Trees

- References:

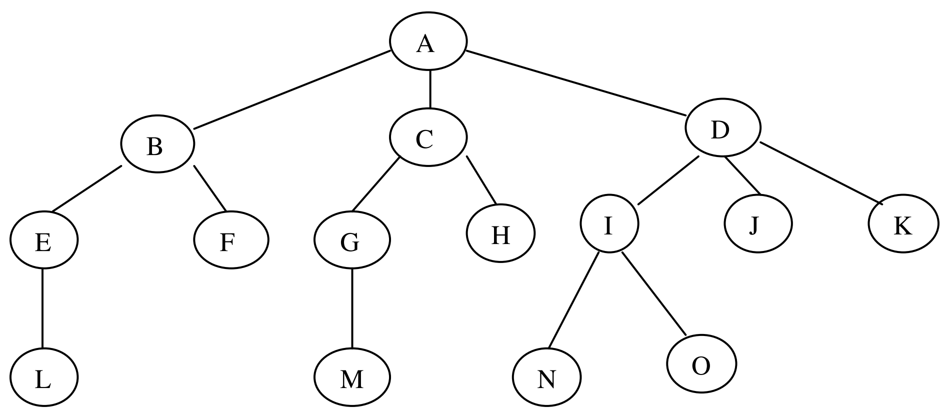
  - Text book : Chapter 23, 24, 25 & 26
  - Only include concepts & important algorithms

1. Introduction

- Example Applications :
  Family Trees
  Company organization
  Computer file directories

- Terminology

  A tree is a data structure that organizes data in hierarchical way
  Example :



  Note : all examples below will refer to the above tree

- Definition: A *tree* is a finite set of one or more nodes such that

  - There is a specially designated node called the *root*

  - The remaining nodes are partitioned into n >= 0 disjoint sets $T_1, T_2, \ldots, T_n$, where each of these sets is a tree. $T_1, T_2, \ldots, T_n$ are called the *subtrees* of the root.

- Nodes are connected by edges. Edges indicate relationship among nodes
- Note : may consider the empty tree is a tree with 0 node

- Definition : The number of subtrees of a node is called its *degree*

  Example :

Node	Degree
A	3
B	2
E	1
M	0

- Definition : Nodes that have degree zero are called *leaf nodes* or *terminal nodes*.

  Example : L,F,M,H,N,O,J,K are leaf nodes

- Definition : Nodes that are not leaf nodes are called *interior nodes* or *nonterminal nodes.*

  Example : A,B,C,D,E,G,I are interior nodes

- Definition : The roots of the subtrees of a node X, are called the *children* of X. X is the *parent* of its children.

  Example :

Node	Children
A	B, C and D
G	M

Node	Parent
G	C
N	I

- You may also use the notions of grandchildren, grand parent etc

- Definition : Children of the same parent are called *siblings*

  Example :

Siblings
B, C, D
I, J, K
N,O

- Definition : The *degree of a tree* is the maximum degree of the nodes in the tree

  Example : Degree of the above tree is 3

- Definition : The *ancestors* of a node are all the nodes along the path from the root to that node

  Example :

Node	Ancestors
M	A, C and G
E	B, A

- Definition : The *descendants* of a node are all nodes in all the subtrees of the node

  Example :

Node	Descendents
A	B, C, D, …, O
B	E, L, F
D	I, J,K,N,O
G	M
N	none

- Property : There is only one path from the root of a tree to every other node

- Definition : The *level* of a node is defined recursively as follows :

  The root of a tree is at level 1

  If a node is at level k, then its children are at level k+1

  Example :

Node	Level
A	1
B, C, D	2
M	4

- Definition : The *height* (or *depth*) of a tree is defined to be the maximum level of any node in the tree

  Example : The height of the above tree is 4

  May consider the height of an empty tree is 0

- Definition: A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint trees called the *left subtree* and the *right subtree.*

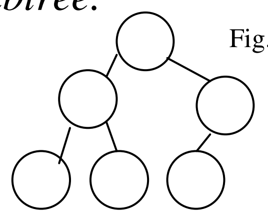


Fig.1

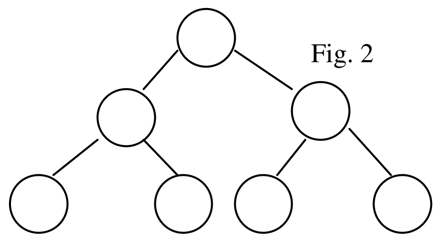


Fig. 2

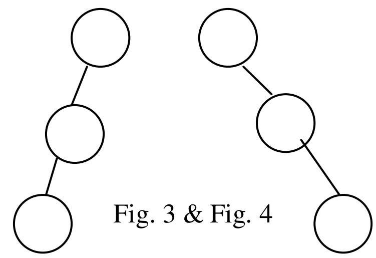


Fig. 3 & Fig. 4

- In general, the degree of a binary tree is two

- Definition : a *full binary tree* is a binary tree where all interior nodes have 2 children and all terminal nodes are in the same level. Example : Figure 2.

- Definition : a *leftist (rightist)* tree is a binary tree where every interior node has only a *left (right)* subtree. Example : Figure 3 (Figure 4)

- Definition : A *balanced binary tree* is a binary tree with n nodes and its height is O(log n).

- Definition : a *complete binary tree* is a full binary tree except that some of the rightmost leave may be missing. Example : Figure 1

- Note : A full binary tree is a complete binary tree. A complete binary tree is balanced tree

- Properties :

  The maximum number of nodes on level j of a binary tree is 2^{j-1}, j >= 1
  The maximum number of nodes in a binary tree of height k is 2^k-1, k >= 1

- Binary Tree Traversal

  - There are 3 ways to traverse all the nodes of a binary tree and produce a listing. Note : Each node only visit once

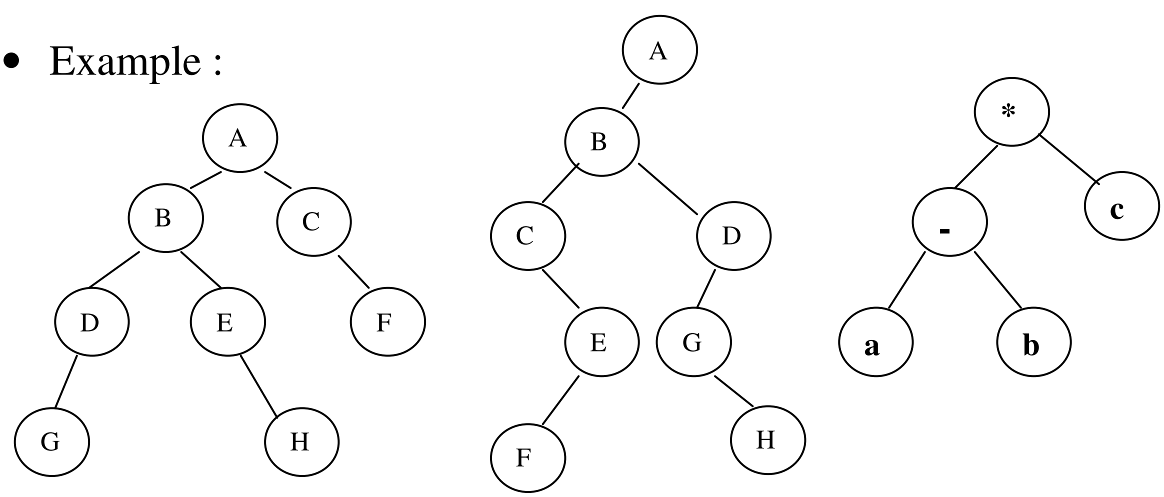
  - Let T be a binary tree with root r

  - Inorder Traversal of T is
    - an inorder traversal of all nodes in the left subtree of r
    - r
    - an inorder traversal of all nodes in the right subtree of r

  - Preorder Traversal of T is
    - r
    - a preorder traversal of all nodes in the left subtree of r
    - a preorder traversal of all nodes in the right subtree of r

  - Postorder Traversal of T is
    - a postorder traversal of all nodes in the left subtree of r
    - a postorder traversal of all nodes in the right subtree of r
    - r

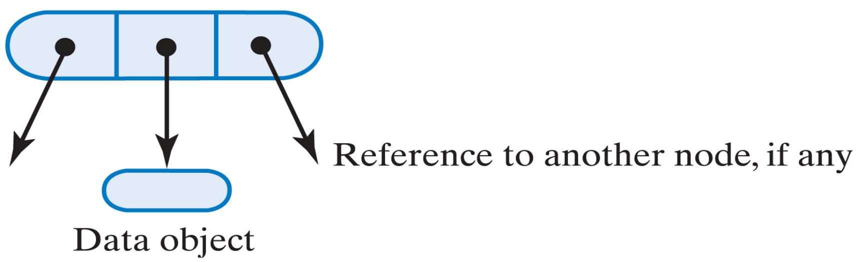
  - Example :

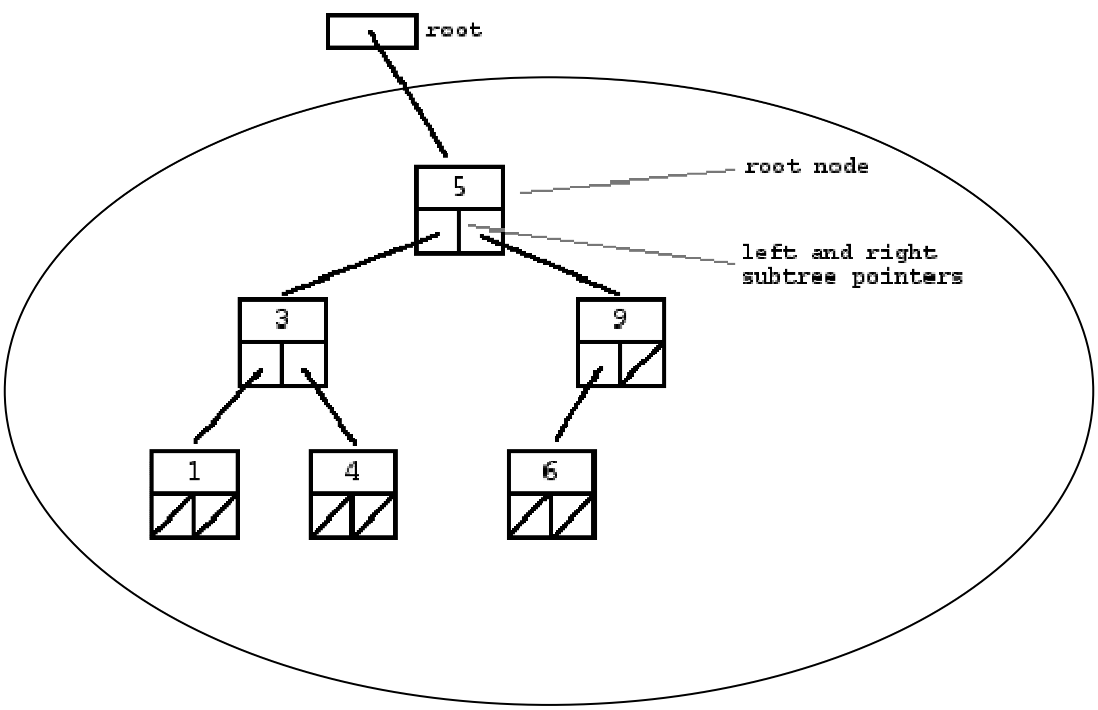


	Tree 1	Tree 2	Tree 3
Inorder	GDBEHACF	CFEBGHDA	a-b*c
Preorder	ABDGEHCF	ABCEFDGH	*-abc
Postorder	GDHEBFCA	FECHGDBA	ab-c*

- Most common Binary Tree Implementation :

  - Linked structure
  - Just need to know reference to root node
  - A node in binary tree: Ref. left child node, Data, Ref right child node





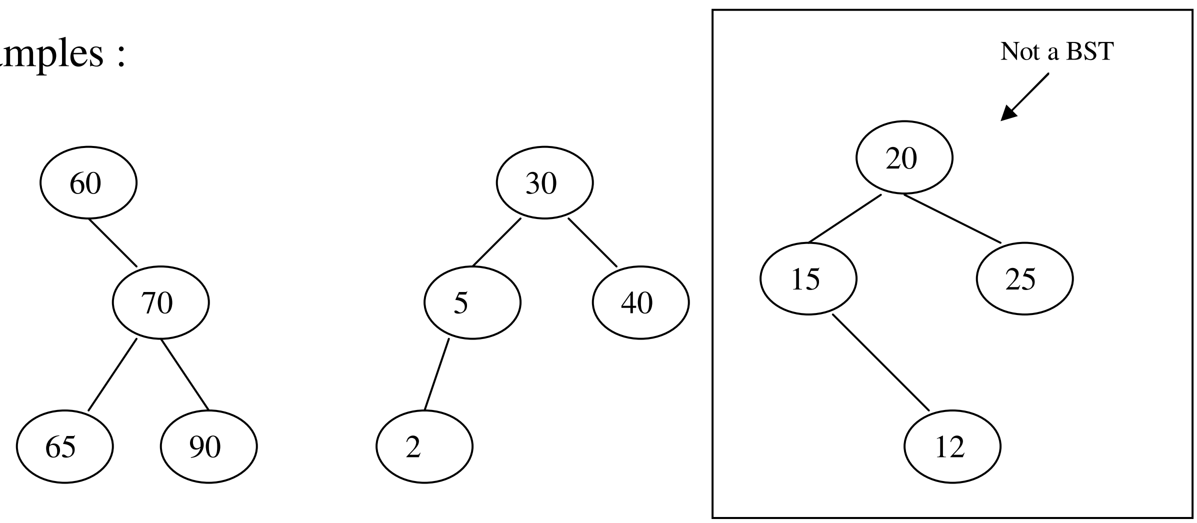
3. Binary Search Tree

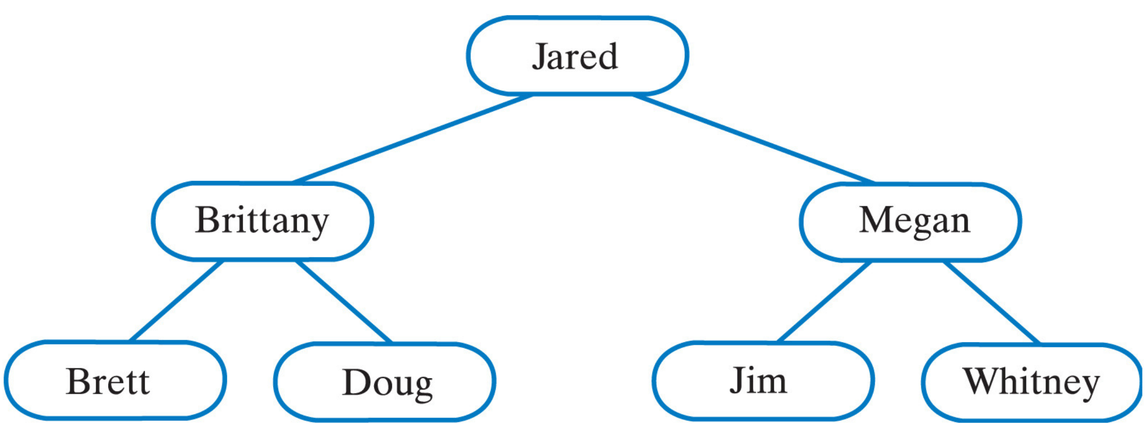
- Problem: Binary tree does not support "search" operation efficiently

- Definition: a binary search tree is a binary tree that satisfies the following 3 properties

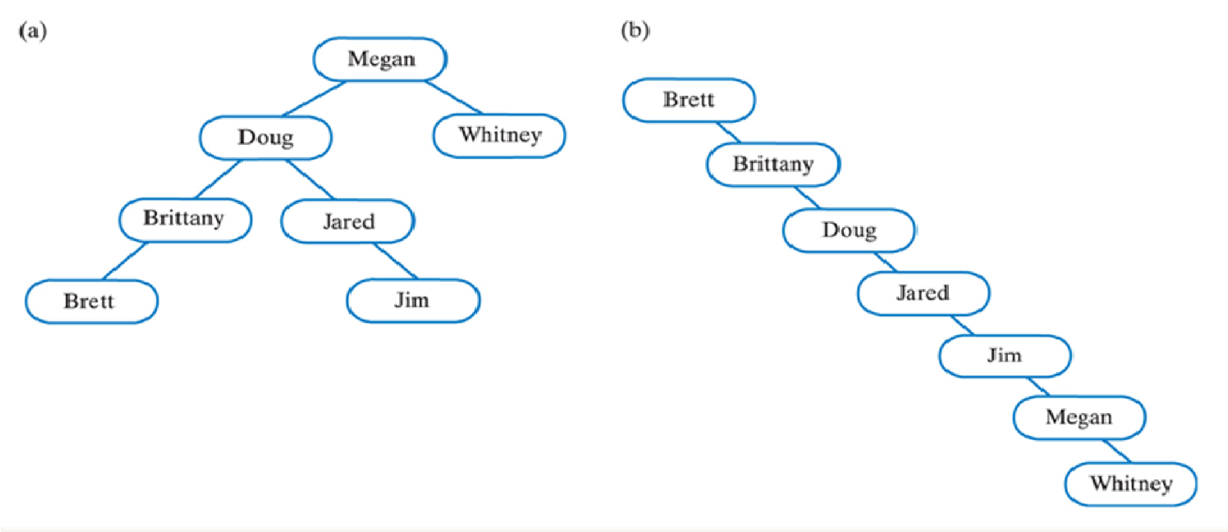
  - Each node has a Comparable object with unique key (or search key)
  - All keys in left subtree < key in the root < All keys in right subtree
  - Both left subtree and right subtree are binary search trees

- Examples :





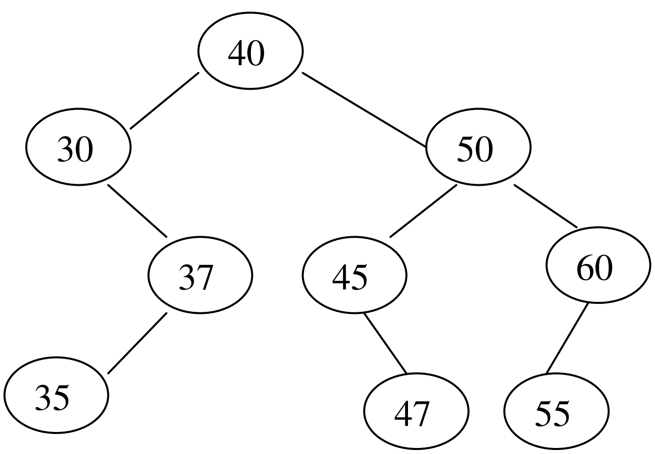
Two binary search trees containing same names:



- Inorder Traversing

  - Theorem : The inorder traversal of binary search tree will visit its nodes in sorted search key order
    Proof : Easy! use induction by the number of nodes

  - Example :



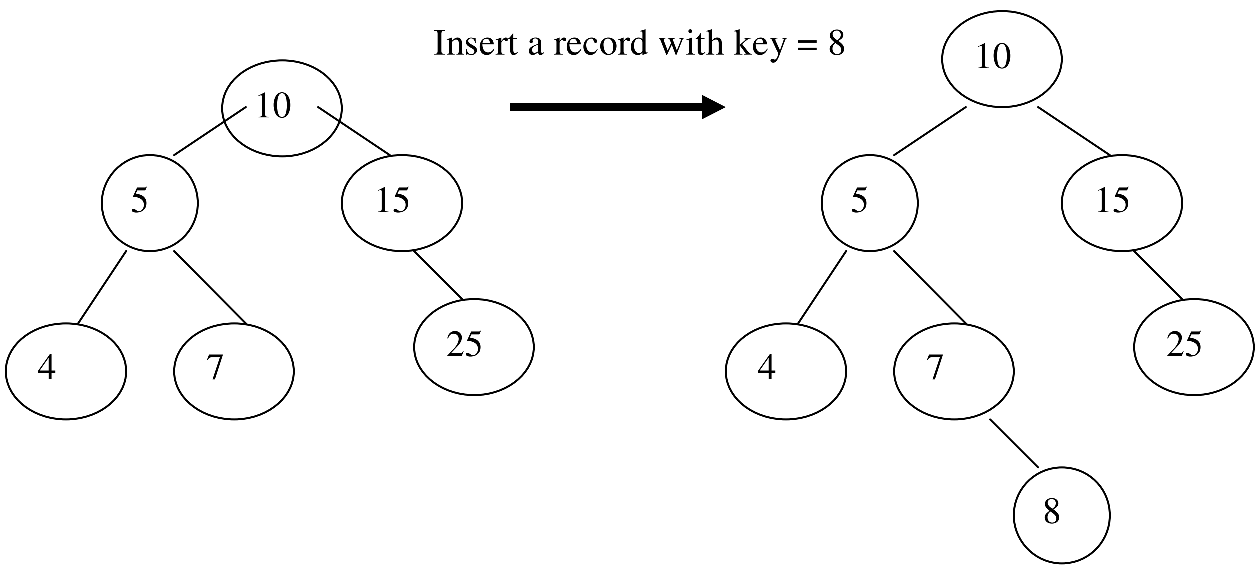
Result :
30 35 37 40 45 47 50 55 60

- **General algorithm to search for an object in BST**

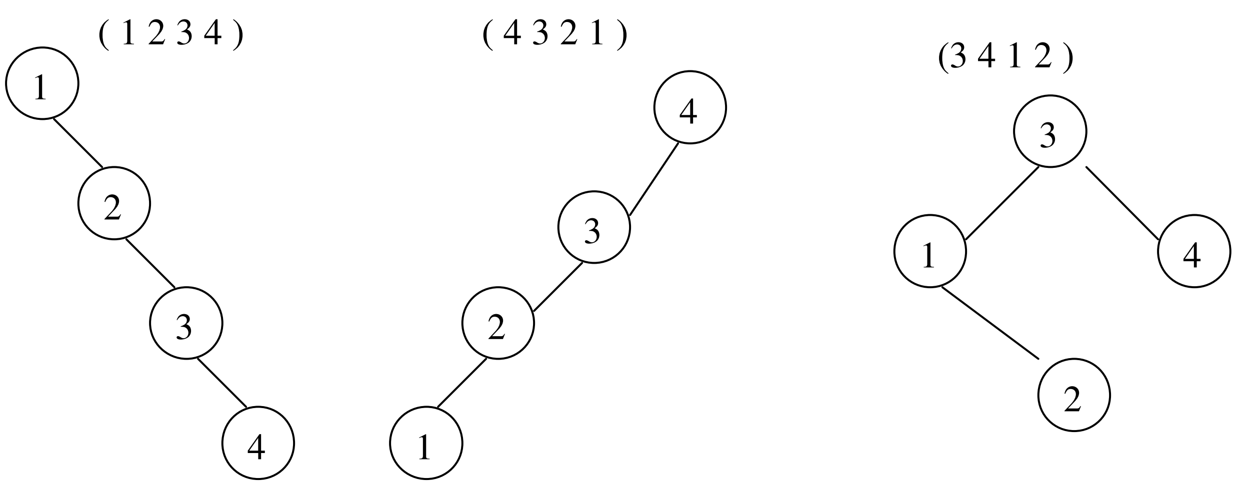
```
Algorithm bstSearch (binarySearchTree, desiredObject)

   // Searches a binary search tree for a given object.
   // Returns true if the object is found.

   if (binarySearchTree is empty)
           return false
   else if (desiredObject == object in the root of binarySearchTree)
           return true
   else if (desiredObject < object in the root of binarySearchTree)
           return bstSearch (left subtree of binarySearchTree,
                                    desiredObject)
   else
    return bstSearch (right subtree of binarySearchTree,
                         desiredObject)
```

- Outline of insertion algorithm : add()

  - Need to make sure that the tree is still a binary search tree after insertion!
  - Starting from root, use search() strategy to locate the correct position
  - If the item already in the tree, the item is not inserted.
  - If the item is not in the tree, search() stops when the nodeRef is NULL. Which is the proper location for the new node. Let nodeRef = new node.

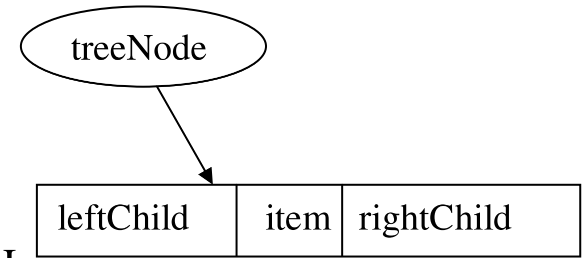


- Example : May get different binary search trees if input orderings of data are different



- Outline of deletion Algorithm: remove

  - Need to make sure that the tree is still a binary search tree after deletion!
  - Step1: locate the desired node.
    Assume treeNode refers to the desired node.



  - Step 2: There are 3 cases for desired node :

    It is a leaf → easy case, set treeNode to NULL
    It has only one child → set treeNode to the valid child node
    It has two children → Step 3:

    - find the location of inorder successor item, i.e. the node with smallest key which is larger than deleted key.
      OR find the location of inorder predecessor item, i.e. the node with largest key which is smaller than deleted key.

    - The successor item is in the left most leaf (node X) of right subtree
      OR the predecessor is in right most leaf of a left subtree

    - copy successor item to the desired node's item
      OR copy predecessor item to the desired node's item

    - Step 4: delete node X (either case I or case II)

Examples are given in class!!

- Efficiency of Operations

  Assume a binary search tree with N nodes, the maximum height is N, and minimum height is $\log_2(N+1)$

  Worst case running time of binary search tree operation, let h = height of binary tree

Operations	Running Time	Worst Case Running Time
Retrieval	O(h)	O(N)
Insertion	O(h)	O(N)
Deletion	O(h)	O(N)
Traversal	O(N)	O(N)

  Note : You will study how to maintain a balanced binary search tree, i.e. minimum height binary search tree (only need to consider insertion and deletion operations) → the height of the tree is O(log N)
  → Worst case running time of above operations (excluding Traversal) are O(log N)

Note: Java class libraries: TreeMap and TreeSet support sorted order traversal. They are using balanced trees

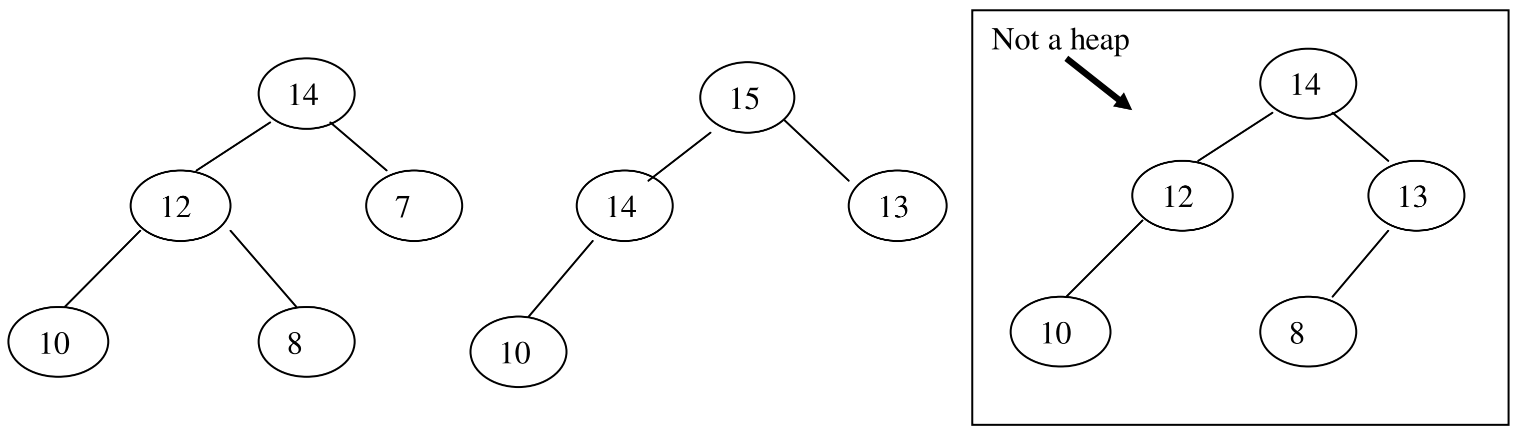
4. Heap

- A complete binary tree
  Nodes contain Comparable objects
  Each node contains no smaller (or no larger) than objects in its descendants

- Maxheap : Object in a node is ≥ its descendant objects

- Minheap : Object in a node is ≤ descendant objects

- Examples : Max heaps



- Array-based implementation

  - Array-based implementation supports complete binary tree.

    Can use level-order traversal to store data in consecutive locations of an array

  - Assume array heap[] contains all items starting at index 1
    For index i, parent index is i/2, children are 2i & 2i+1

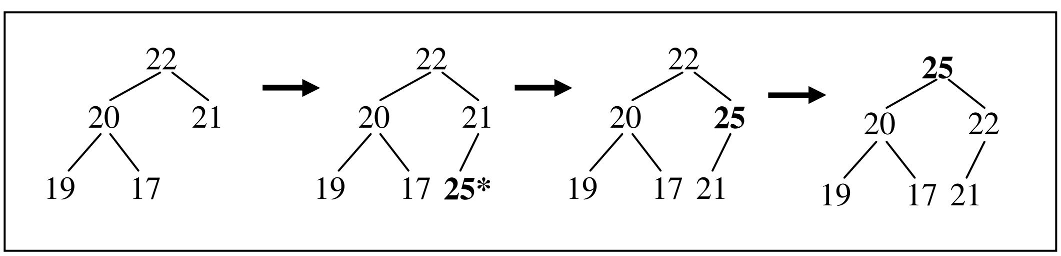
    Note: may also start with index 0

- Outline of Insertion Algorithm

  - insert the item into bottom of the tree, say, heap[size+1]

  - trickle new item up to appropriate spot in the tree by comparing with its parent

  - Example :



Index	Heap	Heap	Heap	Heap	
1	22	22	22	25	
2	20	20	20	20	
3	21	21	25	22	Size = 6
4	19	19	19	19	
5	17	17	17	17	
6		25	21	21	
Size = 5		After step 1	After swap(25,21)	After swap(25,22)	

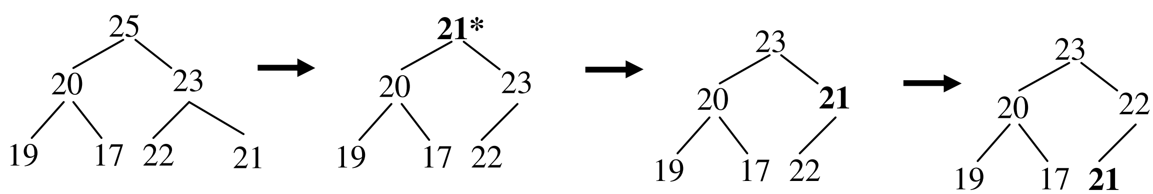
- Outline of Deletion Algorithm : return root item

  - rootItem = heap[1]   // copy 1st item to rootItem

  - heap[1] = heap[size]   // copy the last item into root position

  - At this point, Heap[] is no longer a maxheap.
    - We need to trickle down the item in Heap[1] as much as possible
    - by comparing with its children. Swap the item with its largest child.
    - Stop the process, when the item value is bigger than both children\

Example :



Index	Item	Item	Item	Item	
1	25	21	23	22	
2	20	20	20	20	
3	23	23	21	22	Size = 6
4	19	19	19	19	
5	17	17	17	17	
6	22	22	22	21	
7	21				
size = 7	size=6	move 21 to item[1]	after swap(21,23)	after swap(21,22)	

- Discussion

  - Heap is a binary tree with minimum height (since it is a complete binary tree)

  - To balance the binary search tree (minimum height), the algorithm is quite complex

  - For handling duplicate keys, may use a queue in every node to queue duplicate keys. Delete the node when the last key is remove from the queue

  - Comparison of running time

	Insertion	Deletion	Retrieve (largest key)
Unsorted array based	O(1)	O(N)	O(N)
Unsorted pointer based	O(1)	O(N)	O(N)
Sorted array based	O(N)	O(1)	O(1)
Sorted pointer based	O(N)	O(1)	O(1)
Binary Search Tree(balanced)	O(log N)	O(log N)	O(log N)
Maxheap	O(log N)	O(log N)	O(1)