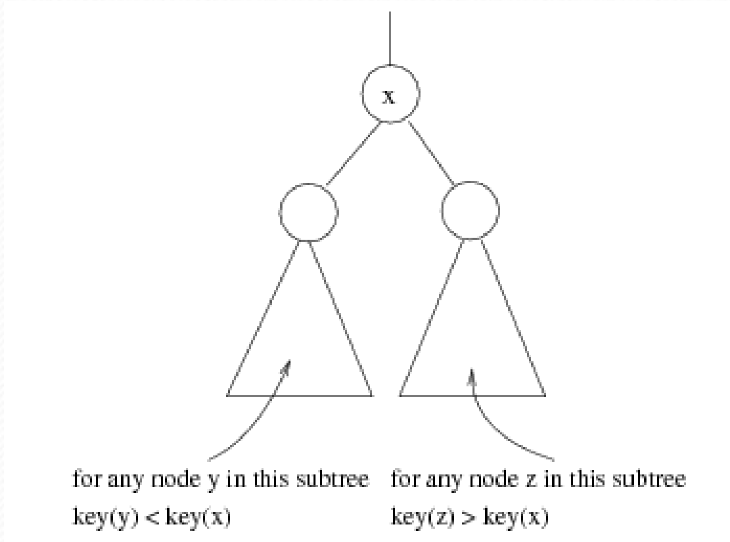
Binary Search Tree

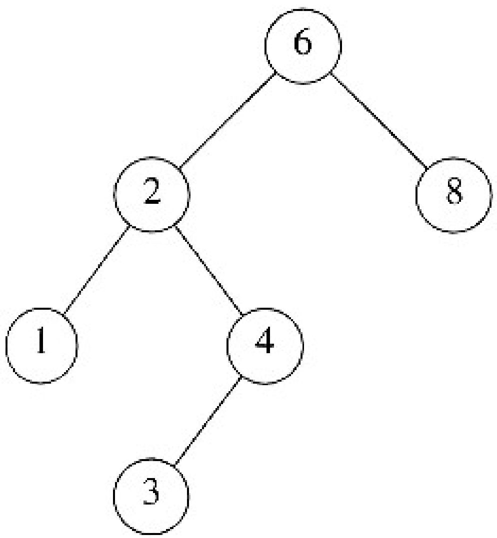
- introduction,
- searching,
- insertion and
- deletion

Binary Search Trees (BST)

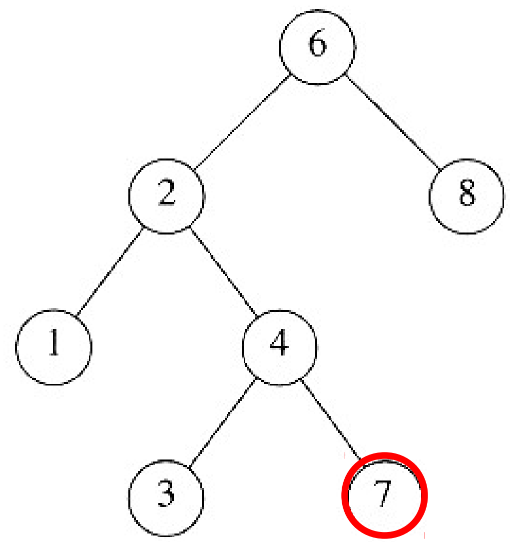
- A data structure for efficient searching, insertion and deletion
- Binary search tree property
  - For every node X
  - All the keys in its left subtree are smaller than the key value in X
  - All the keys in its right subtree are larger than the key value in X



Binary Search Trees



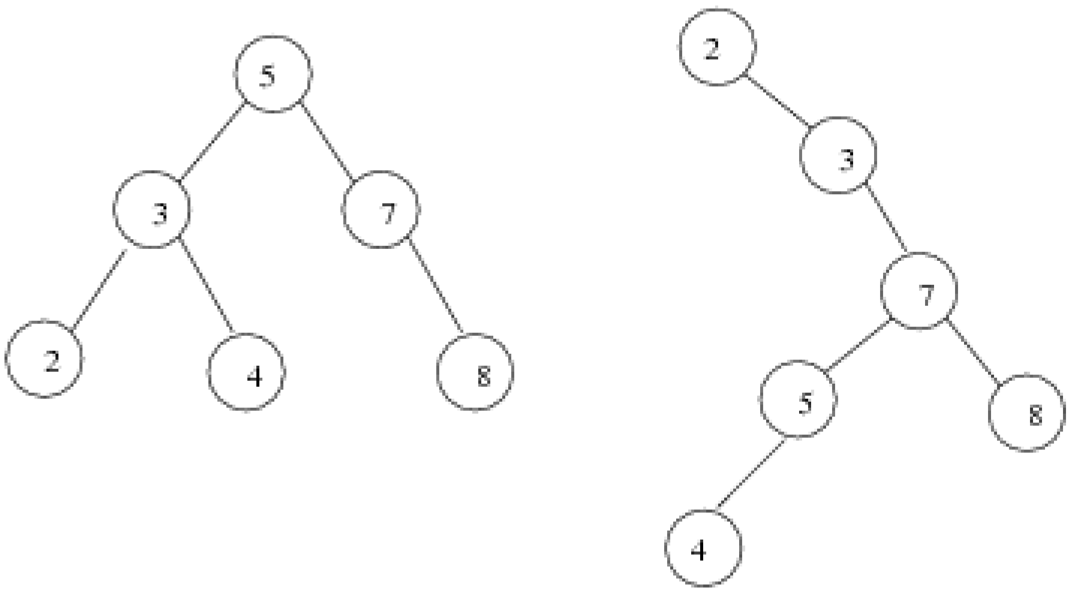
A binary search tree



Not a binary search tree

Binary Search Trees

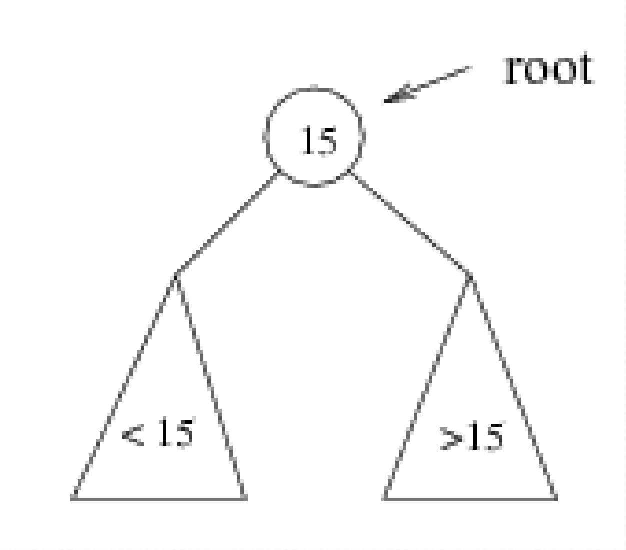
The same set of keys may have different BSTs



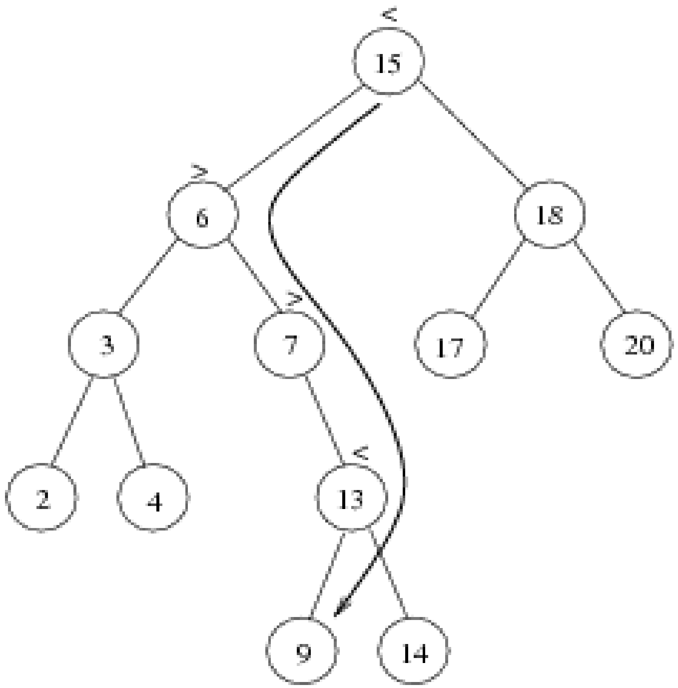
- Average depth of a node is O(log N)
- Maximum depth of a node is O(N)

Searching BST

- If we are searching for root (15), then we are done.
- If we are searching for a key < root , then we should search in the left subtree.
- If we are searching for a key > root, then we should search in the right subtree.



Example: Search for 9 ...



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Searching (Find)

FIND(info, left, right, root, item, loc, par)- finds the item in tree T with root is root and info, left and right is three array represented in memory. This algorithm returns ***loc*** i.e. location of item and ***par*** i.e. parent.

1. [Tree Empty??]

   if *root==NULL*, then set LOC=NULL & PAR=NULL and return.

1. [Item root ??]

   If *item==INFO[ROOT]*, then LOC=ROOT & PAR=NULL and return.

1. [Initialize pointer ptr and save]

   If *item<INFO[ROOT]*

   then set PTR = LEFT[ROOT] and SAVE=ROOT

   Else

   set PTR = RIGHT[ROOT] and SAVE=ROOT

   [End of if]

1. Repeat 5 and 6 while ptr!=NULL
2. [item found??]

   If *ITEM=INFO[PTR],* then set LOC=PTR and PAR=SAVE, and return.

1. If *ITEM<INFO[PTR],* then SAVE=PTR and PTR=LEFT[PTR]

   Else

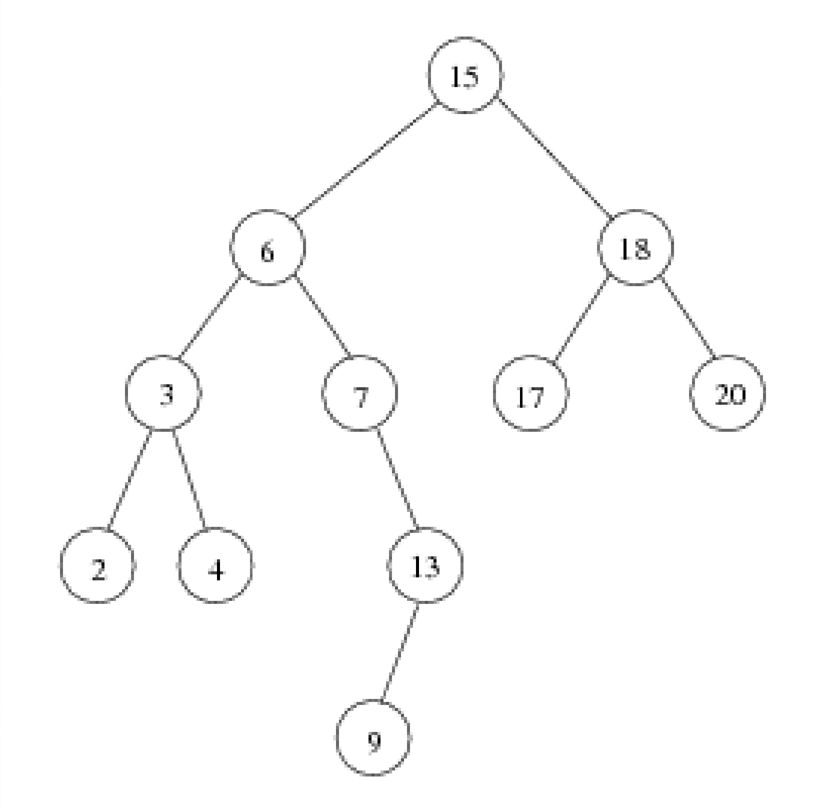
   Set SAVE=PTR and PTR=RIGHT[PTR]

1. [Search unsuccessful] Set, LOC=NULL and PAR = SAVE
2. Exit

- Time complexity: O(height of the tree)

Sorting: Inorder Traversal of BST

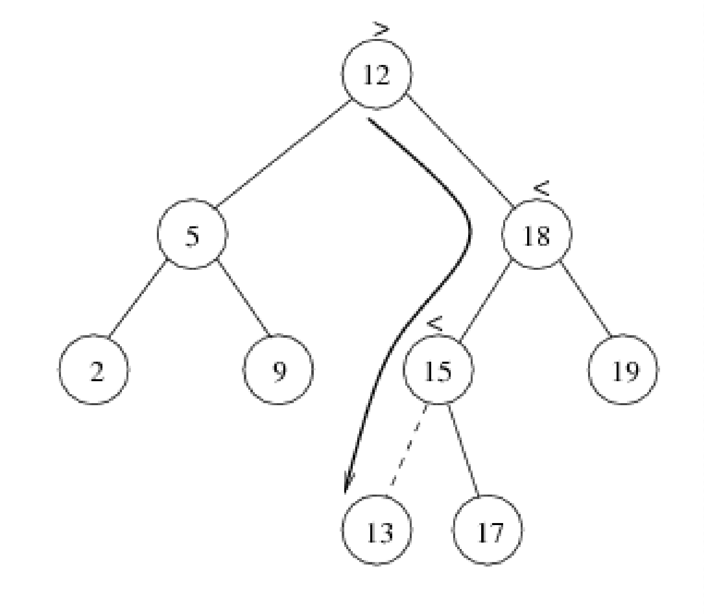
- ***Inorder Traversal*** of BST prints out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Insertion

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed



- Time complexity = O(height of the tree)

Inserting (ADD node)

INSBST(info, left, right, root, item, loc, avail)- insert the item in tree T with root is root and info, left and right is three array represented in memory. This algorithm returns ***loc*** i.e. location of item or ***ADD*** item as new node in tree.

1. Call ***FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)***
2. If *LOC!=NULL*, then Exit.
3. [Copy ITEM into new node in AVAIL list]
   a) If AVAIL==NULL, Print "OVER FLOW";
   b) Set NEW=AVAIL, AVAIL=LEFT[AVAIL] and INFO[NEW]=ITEM.
   c) Set LOC=NEW,LEFT[NEW]=RIGHT[NEW]=NULL
4. [ADD ITEM to TREE]

   If PAR=NULL then, Set ROOT=NEW.

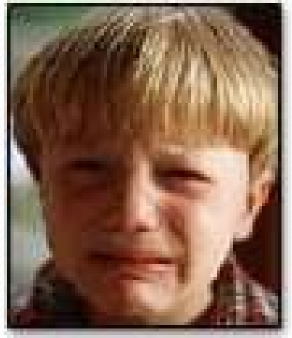
   Else IF ITEM<INFO[PAR] , Set LEFT[PAR]=NEW

   Else Set RIGHT[PAR]=NEW
1. Exit

- Time complexity: O(height of the tree)

Deletion

- When we delete a node, we need to consider how we take care of the children of the deleted node.



- This has to be done such that the property of the search tree is maintained.

Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass that node

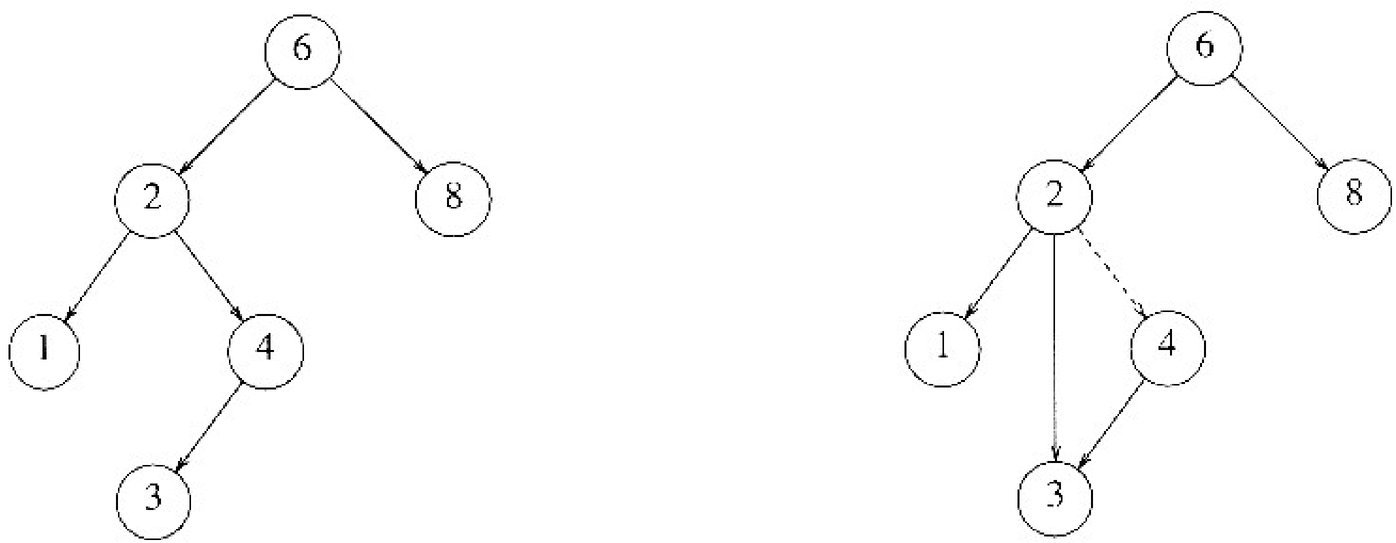


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion Case 3

- Case 3: the node has 2 children
  - Replace the key of that node with the minimum element at the right subtree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

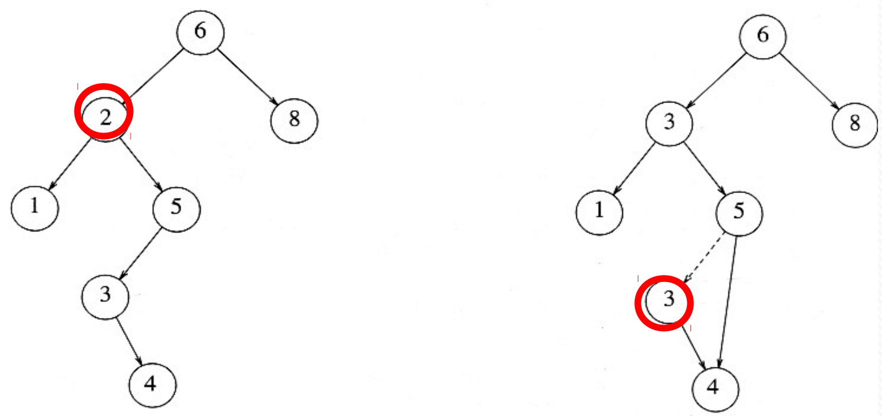


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Deletion Algorithm

- DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)
- A binary search tree T is in memory, and an ITEM of information is given. This algorithm delete ITEM from the tree.

1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. If LOC=NULL, then write ITEM not in tree and Exit
3. If RIGHT[LOC]!=NULL and LEFT[LOC]!=NULL, then:

   Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

   Else:

   Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
4. Set LEFT[LOC]:=AVAIL and AVAIL :=LOC.
5. Exit

CASEA: only one or, no child

- CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)-delete the Node N at location LOC, where N doesn't have two Children. PAR is location of parent node or, PAR=NULL i.e. ROOT node.

1. [initialize CHILD]
   If *LEFT[LOC]=NULL* and RIGHT[LOC]=NULL, then
   CHILD=NULL
   Else if *LEFT[LOC]!=NULL* , then CHILD=LEFT[LOC]
   Else CHILD=RIGHT[LOC]
1. If *PAR != NULL* then: *(i.e. NOT A ROOT NODE)*
   If *LOC=LEFT[PAR]*, then set LEFT[PAR]=CHILD
   Else RIGHT[PAR]=CHILD
   [End of IF]
   Else set ROOT=CHILD.
   [End of IF]
1. Exit

CASEB: has 2 children

- CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)-delete the Node N at location LOC, where N has two Children. PAR is location of parent node or, PAR=NULL i.e. ROOT node. SUC gives location of inorder successor and PARSUC gives location of parent of inorder successor .

1. [Find SUC and PARSUC]
   a) Set PTR=RIGHT[LOC] and SAVE=LOC
   b) Repeat while LEFT[PTR]!=NULL
      Set, SAVE=PTR and PTR=LEFT[PTR]
      [END OF LOOP]
   a) Set SUC=PTR and PARSUC=SAVE.
2. [Delete SUC] Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC,PARSUC)
3. [replace node N by SUC]
   a) If *PAR != NULL* then: *(i.e. NOT A ROOT NODE)*
   If *LOC=LEFT[PAR]*, then set LEFT[PAR]=SUC
   Else RIGHT[PAR]=SUC
   [End of IF]
   Else set ROOT=SUC.
   [End of IF]
   b) Set, LEFT[SUC]=LEFT[LOC] and
   RIGHT[SUC]=RIGHT[LOC]
4. Exit

Thank you !!!