

Algorithm Analysis

Some Related Formulas

➤ Exponents

Some of the formulas that are helpful are:

- i. $x^a x^b = x^{a+b}$
- ii. $x^a / x^b = x^{a-b}$
- iii. $(x^a)^b = x^{ab}$
- iv. $x^n + x^n = 2x^n$
- v. $2^n + 2^n = 2^{n+1}$

➤ Logarithms

Some of the formulas that are helpful are:

- i. $\log_a b = \log_c b / \log_c a ; c > 0$
- ii. $\log ab = \log a + \log b$
- iii. $\log a/b = \log a - \log b$
- iv. $\log (a^b) = b \log a$
- v. $\log x < x$ for all $x > 0$
- vi. $\log 1 = 0, \log 2 = 1, \log 1024 = 10.$
- vii. ${}_a \log b^n = n \log b^a$

Techniques for Solving Recurrences

Four techniques for solving recurrences are:

➤ Substitution

Takes two steps:

1. Guess the form of the solution, using unknown constants.

2. Use induction to find the constants & verify the solution.
3. Completely dependent on making reasonable guesses

Q. Solve $T(n) = 2T(n/2) + n$ using substitution

- **Base case:** Guess $T(n) \leq c * n \log n$ for some constant c (that is, $T(n) = O(n \log n)$). Now, we need to show that our guess holds for some base case.
- **Inductive Case:** Now, we need to prove that holds for n :

$$T(n) \leq c * n \log n.$$

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \leq 2\left(c * \frac{n}{2} \log \frac{n}{2}\right) + n \\ &= c * n \log \frac{n}{2} + n \\ &= c * n \log n - c * n \log 2 + n \\ &= c * n \log n - c * n + n \end{aligned}$$

This is true if $c > 1$ and guess is also true

Q. Solve $T(n) = 4T(n/2) + n$ using substitution.

➤ Iteration method

In the iteration method we iteratively “clarify” the recurrence until we “see the pattern”. The iteration method does not require making a good guess like the substitution method.

Q. Solve the following recurrence relation using iteration method $T(n) = 3T\left(\frac{n}{3}\right) + n^3$

Soln:

$$\text{Given } T(n) = 3T\left(\frac{n}{3}\right) + n^3$$

$$T(1) = 0$$

K=1	$T(n) = 3T\left(\frac{n}{3}\right) + n^3$
-----	---

	$T\left(\frac{n}{3}\right) = 3T\left(\frac{n/3}{3}\right) + \left(\frac{n}{3}\right)^3$ $= 3T\left(\frac{n}{9}\right) + \left(\frac{n}{3}\right)^3$
K=2	$T(n) = 3 \left[3T\left(\frac{n}{9}\right) + \left(\frac{n^3}{27}\right) \right] + n^3$ $= 9 T\left(\frac{n}{9}\right) + \left(\frac{n^3}{9}\right) + n^3$ $T\left(\frac{n}{9}\right) = \left[3T\left(\frac{n/9}{3}\right) + \left(\frac{n^3}{9^3}\right) \right]$ $= \left[3T\left(\frac{n}{27}\right) + \left(\frac{n^3}{9^3}\right) \right]$
K= 3	$T(n) = 9 \left[3T\left(\frac{n}{27}\right) + \left(\frac{n^3}{9^3}\right) \right] + \left(\frac{n^3}{9}\right) + n^3$ $= 27 T\left(\frac{n}{27}\right) + \left(\frac{n^3}{9^2}\right) + \left(\frac{n^3}{9}\right) + n^3$

Now,

The General form is

$$\begin{aligned}
 T(n) &= 3^k T\left(\frac{n}{3^k}\right) + \left\{ \left(\frac{1}{9^{k-1}}\right) + \left(\frac{1}{9^{k-2}}\right) + \dots + \left(\frac{1}{9^0}\right) \right\} * n^3 \\
 &= 3^k T\left(\frac{n}{3^k}\right) + \left(\sum_{i=0}^{k-1} \frac{1}{9^i} \right) n^3 \\
 &= 3^k T\left(\frac{n}{3^k}\right) + \left(\frac{1 - (1/9)^k}{1 - 1/9} \right) n^3 \\
 &\quad \left(\text{Since } \left(\sum_{i=0}^{n-1} a^i = \frac{1 - (a)^n}{1 - a} \right) \right) \\
 &= 3^k T\left(\frac{n}{3^k}\right) + \left(\frac{1 - 1/9^k}{8/9} \right) n^3 \quad \dots\dots\dots (I)
 \end{aligned}$$

We can have, $\left(\frac{n}{3^k} = 1\right)$ since $T(1) = 0$

$$n = 3^k$$

$$\log_3 n = k$$

Replacing the value of k in eqn (I)

$$\begin{aligned}
&= 3^{\log 3n} T\left(\frac{n}{3^{\log 3n}}\right) + \left(\frac{1-1/9^{\log 3n}}{8/9}\right) n^3 \\
&= n T\left(\frac{n}{n}\right) + \left(\frac{1-1/n^2}{8/9}\right) n^3 \\
&= n T(1) + \frac{9}{8} \left(1 - \frac{1}{n^2}\right) n^3 \\
&= n * 0 + \frac{9}{8} (n^3 - n) \\
&= \frac{9}{8} (n^3 - n)
\end{aligned}$$

Q. Solve the recurrence relation $T(n) = 2T(n/2) + 1$ when $n > 1$ using iteration method.

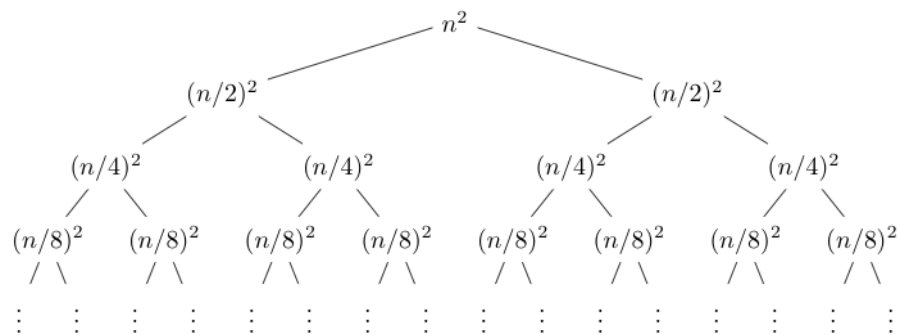
➤ Recursion Tree

A recursion tree is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

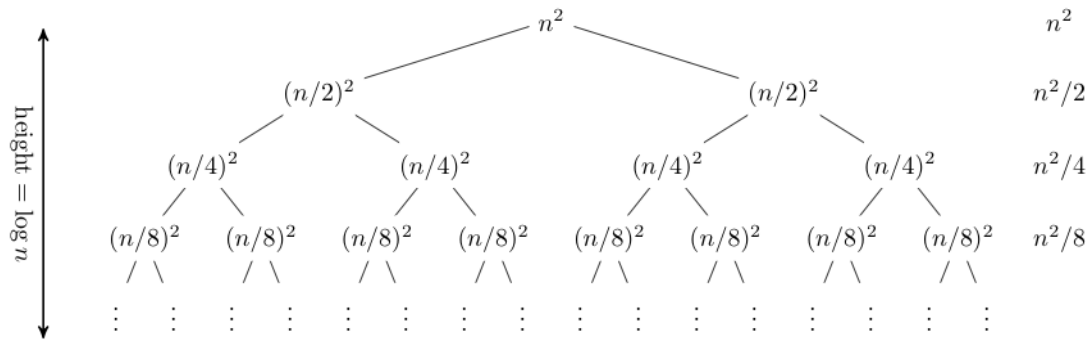
For Example, Consider the following recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

The recursion tree for this recurrence has the following form



In this case, it is straight forward to sum across each row of the tree to obtain the total work done at a given level



Here, $T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \dots + \frac{n^2}{2^{k-1}} + \frac{n^2}{2^k}$

$$T(n) = n^2 \left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \dots + \frac{1}{2^{k-1}} + \frac{1}{2^k} \right)$$

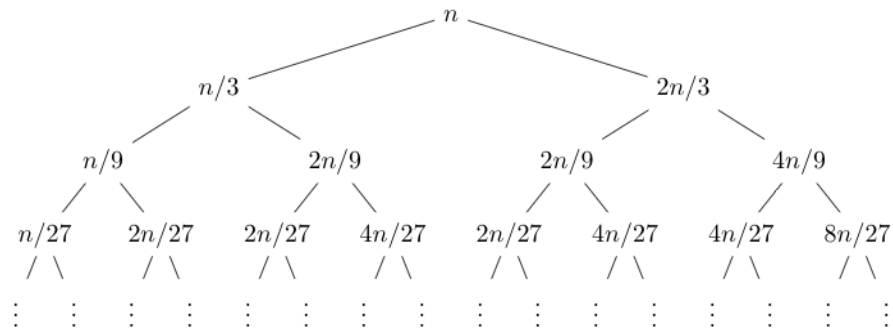
$$T(n) \leq n^2 \left(\frac{1 - \frac{1}{2^{k+1}}}{1 - \frac{1}{2}} \right)$$

$$T(n) = O(n^2)$$

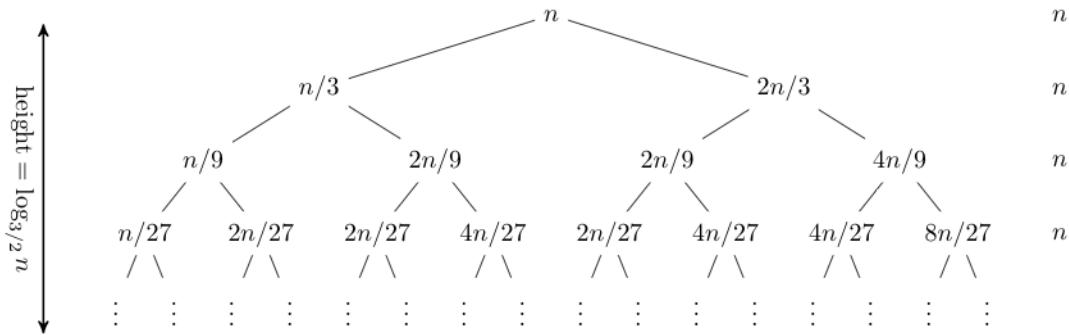
This is a geometric series, thus in the limit the sum is $O(n^2)$.

Q. Solve the relation $T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

Expanding out the first few levels, the recurrence tree is



In this case, the tree here is not balanced and the work done at a given level is



The recurrence is closed to $O(n \log n)$

Q. Solve $T(n) = 4T(n/2) + n$ by using Recurrence tree.

➤ Master Method

The master method is a cookbook method for solving recurrence, although it cannot solve all the recurrences. It is useful in many cases for “divide and conquer” algorithms.

Master method is used to solve the recurrences of the form

$$T(n) = a T(n/b) + f(n)$$

Where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is asymptotically positive function. This method is very handy because most of the recurrence relations we encounter in the analysis of algorithms are of the above kinds and it is very easy to use this method.

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non negative integers by the recurrence

$$T(n) = a T(n/b) + f(n)$$

Case I: The running time is dominated by the cost of leaves

$$\text{If } f(n) = O(n^{\log_b a - \epsilon}) \text{ for some } \epsilon > 0 \text{ Then } T(n) = \Theta(n^{\log_b a})$$

Case II: The running time is evenly distributed throughout the tree

$$\text{If } f(n) = \Theta(n^{\log_b a}) \text{ Then } T(n) = \Theta(n^{\log_b a} \log n)$$

Case III: The running time is dominated by the cost at the root

$$\text{If } f(n) = \Omega(n^{\log_b a + \epsilon}) \text{ Then } T(n) = \Theta(f(n))$$

Q. How to apply the master method.

If $T(n) = a T(n/b) + f(n)$

Then,


- Extract a , b , and $f(n)$ from the given recurrence
- Determine $n^{\log_b a}$
- Compare $f(n)$ and $n^{\log_b a}$ asymptotically
- Determine the appropriate Master Method Case and apply it.

Q. Solve the recurrence relation $T(n) = 2T(n/2) + n$ by using Master method

Here,

→ $a=2, b=2$ and $f(n) = n$

→ $n^{\log_b a} = n^{\log_2 2} = n^1 = n$

→ Compare $n^{\log_b a} = n$ and $f(n) = n$ → 

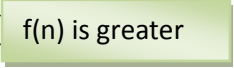
→ Thus, $T(n) = \Theta(n^{\log_b a} \log n)$
 $= \Theta(n^1 \log n)$
 $= \Theta(n \log n)$

Q. Solve the recurrence relation $T(n) = 3T(n/4) + n \log(n)$ by using Master method

Here,

→ $a=3, b=4$ and $f(n) = n \log(n)$

→ $n^{\log_b a} = n^{\log_4 3}$

→ Compare $n^{\log_b a} = n^{\log_4 3}$ and $f(n) = n \log(n)$ → 

→ Thus, $T(n) = \Theta(f(n))$
 $= \Theta(n \log(n))$

Q. Solve the recurrence relation $T(n) = 9T(n/3) + n$ by using Master method

Here,

$$\rightarrow a=9, b=3 \text{ and } f(n) = n$$

$$\rightarrow n^{\log_b a} = n^{\log_3 9} = n^2$$

$$\rightarrow \text{Compare } n^{\log_b a} = n^2$$

$$f(n) = n$$

f(n) is Smaller

$$\rightarrow T(n) = \Theta(n^{\log_b a})$$

$$= \Theta(n^2)$$

Data Structure Review

Data Structure

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

Simple, two types of data structures are

- a. Linear Data Structure
- b. Tree Data Structure

Linear Data Structure

A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays, Linked Lists

- a. List

List is the simplest general-purpose data structure. They are of different variety. Most fundamental representation of a list is through an array representation. The other representation includes linked list. There are also varieties of representations for lists as linked list like singly linked, doubly linked, circular, etc.

List Operation

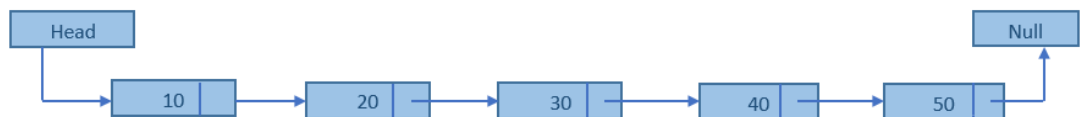
Following are the basic operations supported by a list.

- Insertion – Adds an element at the beginning of the list.
- Deletion – Deletes an element at the beginning of the list.
- Display – Displays the complete list.
- Search – Searches an element using the given key.
- Delete – Deletes an element using the given key.

Types of List

☞ Singly Linked list

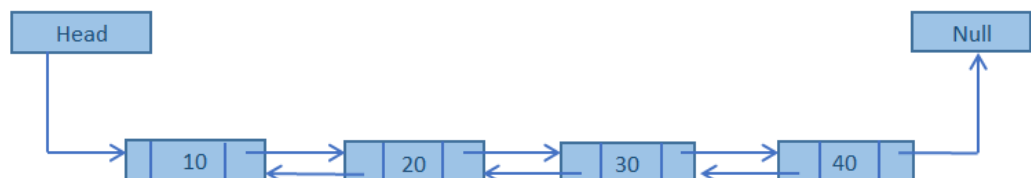
In this type of linked list two nodes are linked with each other in sequential linear manner. Movement in forward direction is possible



In Singly Link List, Insert and delete can be done in $O(1)$ time if the pointer to the node is given, otherwise $O(n)$ time. Likewise Search and traversing can be done in $O(n)$ time.

☞ Doubly Linked list

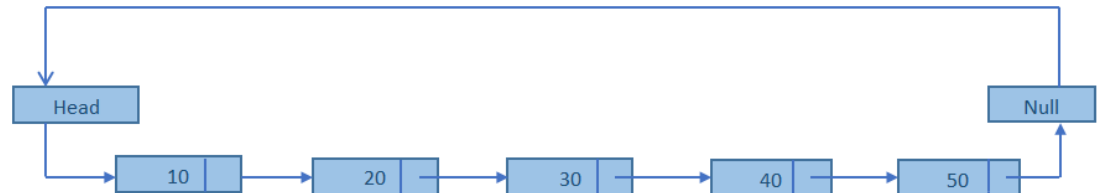
In this type of linked list each node holds two-pointer field. Pointers exist between adjacent nodes in both directions. The list can be traversed either forward or backward.



In Doubly Link List, Insert and delete can be done in $O(1)$ time if the pointer to the node is given, otherwise $O(n)$ time. Likewise Search and traversing can be done in $O(n)$ time.

☞ Circular Linked list

In circular linked list the first and last node are adjacent.



In Circular Link List, Insert and delete can be done in $O(1)$ time if the pointer to the node is given, otherwise $O(n)$ time. Likewise Search and traversing can be done in $O(n)$ time.

b. Stack and Queues

Stack

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

Stack also called LIFO (Last in First Out) list. In this structure items can be added or removed from only one end. Stacks are generally represented either in array or in singly linked list and in both cases insertion/deletion time is $O(1)$, but search time is $O(n)$.

Stack Operation and Their Complexities

→ `isEmpty ();`

Return true if and only if this stack is empty. Complexity is $O(1)$.

→ `int getLast ();`

Return the element at the top of this stack. Complexity is $O(1)$.

→ `void clear ();`

Make this stack empty. Complexity is $O(1)$.

→ `void push (int elem);`

Add elem as the top element of this stack. Complexity is $O(1)$.

→ `int pop ();`

Remove and return the element at the top of this stack. Complexity is $O(1)$.

Queue

The queues are also like stacks but they implement FIFO (First In First Out) policy. One end is for insertion and other is for deletion. They are represented mostly circularly in array for $O(1)$ insertion/deletion time. Circular singly linked representation takes $O(1)$ insertion time and $O(1)$ deletion time. Again Representing queues in doubly linked list have $O(1)$ insertion and deletion time.

Operations on queues

→ `isEmpty ();`

Return true if and only if this queue is empty. Complexity is $O(1)$.

→ `int size ();`

Return this queue's length. Complexity is $O(n)$.

→ `int getFirst ();`

Return the element at the front of this queue. Complexity is $O(1)$.

→ `void clear ();`

Make this queue empty. Complexity is $O(1)$.

→ `void insert (int elem);`

Add elem as the rear element of this queue. Complexity is $O(1)$.

→ `int delete ();`

Remove and return the front element of this queue. Complexity is $O(1)$.

c. Array

.....Assignment 1.....

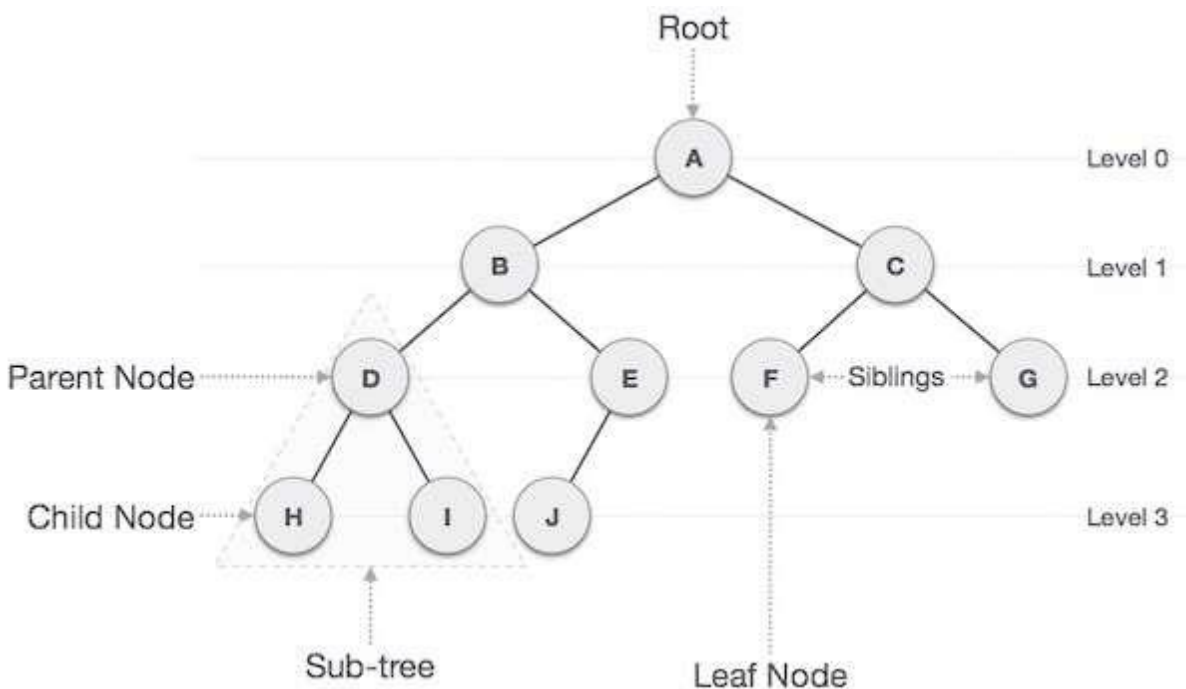
Tree Data Structures

Tree is a collection of nodes .Tree represents the nodes connected by edges. If the collection is empty the tree is empty otherwise it contains a distinct node called root (r). The root of each tree is called child of r, and r the parent.

Any node without a child is called leaf.

The main concern with this data structure is due to the running time of most of the operation require $O(\log n)$.

We can represent tree as an array or linked list.

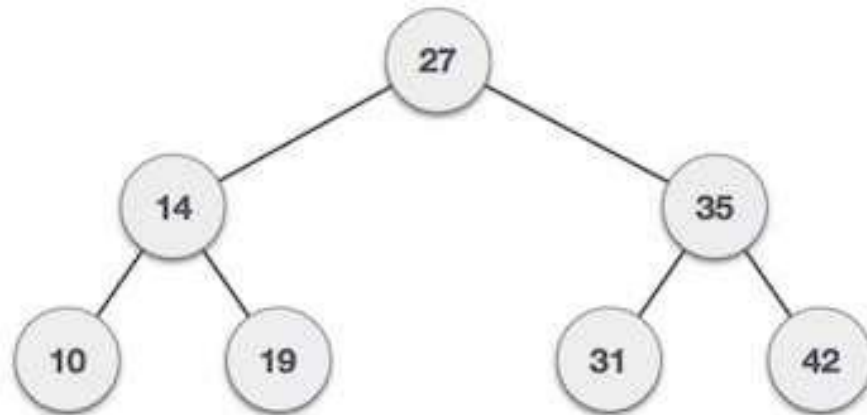


- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – any node except the root node has one edge upward to a node called parent.
- **Child** – The node below a given node connected by its edge downward is called its child node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Subtree** – Subtree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.

- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



If height of the Tree of node n is h , then

Operation	Time Complexity
BST search	$O(\log n)$ Average $O(n)$ worst
BST insertion	$O(\log n)$ Average $O(n)$ worst
BST deletion	$O(\log n)$ Average $O(n)$ worst

AVL Trees

An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child sub trees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.

If n is the node of the AVL Tree then,

Operation	Time Complexity
BST search	$O(\log n)$ Average $O(\log n)$ worst
BST insertion	$O(\log n)$ Average $O(\log n)$ worst
BST deletion	$O(\log n)$ Average $O(\log n)$ worst

Priority Queues

Priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue

Priority queues can be implemented by using arrays, linked list.

If n is the node, then

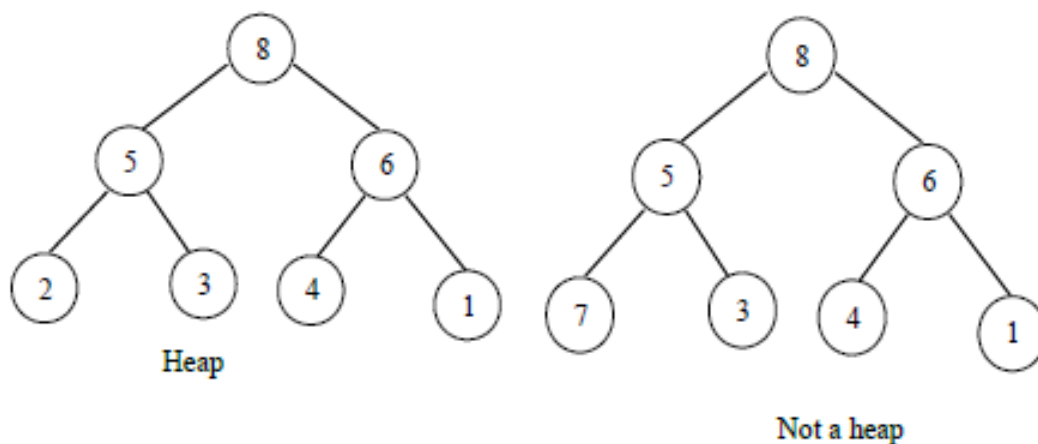
Operation	Sorted SLL	Unsorted SLL	Sorted Array	Unsorted Array
add	$O(n)$	$O(1)$	$O(n)$	$O(1)$
removeLeast	$O(1)$	$O(n)$	$O(1)$	$O(n)$
getLeast	$O(1)$	$O(n)$	$O(1)$	$O(n)$

Heaps

Heap is a specialized tree-based data structure that satisfies the heap property: If A is a parent node of B , then the key (the value) of node A is ordered with respect to the key of node B with the same ordering applying across the heap.

A heap is a complete tree with an ordering-relation R holding between each node and its descendant. Note that the complete tree here means tree can miss only rightmost part of the bottom level. R can be smaller-than, bigger-than.

E.g. Heap with degree 2 and R is "bigger than"



Operation	Time complexity
add	$O(\log n)$
delete	$O(\log n)$
getLeast access root element	$O(1)$

Hash Table

A hash table (hash map) is a data structure used to implement an array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Operation	Time complexity
Search	$O(1)$ Average, $O(n)$ worst Case
Insert	$O(1)$ Average, $O(n)$ worst Case
Delete	$O(1)$ Average, $O(n)$ worst Case