org/counting-sort/) can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.   
The idea is to use bucket sort. Following is bucket algorithm.

bucketSort(arr[], n)

1) Create n empty buckets (Or lists).2) Do following for every array element arr[i].

.......a) Insert arr[i] into bucket[n\*array[i]]

3) Sort individual buckets using insertion sort.

4) Concatenate all sorted buckets.

**Time Complexity:** If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket (In the following code, C++ vector is used for simplicity). Step 4 also takes O(n) time as there will be n items in all buckets.  
The main step to analyze is step 3. This step also takes O(n) time on average if all numbers are uniformly distributed (please refer [CLRS book](http://www.flipkart.com/introduction-algorithms-3rd/p/itmdvd93bzvrnc7b?pid=9788120340077&affid=sandeepgfg) for more details)

public class BucketSort{

public static void sort(int[] a, int maxVal) {

int [] bucket=new int[maxVal+1];

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

bucket[i]=0;

}

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

bucket[a[i]]++;

}

int outPos=0;

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

for (int j=0; j<bucket[i]; j++) {

a[outPos++]=i;

}

}

}

public static void main(String[] args) {

int maxVal=5;

int [] data= {5,3,0,2,4,1,0,5,2,3,1,4};

System.out.println("Before: " + Arrays.toString(data));

sort(data,maxVal);

System.out.println("After: " + Arrays.toString(data));

}

}

**Infix to Postfix**

**1.** Scan the infix expression from left to right.  
**2.** If the scanned character is an operand, output it.  
**3.**Else,  
…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.  
…..**3.2** Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.  
**4.** If the scanned character is an ‘(‘, push it to the stack.  
**5.** If the scanned character is an ‘)’, pop and output from the stack until an ‘(‘ is encountered.  
**6.** Repeat steps 2-6 until infix expression is scanned.  
**7.**Pop and output from the stack until it is not empty.

class Test

{

    // A utility function to return precedence of a given operator

    // Higher returned value means higher precedence

    static int Prec(char ch)

    {

        switch (ch)

        {

        case '+':

        case '-':

            return 1;

        case '\*':

        case '/':

            return 2;

        case '^':

            return 3;

        }

        return -1;

    }

    // The main method that converts given infix expression

    // to postfix expression.

    static String infixToPostfix(String exp)

    {

        // initializing empty String for result

        String result = new String("");

        // initializing empty stack

        Stack<Character> stack = new Stack<>();

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

        {

            char c = exp.charAt(i);

             // If the scanned character is an operand, add it to output.

            if (Character.isLetterOrDigit(c))

                result += c;

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

            else if (c == '(')

                stack.push(c);

            //  If the scanned character is an ')', pop and output from the stack

            // until an '(' is encountered.

            else if (c == ')')

            {

                while (!stack.isEmpty() && stack.peek() != '(')

                    result += stack.pop();

                if (!stack.isEmpty() && stack.peek() != '(')

                    return "Invalid Expression"; // invalid expression

                else

                    stack.pop();

            }

            else // an operator is encountered

            {

                while (!stack.isEmpty() && Prec(c) <= Prec(stack.peek()))

                    result += stack.pop();

                stack.push(c);

            }

        }

        // pop all the operators from the stack

        while (!stack.isEmpty())

            result += stack.pop();

        return result;

    }

    // Driver method

    public static void main(String[] args)

    {

        String exp = "a+b\*(c^d-e)^(f+g\*h)-i"; // abcd^e-fgh\*+^\*+i-

        System.out.println(infixToPostfix(exp));

    }

}

Infix to Prefix

The only difference in converting Infix to Prefix from converting Infix to Postfix is that we should reverse the input string and use the same logic and again reverse the output. The underlying logic remains the same.

ALGORITHM : Infix to Prefix

STEP 1 : Read the given infix expression into string called infix.

STEP 2 : Reverse the infix string and read one character at a time and perform the following operations :

If the read character is an operand, then add the operand to the prefix string.

If the read character is not an operand, then check

If the stack is not empty and precedence of the top of the

stack operator is higher than the read operator,

then pop the operator from stack and add this

operator to the prefix string.

Else push the operator onto the stack.

STEP 3 : Repeat STEP 2 till all characters are processed from the input string.

STEP 4 : If stack is not empty, then pop the operator from stack and add this operator to the prefix string.

STEP 5 : Repeat STEP 4 till all the operators are popped from the stack.

STEP 6 : Reverse the prefix string and display the result of the given infix expression or the resultant prefix expression stored in a string called prefix from this algorithm.

Program :

import java.util.Stack;

public class InfixToPrefix {

static Stack inputStack;

static String output = "";

public static void main(String[] args) {

String input = "1+2\*4/5-7+3/6";

int len = input.length();

char[] charr = new char[len];

for (int i = 0; i < len; i++) {

charr[i] = input.charAt(i);

}

String reverseInput = reverse(input);

System.out.println(infixToPrefix(reverseInput));

}

public static String infixToPrefix(String input) {

inputStack = new Stack();

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

char current = input.charAt(i);

if (current == '+' || current == '-') {

isOperator(current, 1);

} else if (current == '\*' || current == '/') {

isOperator(current, 2);

} else {

output += current;

}

}

while (!inputStack.isEmpty()) {

char top = (Character) inputStack.pop();

output += top;

}

output = reverse(output);

return output;

}

public static void isOperator(char c, int prec) {

while (!inputStack.isEmpty()) {

char top = (Character) inputStack.pop();

int topPrec = 0;

if (top == '+' || top == '-') {

topPrec = 1;

} else {

topPrec = 2;

}

if (topPrec >= prec) {

output += top;

} else {

inputStack.push(top);

break;

}

}

inputStack.push(c);

}

public static String reverse(String input) {

int len = input.length();

String reverse = "";

char[] charr = new char[len];

for (int i = 0; i < len; i++) {

charr[i] = input.charAt(i);

}

for (int i = 0; i < len / 2; i++) {

char temp = charr[i];

charr[i] = charr[len - i - 1];

charr[len - i - 1] = temp;

}

for (int j = 0; j < len; j++) {

reverse += charr[j];

}

return reverse;

}

}

**Evaluation of Postfix**

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have discussed [infix to postfix conversion](http://quiz.geeksforgeeks.org/stack-set-2-infix-to-postfix/). In this post, evaluation of postfix expressions is discussed.

Following is algorithm for evaluation postfix expressions.  
1) Create a stack to store operands (or values).  
2) Scan the given expression and do following for every scanned element.  
…..a) If the element is a number, push it into the stack  
…..b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack  
3) When the expression is ended, the number in the stack is the final answer

public class Test

{

// Method to evaluate value of a postfix expression

static int evaluatePostfix(String exp)

{

//create a stack

Stack<Integer> stack=new Stack<>();

// Scan all characters one by one

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

{

char c=exp.charAt(i);

// If the scanned character is an operand (number here),

// push it to the stack.

if(Character.isDigit(c))

stack.push(c - '0');

// If the scanned character is an operator, pop two

// elements from stack apply the operator

else

{

int val1 = stack.pop();

int val2 = stack.pop();

switch(c)

{

case '+':

stack.push(val2+val1);

break;

case '-':

stack.push(val2- val1);

break;

case '/':

stack.push(val2/val1);

break;

case '\*':

stack.push(val2\*val1);

break;

}

}

}

return stack.pop();

}

// Driver program to test above functions

public static void main(String[] args)

{

String exp="231\*+9-";

System.out.println(evaluatePostfix(exp));

}

}

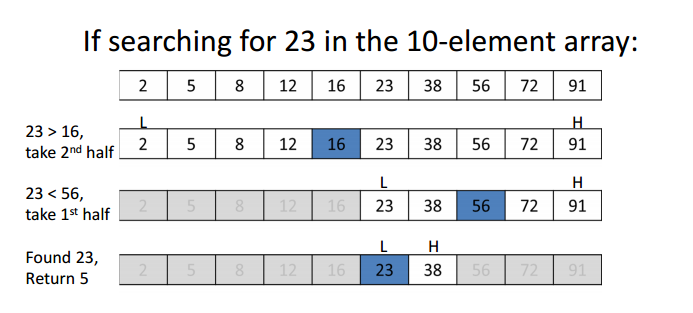
**Binary Search**

**Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].**

**A simple approach is to do**[linear search](http://quiz.geeksforgeeks.org/linear-search/).**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.**

Binary Search:**Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.**

**Example :**



We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

class BinarySearch

{

// Returns index of x if it is present in arr[l..

// r], else return -1

int binarySearch(int arr[], int l, int r, int x)

{

if (r>=l)

{

int mid = l + (r - l)/2;

// If the element is present at the

// middle itself

if (arr[mid] == x)

return mid;

// If element is smaller than mid, then

// it can only be present in left subarray

if (arr[mid] > x)

return binarySearch(arr, l, mid-1, x);

// Else the element can only be present

// in right subarray

return binarySearch(arr, mid+1, r, x);

}

// We reach here when element is not present

// in array

return -1;

}

// Driver method to test above

public static void main(String args[])

{

BinarySearch ob = new BinarySearch();

int arr[] = {2,3,4,10,40};

int n = arr.length;

int x = 10;

int result = ob.binarySearch(arr,0,n-1,x);

if (result == -1)

System.out.println("Element not present");

else

System.out.println("Element found at index " +

result);

}

}

**Binary Tree:-**

**(Array implementation**

**class Tree {**

**public static void main(String[] args)**

**{**

**Array\_imp obj = new Array\_imp();**

**obj.Root("A");**

**// obj.set\_Left("B", 0);**

**obj.set\_Right("C", 0);**

**obj.set\_Left("D", 1);**

**obj.set\_Right("E", 1);**

**obj.set\_Left("F", 2);**

**obj.print\_Tree();**

**}**

**}**

**class Array\_imp {**

**static int root = 0;**

**static String[] str = new String[10];**

**/\*create root\*/**

**public void Root(String key)**

**{**

**str[0] = key;**

**}**

**/\*create left son of root\*/**

**public void set\_Left(String key, int root)**

**{**

**int t = (root \* 2) + 1;**

**if(str[root] == null){**

**System.out.printf("Can't set child at %d, no parent found\n",t);**

**}else{**

**str[t] = key;**

**}**

**}**

**/\*create right son of root\*/**

**public void set\_Right(String key, int root)**

**{**

**int t = (root \* 2) + 2;**

**if(str[root] == null){**

**System.out.printf("Can't set child at %d, no parent found\n",t);**

**}else{**

**str[t] = key;**

**}**

**}**

**public void print\_Tree()**

**{**

**for (int i = 0; i < 10; i++) {**

**if (str[i] != null)**

**System.out.print(str[i]);**

**else**

**System.out.print("-");**

**}**

**}**

**})**

**LinkedList Implementation of Binary Tree**

**class Node**

**{**

**int key;**

**Node left, right;**

**public Node(int item)**

**{**

**key = item;**

**left = right = null;**

**}**

**}**

**// A Java program to introduce Binary Tree**

**class BinaryTree**

**{**

**// Root of Binary Tree**

**Node root;**

**// Constructors**

**BinaryTree(int key)**

**{**

**root = new Node(key);**

**}**

**BinaryTree()**

**{**

**root = null;**

**}**

**public static void main(String[] args)**

**{**

**BinaryTree tree = new BinaryTree();**

**/\*create root\*/**

**tree.root = new Node(1);**

**/\* following is the tree after above statement**

**1**

**/ \**

**null null \*/**

**tree.root.left = new Node(2);**

**tree.root.right = new Node(3);**

**/\* 2 and 3 become left and right children of 1**

**1**

**/ \**

**2 3**

**/ \ / \**

**null null null null \*/**

**tree.root.left.left = new Node(4);**

**/\* 4 becomes left child of 2**

**1**

**/ \**

**2 3**

**/ \ / \**

**4 null null null**

**/ \**

**null null**

**\*/**

**}**

**}**

Depth First Traversals:  
(a) Inorder (Left, Root, Right) : 4 2 5 1 3  
(b) Preorder (Root, Left, Right) : 1 2 4 5 3  
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

**class Node**

**{**

**int key;**

**Node left, right;**

**public Node(int item)**

**{**

**key = item;**

**left = right = null;**

**}**

**}**

**class BinaryTree**

**{**

**// Root of Binary Tree**

**Node root;**

**BinaryTree()**

**{**

**root = null;**

**}**

**/\* Given a binary tree, print its nodes according to the**

**"bottom-up" postorder traversal. \*/**

**void printPostorder(Node node)**

**{**

**if (node == null)**

**return;**

**// first recur on left subtree**

**printPostorder(node.left);**

**// then recur on right subtree**

**printPostorder(node.right);**

**// now deal with the node**

**System.out.print(node.key + " ");**

**}**

**/\* Given a binary tree, print its nodes in inorder\*/**

**void printInorder(Node node)**

**{**

**if (node == null)**

**return;**

**/\* first recur on left child \*/**

**printInorder(node.left);**

**/\* then print the data of node \*/**

**System.out.print(node.key + " ");**

**/\* now recur on right child \*/**

**printInorder(node.right);**

**}**

**/\* Given a binary tree, print its nodes in preorder\*/**

**void printPreorder(Node node)**

**{**

**if (node == null)**

**return;**

**/\* first print data of node \*/**

**System.out.print(node.key + " ");**

**/\* then recur on left sutree \*/**

**printPreorder(node.left);**

**/\* now recur on right subtree \*/**

**printPreorder(node.right);**

**}**

**// Wrappers over above recursive functions**

**void printPostorder() { printPostorder(root); }**

**void printInorder() { printInorder(root); }**

**void printPreorder() { printPreorder(root); }**

**// Driver method**

**public static void main(String[] args)**

**{**

**BinaryTree tree = new BinaryTree();**

**tree.root = new Node(1);**

**tree.root.left = new Node(2);**

**tree.root.right = new Node(3);**

**tree.root.left.left = new Node(4);**

**tree.root.left.right = new Node(5);**

**System.out.println("Preorder traversal of binary tree is ");**

**tree.printPreorder();**

**System.out.println("\nInorder traversal of binary tree is ");**

**tree.printInorder();**

**System.out.println("\nPostorder traversal of binary tree is ");**

**tree.printPostorder();**

**}**

**}**