Binary Search Trees
(BST)

CSE 3318 – Algorithms and Data Structures

Alexandra Stefan
Includes materials from Dr. Bob Weems
University of Texas at Arlington

Search Trees

- "search tree" as a term does **<u>NOT</u>** refer to a specific implementation.
- The term refers to a **family of implementations**, that may have **different properties**.
- We will discuss:
  - Binary search trees (BST).
  - 2-3-4 trees (a special type of a B-tree).
  - mention: red-black trees, AVL trees, splay trees, B-trees and other variations.

- All search trees support:
  - *search*, *insert* and *delete* .
  - min, max, successor, predecessor
- Insertions and deletions can differ among trees, and have important implications on overall performance.
- The main goal is to have insertions and deletions that:
  - Are *efficient* (at most logarithmic time).
  - *Leave the tree balanced*, to support efficient search (at most logarithmic time).
  - Preserve the tree properties (restore the tree)

Binary Search Tree (BST)

- Resources:
  - BST in general – CLRS
  - BST in general and solved problems: http://cslibrary.stanford.edu/110/BinaryTrees.html#s
  - leetcode
  - Insertion at root (using insertion at a leaf and rotations)
    - Sedgewick
    - Dr. Bob Weems: Notes 11, parts: '11.D. Rotations' and '11.E. Insertion At Root'
  - Randomizing the tree by inserting at a random position – not covered
    - Sedgewick

Tree Properties - Review

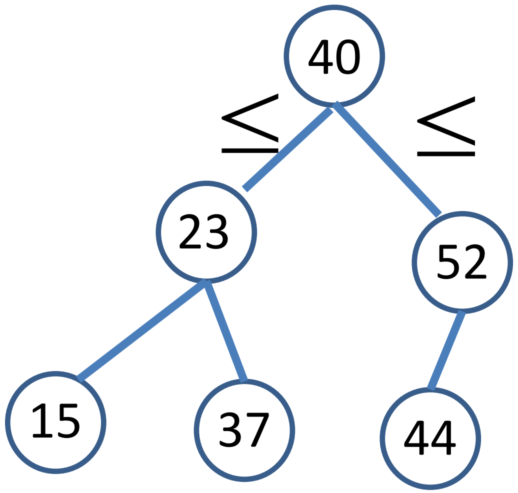
- Full tree
- Nearly Complete tree (e.g. heap tree)
- Complete binary tree

- Tree – connected graph with no cycles, or connected graph with N-1 edges (and N vertices).

Binary Search Trees

- Definition: a binary search tree is a binary tree where the item at each node is:
  - Greater than or equal to all items on the left subtree.
  - Less than or equal to all items in the right subtree.

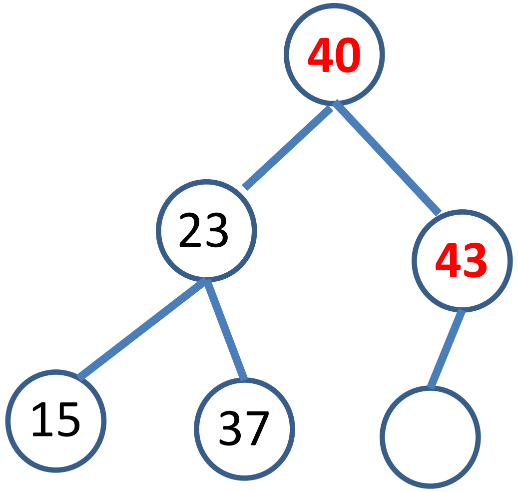
- How do we search?
  - 30? 44?




```
typedef struct TreeNode * TreeNodePT;
struct TreeNode {
  int data;
  TreeNodePT left;
  TreeNodePT right;
};
```

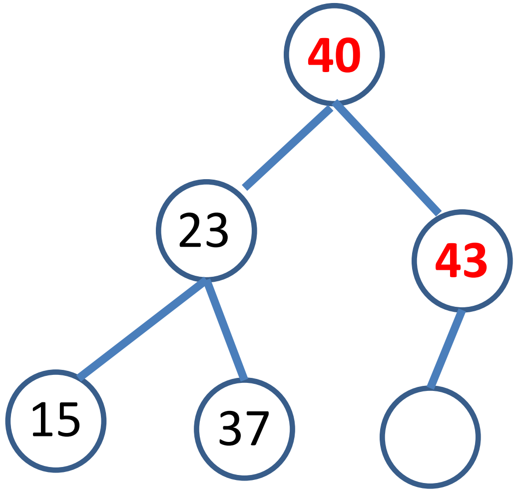
Example 1

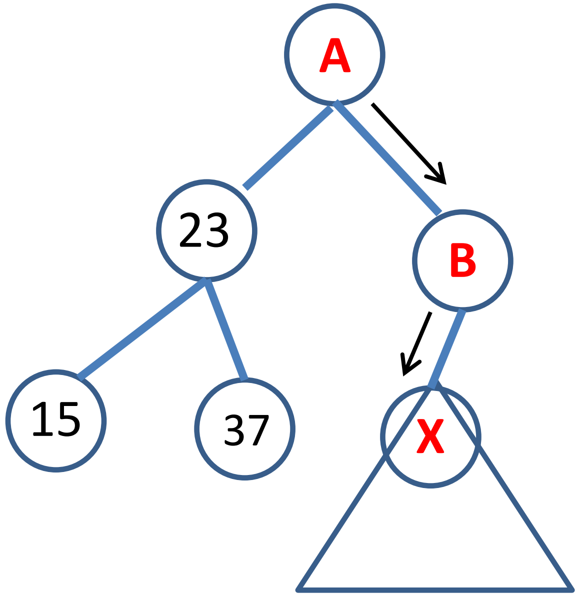
- What values could the empty leaf have?



Example 1

- If you change direction, all the nodes in the subtree rooted X will be in the range [A,B].

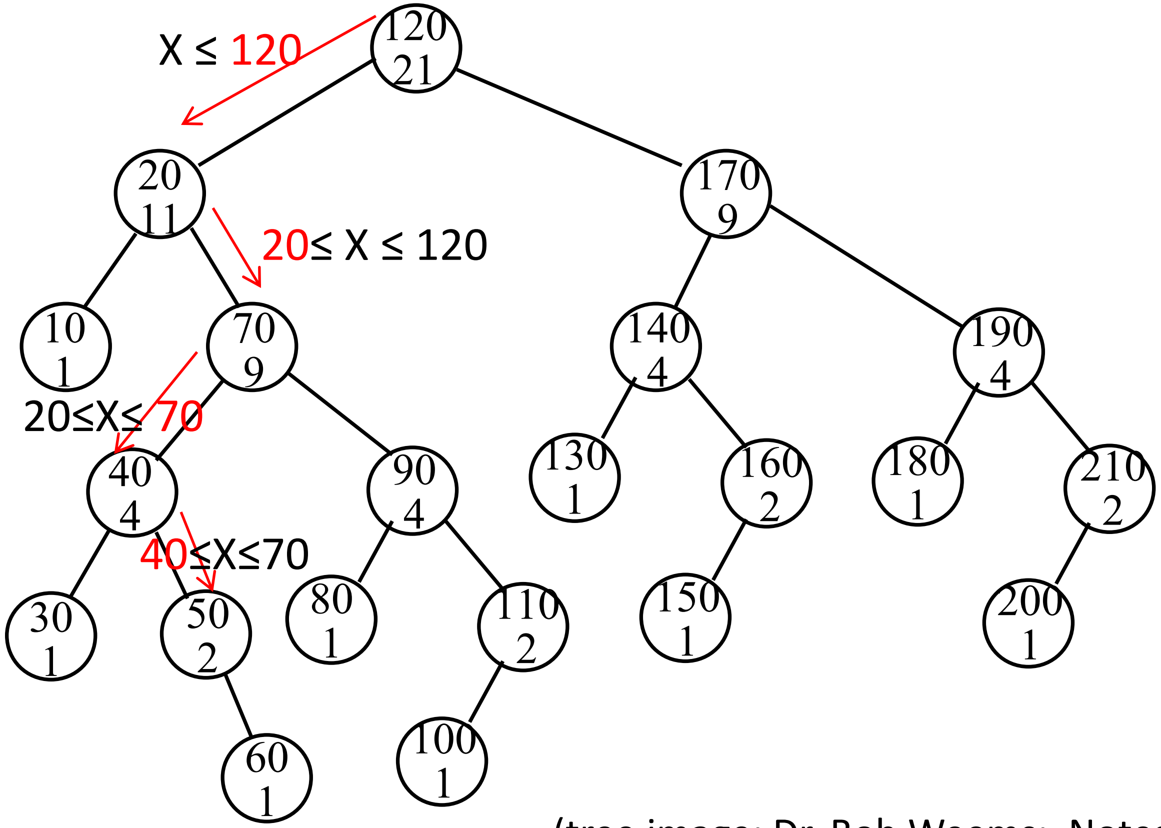




A ≤ X ≤ B

40 ≤ X ≤ 43

Range of possible values



Content of each node:

- the 1st number is the **item/key** and
- the 2nd number is the **tree size** (of subtree rooted here)
- E.g. root has key 120 and size 21 - the tree has 21 nodes

- the path root to node 50 identifies the interval of possible values in the tree rooted at 50 to be: [40,70]

(tree image: Dr. Bob Weems: Notes 11, parts: '11.C. Binary Search Trees')

Valid search path in a BST?

- Assume the **search for 50** gave the sequence:

120, 20, 70, 40, 50 .

Can that be a valid search in a BST?

- Assume the **search for 50** gave the sequence:

120, 20, 70, 80, 50 .

Can that be a valid search in a BST?

- Assume the **search for 50** gave the sequence:

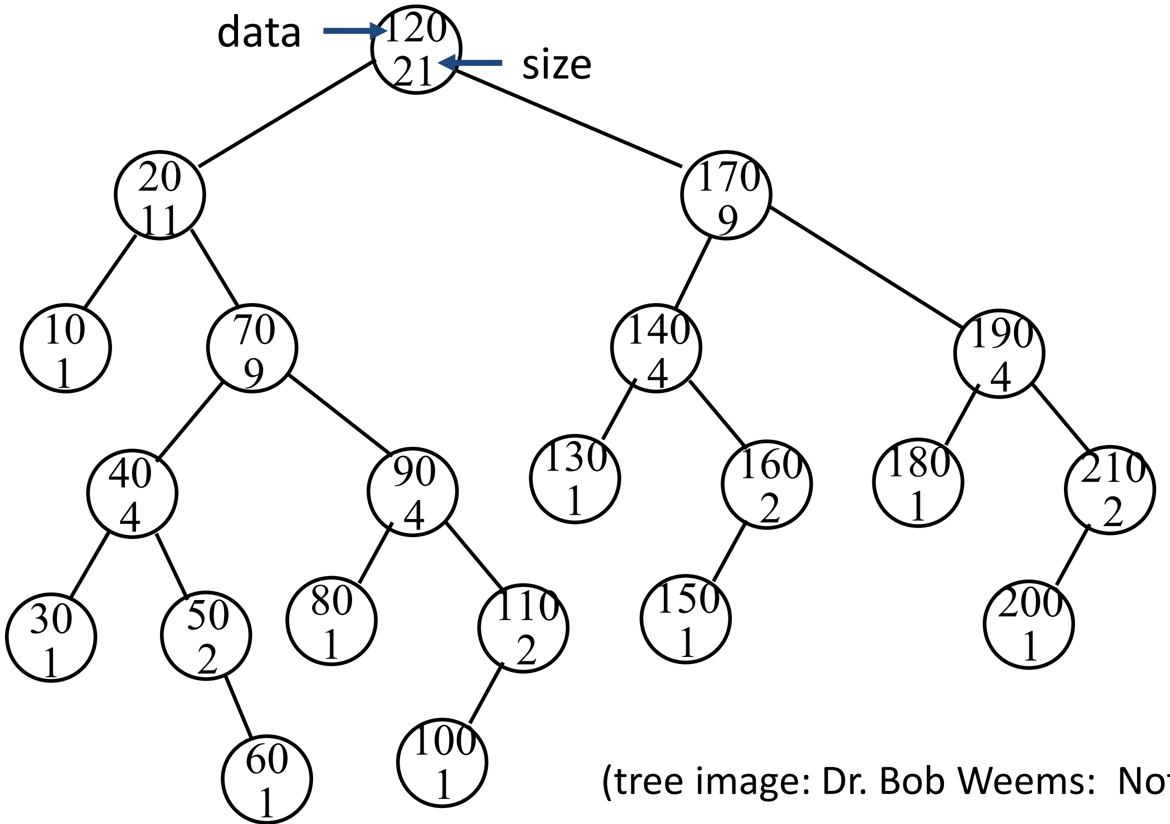
120, 20, 70, 10, 50 .

Can that be a valid search in a BST?

Build the path and check that each node is correct with the tree reconstructed so far: 120 is root, …
If you can solve it on paper, how would you implement it?

Properties

- Where is the item with the **smallest** key?
- Where is the item with the **largest** key?
- What traversal prints the data in **increasing** order?
  – How about **decreasing** order?



Consider the special cases where the root has:

- No left child
- No right child

(tree image: Dr. Bob Weems: Notes 11, parts: ‘11.C. Binary Search Trees’)

Predecessor and Successor
(according to key order)

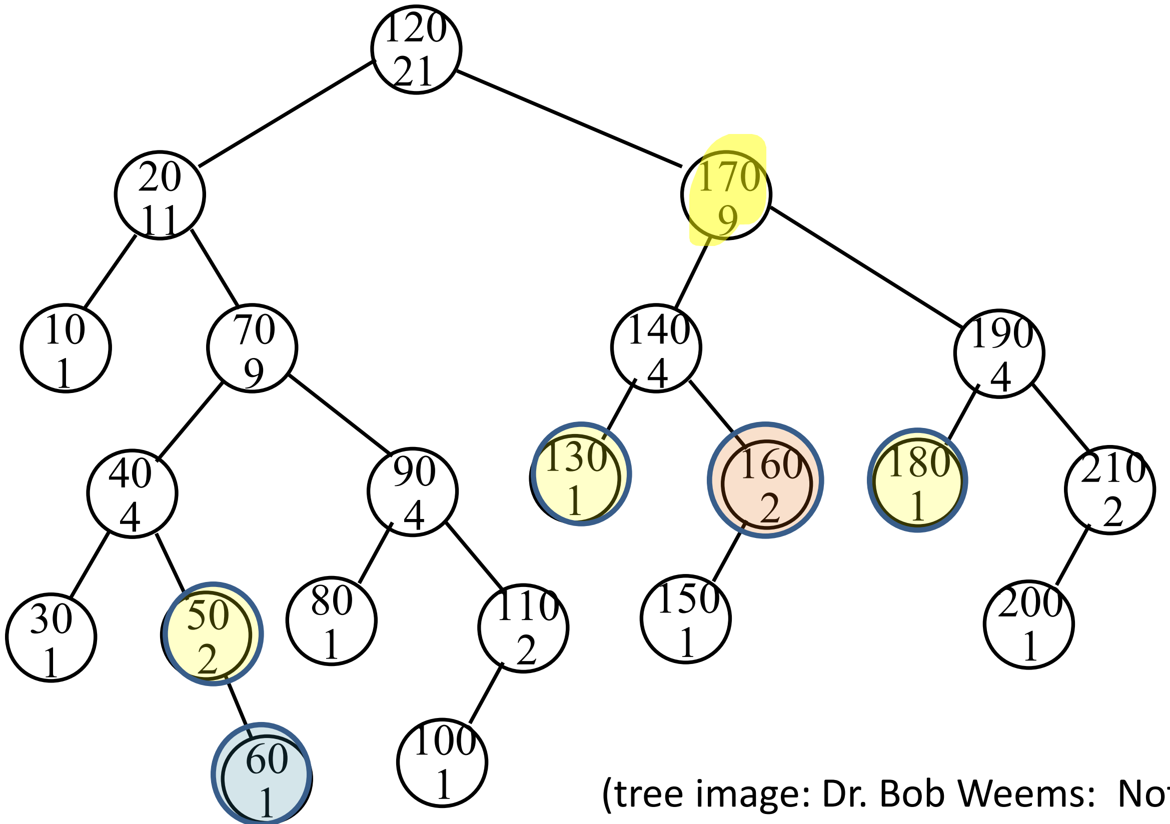
Content of each node:

- the 1st number is the **item/key** and
- the 2nd number is the **tree size** (of subtree rooted here)
- E.g. Root has key 120 and size 21 - the tree has 21 nodes

Predecessor and Successor

- When the node has the child you need.
- When the node does NOT have he child you need.

node keys in sorted order: 10, 20, 30,40,50,60,70,80,90,100,110,120,130,140,150,160,170,180,190,200,210

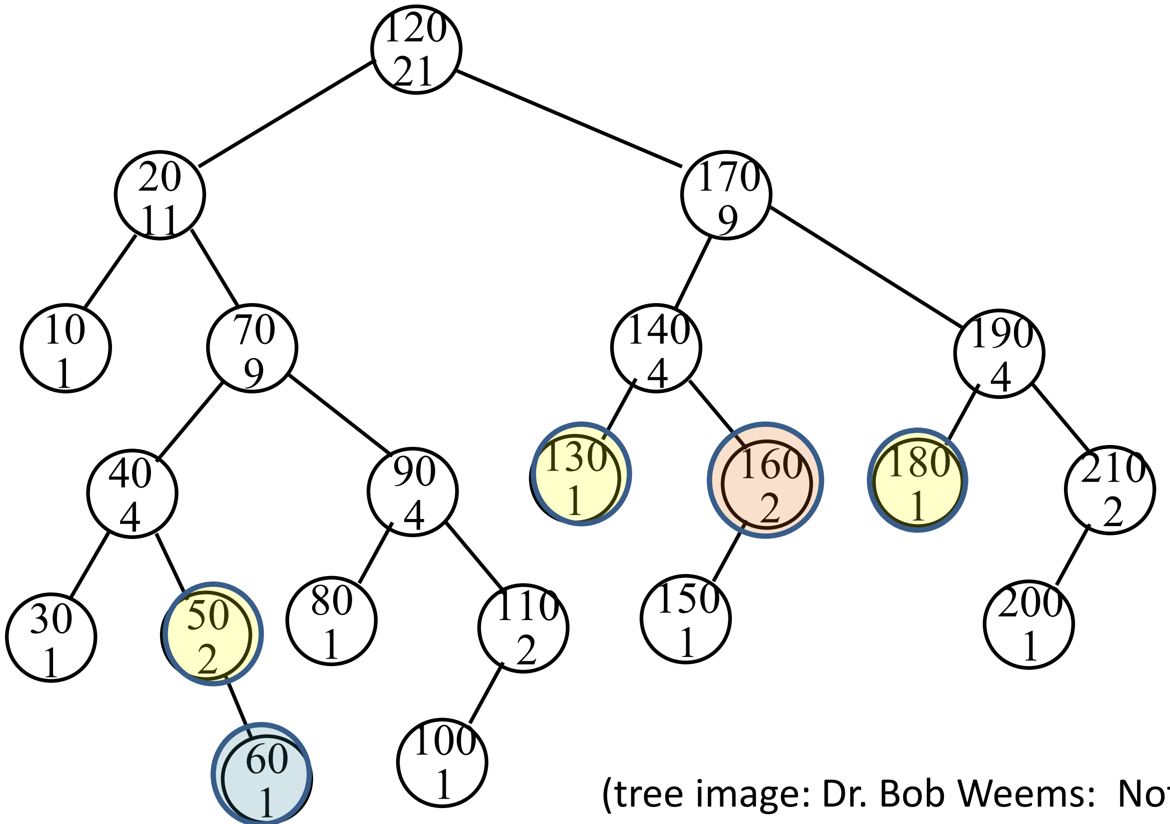


Node	Predecessor	Successor
120		
70		
170		
160		*
60	*	*
130	*	
50	*	
180	*	

(tree image: Dr. Bob Weems: Notes 11, parts: '11.C. Binary Search Trees')

Predecessor and Successor (according to key order)

- Successor of node x with key k (go right):
  - Smallest node in the right subtree
  - Special case: no right subtree: first parent to the right
- Predecessor of node x with key k (go left):
  - Largest node in the left subtree
  - Special case: no left subtree: first parent to the left



Node	Predecessor	Successor
120	110	130
70	60	80
170	160	180
160		***170**
60	* **50**	***70**
130	***120**	
50	***40**	
180	***170**	

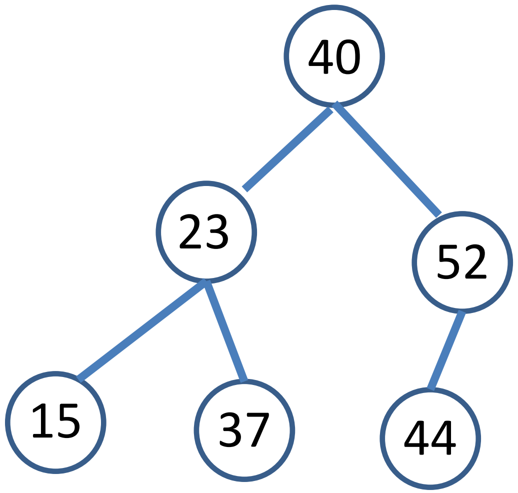
(tree image: Dr. Bob Weems: Notes 11, parts: ‘11.C. Binary Search Trees’)

- Min: leftmost node (from the root keep going left)
  - Special case: no left child => root
- Max: rightmost node (from the root keep going right.
  - Special case: no right child => root
- Print in order:
  - Increasing: Left, Root, Right (inorder traversal)
  - Decreasing: Right, Root, Left
- Successor of node x with key k (go right):
  - Smallest node in the right subtree
  - Special case: no right subtree: first parent to the right
- Predecessor of node x with key k (go left):
  - Largest node in the left subtree
  - Special case: no left subtree: first parent to the left

Binary Search Trees - Search

```
TreeNodePT search(TreeNodePT tree, int s_data) {
    if (tree == NULL) return NULL;
    else if (s_data == tree->data)
        return tree;
    else if (s_data < tree->data)
        return search(tree->left, s_data);
    else return search(tree->right, s_data);
}
```

```
typedef struct TreeNode * TreeNodePT;
struct TreeNode {
  int data;
  TreeNodePT left;
  TreeNodePT right;
};
```



Runtime

(in terms of ,N, number of nodes in the tree or tree height)

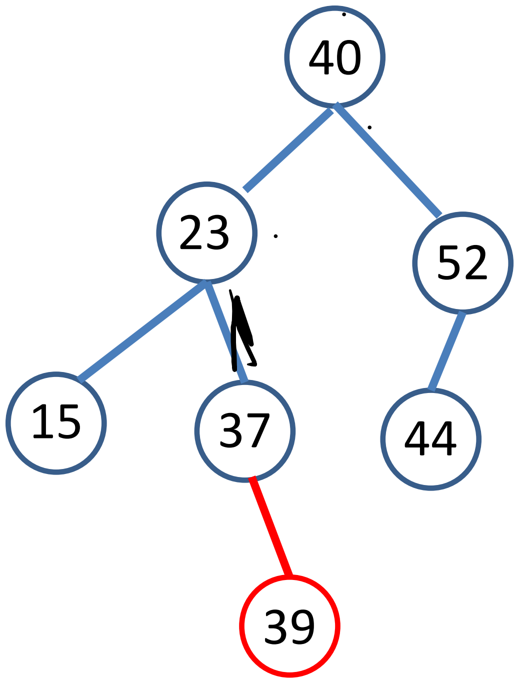
- Best case:
- Worst case:

Naïve Insertion

```
TreeNodePT new_tree_node(int data_in) {
   TreeNodePT ndp = malloc(sizeof(struct TreeNode));
   ndp->data = data_in;
   ndp->left = NULL;
   ndp->right = NULL;
   return ndp;
}
```

To insert an item, the simplest approach is to travel down in the tree until finding a leaf position where it is appropriate to insert the item.

```
/* Assume new_tree(int N) allocates memory for a node struct,
copies N in data, sets left and right to NULL and returns the
address of this node. */
TreeNodePT insert(TreeNodePT h, int n_data)
    if (h == NULL) return new_tree_node(n_data);
    else if (n_data < h->data)
        h->left = insert(h->left, n_data);
    else if (n_data > h->data)
        h->right = insert(h->right, n_data);
    return h;
```



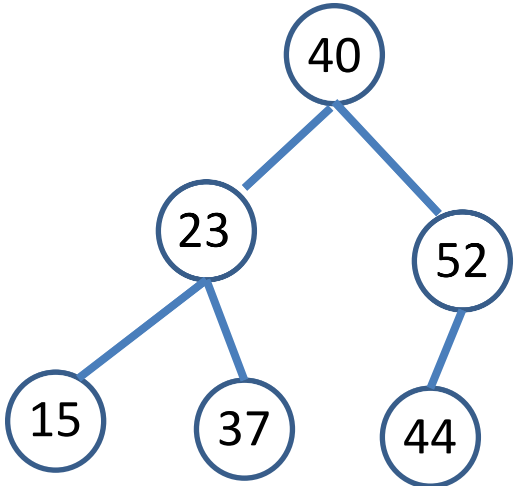
How will we call this method?
root = insert(root, data)

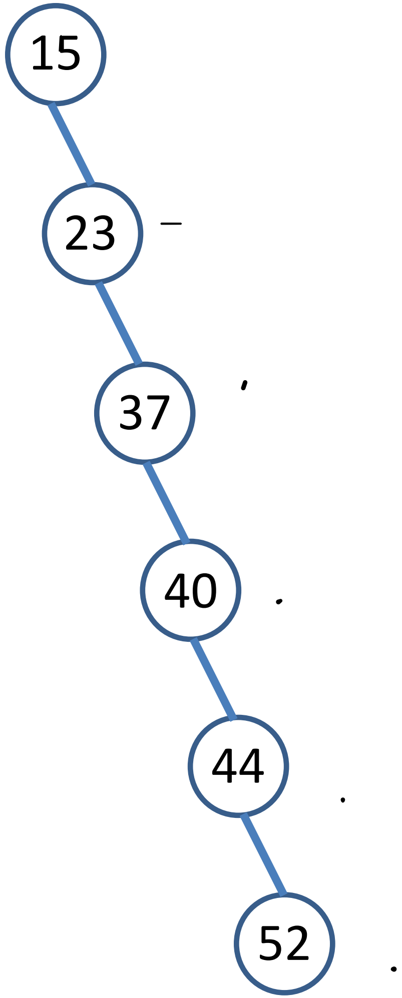
Note that we use:
h->left = insert(h->left, data)
to handle the base case, where we return a new node, and the parent must make this new node a child.

Performance of BST

- Are these trees valid BST?

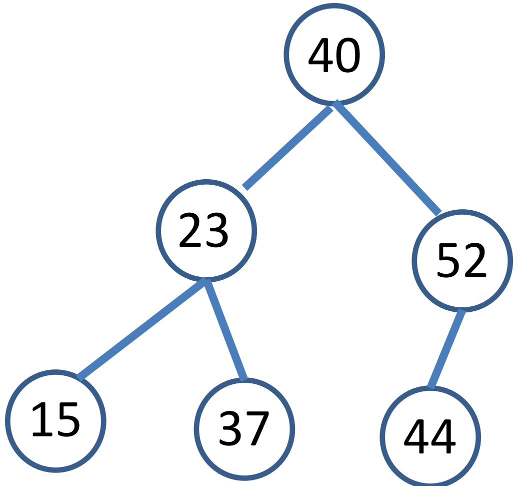
- Give two sequences of nodes s.t. when inserted in an empty tree will produce the two trees shown here (each sequence produces a different tree).





Performance of BST

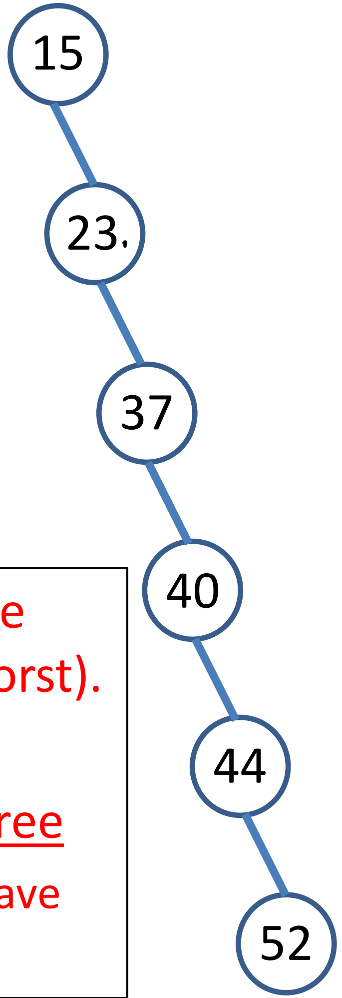
- Are these trees valid BST?
  - Yes
- Give two sequences of nodes s.t. when inserted in an empty tree will produce the two trees shown here (each sequence produces a different tree).
  - **40, 23, 37, 52, 44, 15**
  - **15, 23, 37, 40, 44, 52**



Search, Insert and Delete take time <u>linear to the height</u> of the tree (worst).

Ideal: <u>build and keep a balanced tree</u>

- insertions and deletions should leave the tree balanced.



Performance of BST

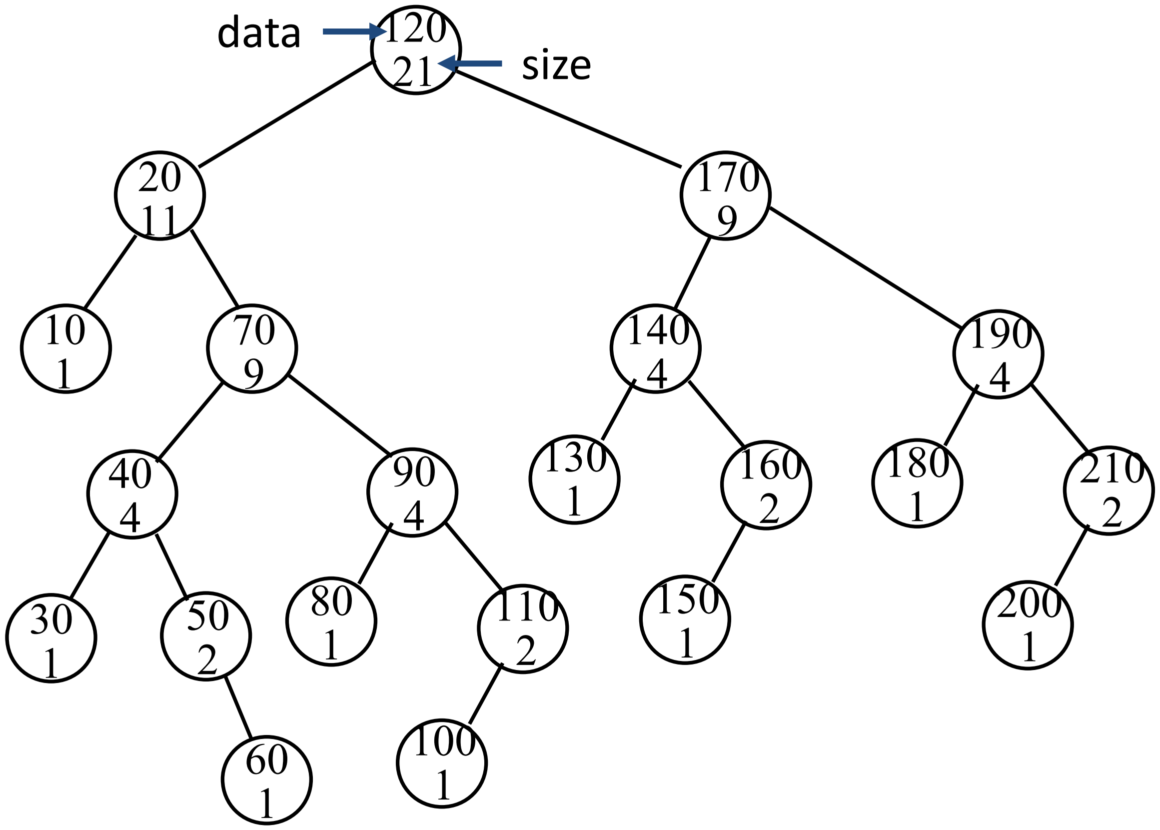
- If items are inserted in:
  - ascending order, the resulting tree is maximally imbalanced.
  - random order, the resulting trees are reasonably balanced.

- Can we insert the items in random order?
  - If we build the tree from a batch of items.
    - Shuffle them first, or grab them from random positions.
  - If they come online (we do not have them all as a batch).
    - Insert in the tree at a random position – see Sedgewick textbook

- Handling duplicates to balance the tree
  - alternate between inserting left and right. Use a flag.
  - keep a list of nodes with equal keys
  - randomly chose to insert left or right

Time complexity for a tree with N nodes

- Min: leftmost node (from the root keep going left)
  - O(___)
- Max: rightmost node (from the root keep going right).
  - O(___)
- Print in order:
  - Increasing: Left, Root, Right (inorder traversal)
  - Decreasing: Right, Root, Left
  - O(______) can we give Theta? Θ(_____)
- Successor of node x with key k (go right):
  - O(___)
- Predecessor of node x with key k (go left):
  - O(___)
- Search for a value (and not found)
  - O(___)
- Build the tree via N repeated insertions:
  - O(___) Best: Θ(_____) Worst: Θ(_____)
- How about space complexity?



BST - Deletion

Delete a node, ***z***, in a BST

1. If z is a leaf, delete it,
2. If z has only one child, replace z with the child
3. If z has 2 children, *replace* it with its order-wise successor, **y**, and delete old **y**. (Note: y will be a leaf or have only one child.)
   A. Method 1 (Simple: ***copy*** the data)
      1. Copy the data from y to z
      2. Delete node **y**.
      3. Problem if other components of the program maintain pointers to nodes in the tree they would not know that the tree was changed and their data cannot be trusted anymore.

   B. Method 2 (***move*** the nodes)
      1. Replaces the **node (not content)** z with node y in the tree.
      2. Delete node **z** (y is now linked in place of z)
      3. Does not have the pointer referencing problem.
      4. 2 implementations: Sedgewick and CLRS.

BST – Deletion – Method 1 (Copy the data)

Delete(d) - delete a node *d* in a BST - Method 1.

1. If d is a leaf, delete it
2. If d has only one child, delete it and readjust the links (the child ‘moves’ in the place of d).
3. If d has 2 children:
   a) Find the successor, s, of d.
      1. Where is the successor of d?
   b) Copy only the data from s to d
   c) Call <u>Delete(s)</u> for node s. Note that s can only be:
      1. Leaf (case 1 above)
      2. A node with only one child (the right child) (This is case 2 above.)

BST – Deletion – Method 2 (Move nodes)

Delete a node *d* in a BST - Method 2.

1. If d is a leaf, delete it
2. If d has only one child, delete it and readjust the links (the child 'moves' in the place of d).
3. If d has 2 children, find the successor, s, of d.

   Is s the right child of d?

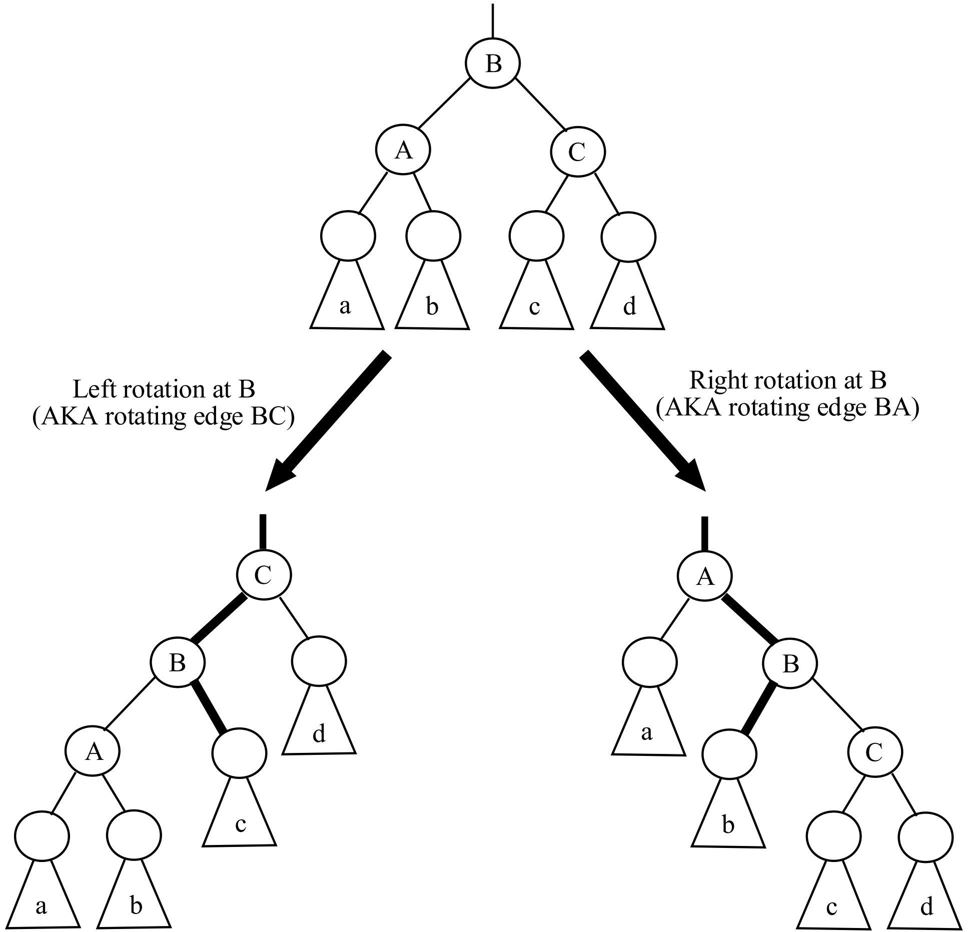
   a) YES: Transplant s over d (s will have only the right child)
   b) NO:

Draw image

BST - Rotations

- Left and right rotations

  (image source: Dr. Bob Weems: Notes 11, parts: ‘11.D. Rotations’)




```
// Sedgewick code:

// rotate to the right
TreeNodePT rotR(TreeNodePT B)
  { nodePT A = B->left;
    B->left = A->right;
    A->right = B;
    return A; }

// rotate to the left
TreeNodePT rotL(TreeNodePT B)
  { nodePT C = B->right;
    B->right = C->left;
    C->left = B;
    return C; }
```

BST – Insertion at Root

(small changes to Sedgewick code)

```
TreeNodePT rotR(TreeNodePT h)
  { nodePT x = h->left; h->left = x->right; x->right = h;
    return x; }
nodePT rotL(nodePT h)
  { nodePT x = h->right; h->right = x->left; x->left = h;
    return x; }
-----
TreeNodePT insertT(TreeNodePT h, int data)
  { if (h == NULL) return new_tree_node(data, NULL, NULL, 1);
    if (data < h->data)
      { h->left = insertT(h->left, data); h = rotR(h); }
    else
      { h->right = insertT(h->right, data); h = rotL(h); }
    return h;
  }
void STinsert(int data)
  { head = insertT(head, data); } // Sedgewick code adaptation
```