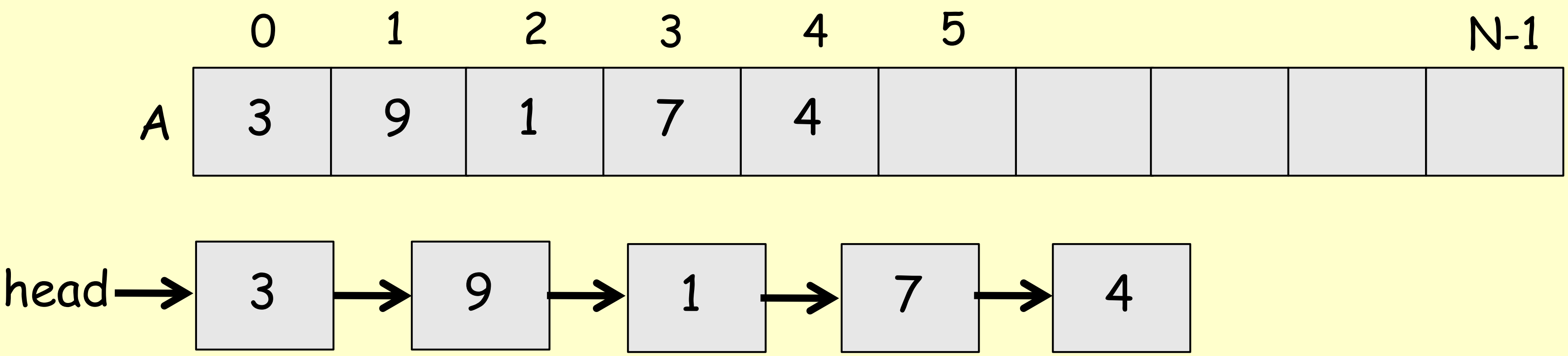
Search Trees - Motivation

- Assume you would like to store several (key, value) pairs in a data structure that would support the following operations efficiently
  - Insert(key, value)
  - Delete(key, value)
  - Find(key)
  - Min()
  - Max()

- What are your alternatives?
  - Use an Array
  - Use a Linked List

Search Trees - Motivation

Example: Store the following keys: 3, 9, 1, 7, 4



Operation	Unsorted Array	Sorted Array	Unsorted List	Sorted List
Find (Search)	O(N)	O(logN)	O(N)	O(N)
Insert	O(1)	O(N)	O(1)	O(N)
Delete	O(N)	O(N)	O(N)	O(N)

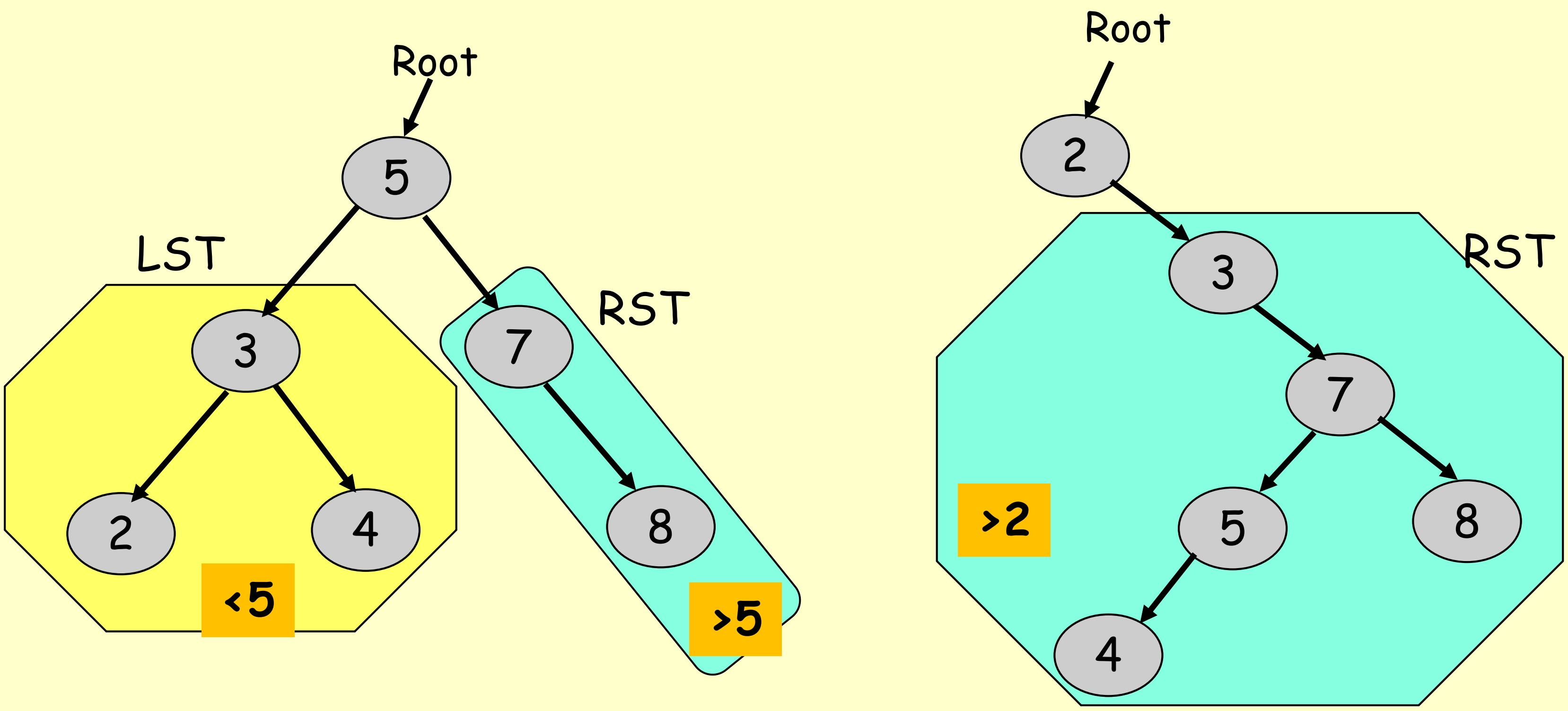
Can we make Find/Insert/Delete all O(logN)?

Search Trees for Efficient Search

- Idea: Organize the data in a search tree structure that supports efficient search operation
    1. Binary search tree (BST)
    2. AVL Tree
    3. Splay Tree
    4. Red-Black Tree
    5. B Tree and B+ Tree

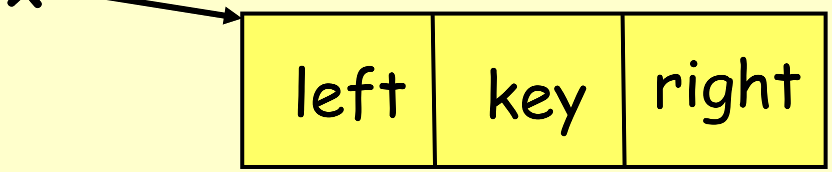
Binary Search Trees

- A Binary Search Tree (BST) is a binary tree in which **the value in every node is**:
  - > all values in the **node's** left **subtree**
  - < all values in the **node's** right **subtree**



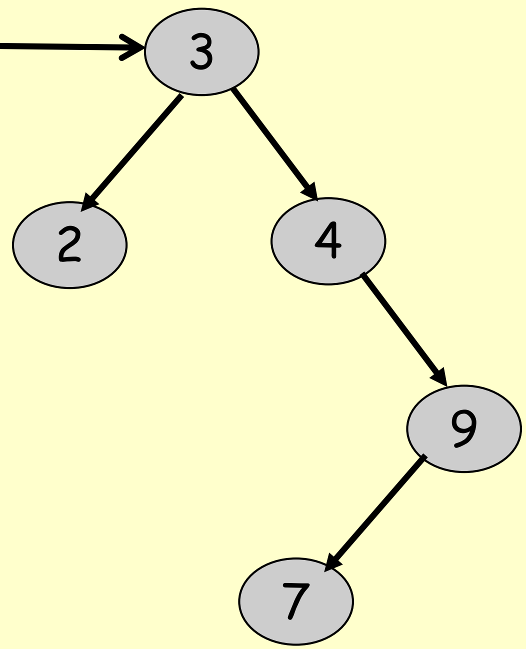
BST ADT Declarations

BST Node



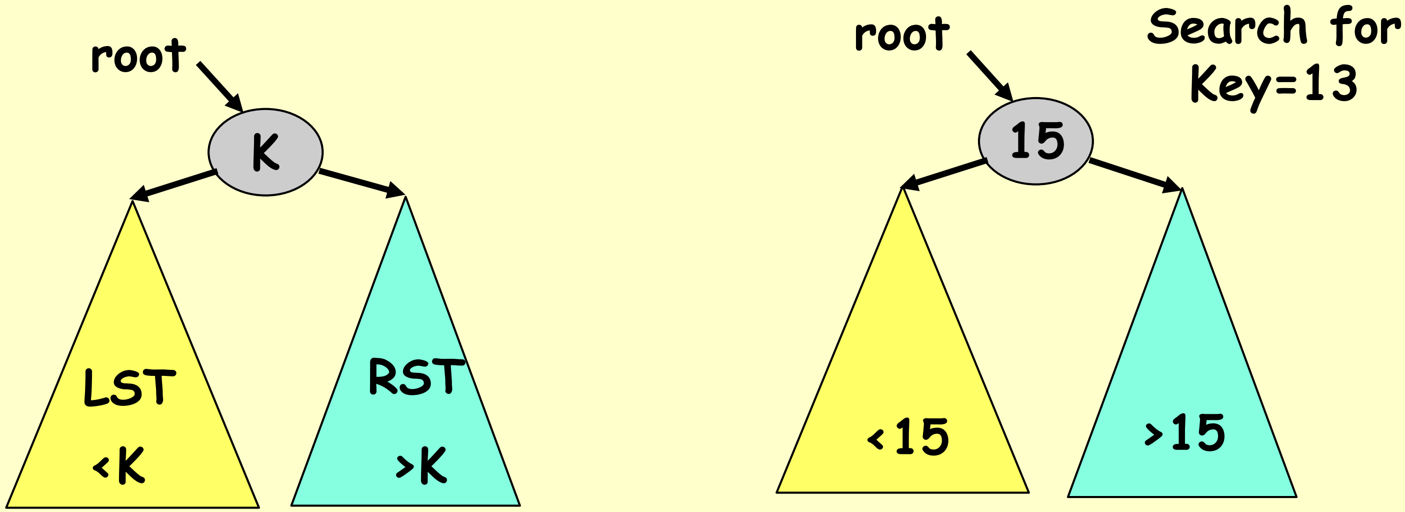

```
class BSTNode {
     BSTNode left;
     int key;
     BSTNode right;
}
```

```
/* BST ADT */
class BST {
private:
  BSTNode root;

public:
  BST(){root=null;}
  void Insert(int key);
  void Delete(int key);
  BSTNode Find(int key);
  BSTNode Min();
  BSTNode Max();
};
```



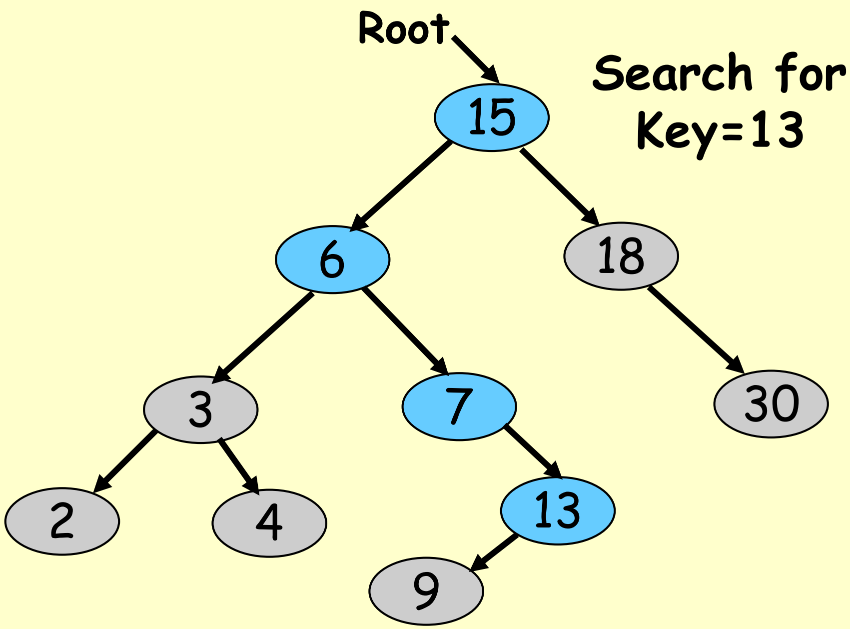
BST Operations - Find

- Find the node containing the key and return a pointer to this node




```
1.  Start at the root
2.  If (key == root.key) return root;
3.  If (key < root.key) Search LST
4.  Otherwise           Search RST
```

BST Operations - Find



Search for Key=13

```
BSTNode Find(int key){
  return DoFind(root, key);
} //end-Find
```

```
BSTNode DoFind(BSTNode root,
               int key){
  if (root == null) return null;
  if (key == root.key)
    return root;
  else if (key < root.key)
    return DoFind(root.left, key);
  else /* key > root.key */
    return DoFind(root.right, key);
} //end-DoFind
```

- Nodes visited during a search for 13 are colored with “blue”
- Notice that the running time of the algorithm is O(d), where d is the depth of the tree

Iterative BST Find

- The same algorithm can be written iteratively by "unrolling" the recursion into a while loop

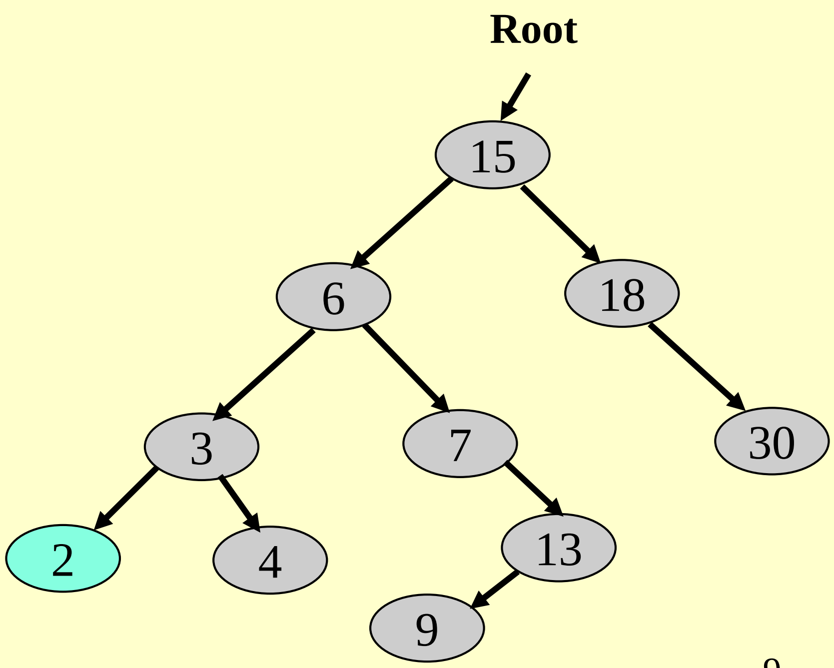
```
BSTNode Find(int key){
  BSTNode p = root;

  while (p != null){
    if (key == p.key)       return p;
    else if (key < p.key)  p = p.left;
    else /* key > p.key */ p = p.right;
  } /* end-while */
  return null;
} //end-Find
```

- Iterative version is more efficient than the recursive version

BST Operations - Min

- Returns a pointer to the node that contains the minimum element in the tree
  - Notice that the node with the minimum element can be found by following left child pointers from the root until a NULL is encountered

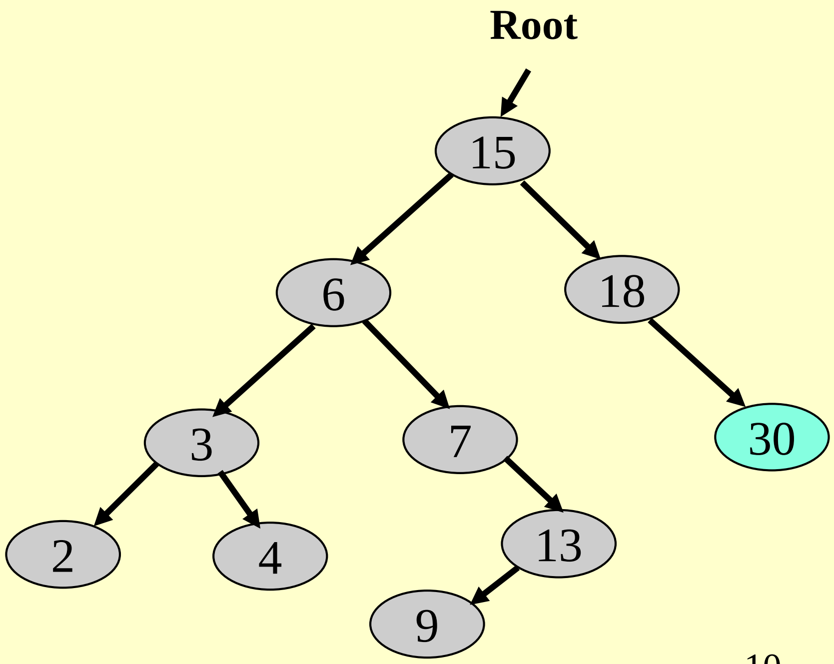
```
BSTNode Min(){
  if (root == null)
    return null;

  BSTNode p = root;
  while (p.left != null){
    p = p.left;
  } //end-while

  return p;
} //end-Min
```



BST Operations - Max

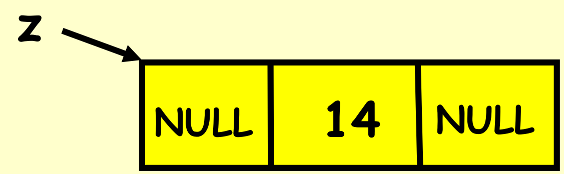
- Returns a pointer to the node that contains the maximum element in the tree
  - Notice that the node with the maximum element can be found by following right child pointers from the root until a NULL is encountered

```
BSTNode Max(){
  if (root == null)
    return null;

  BSTNode p = root;
  while (p.right != null){
    p = p.right;
  } //end-while

  return p;
} //end-Max
```



BST Operations - Insert(int key)

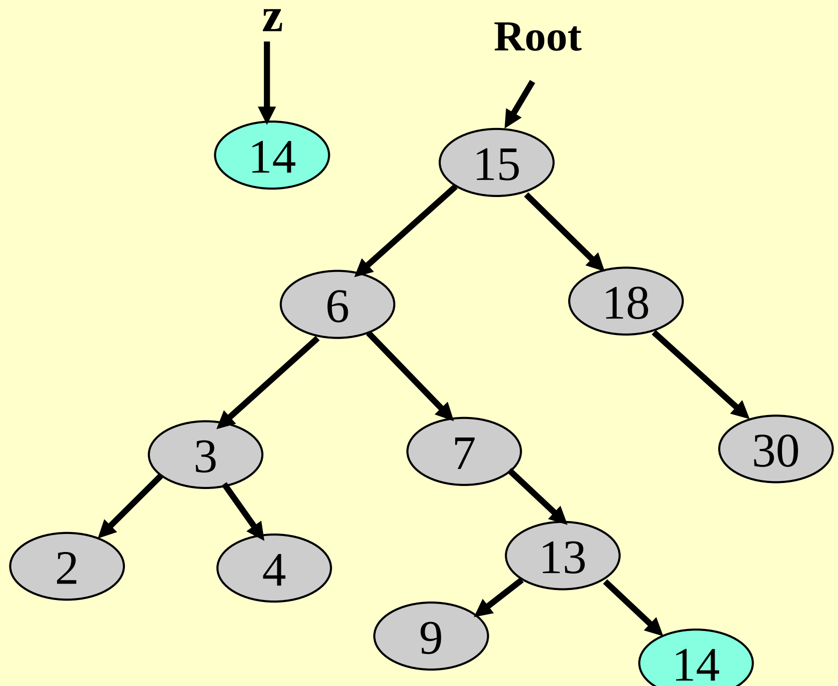
- Create a new node "z" and initialize it with the key to insert
  - E.g.: Insert 14



Node "z" to be inserted
z->key = 14

- Then, begin at the root and trace a path down the tree as if we are searching for the node that contains the key

- The new node must be a child of the node where we stop the search



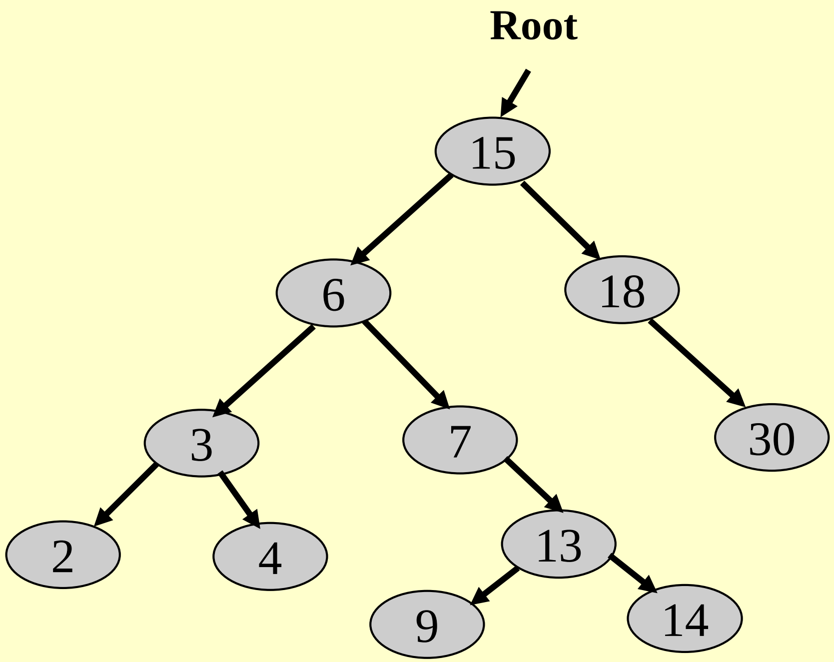
After Insertion

BST Operations – Insert(int key)

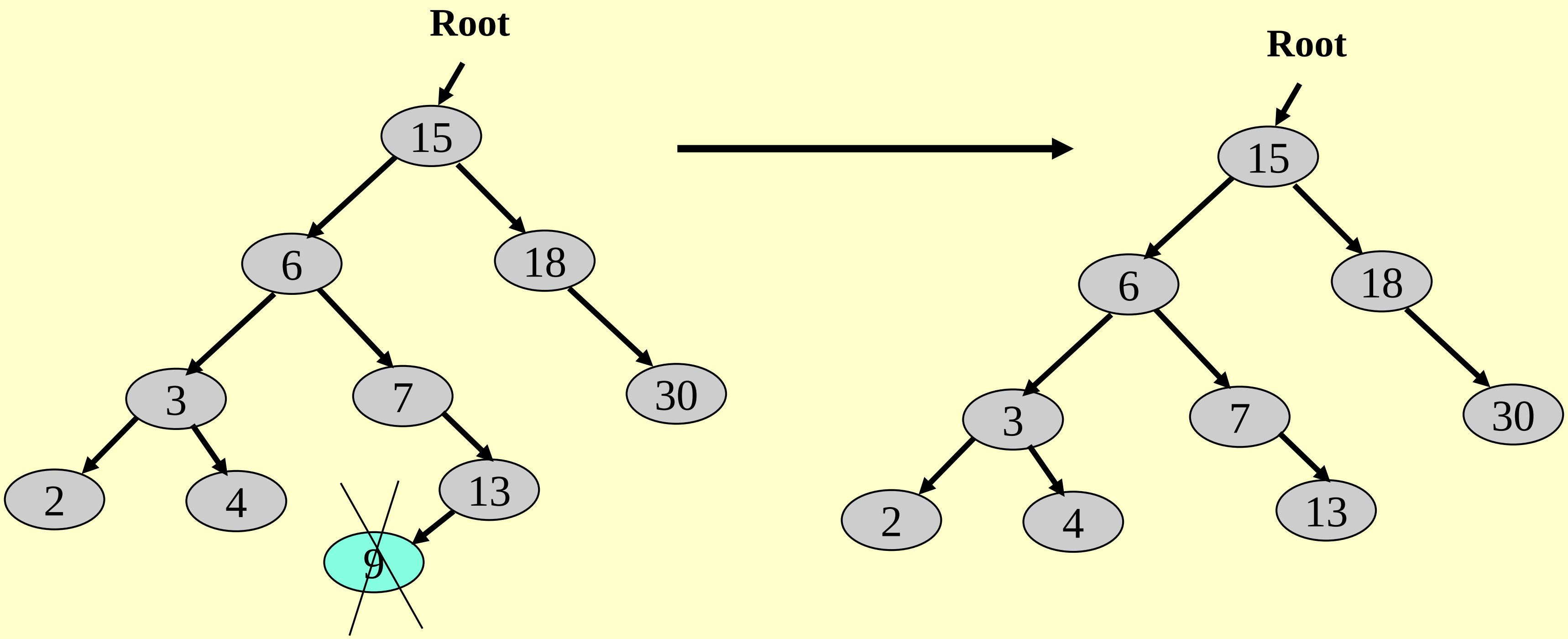
```
void Insert(int key){
  BSTNode pp = null;  /* pp is the parent of p */
  BSTNode p = root;   /* Start at the root and go down */
  while (p != null){
    pp = p;
    if (key == p.key)       return;  /* Already exists */
    else if (key < p.key)  p = p.left;
    else /* key > p.key */ p = p.right;
  } /* end-while */

  BSTNode z = new BSTNode(); /* New node to store the key */
  z.key = key; z.left = z.right = null;

  if (root == null) root = z; /* Inserting into empty tree */
  else if (key < pp.key) pp.left = z;
  else                   pp.right = z;
} //end-Insert
```

BST Operations – Delete(int key)

- Delete is a bit trickier. 3 cases exist
  1. Node to be deleted has no children (leaf node)
     - Delete 9
  2. Node to be deleted has a single child
     - Delete 7
  3. Node to be deleted has 2 children
     - Delete 6



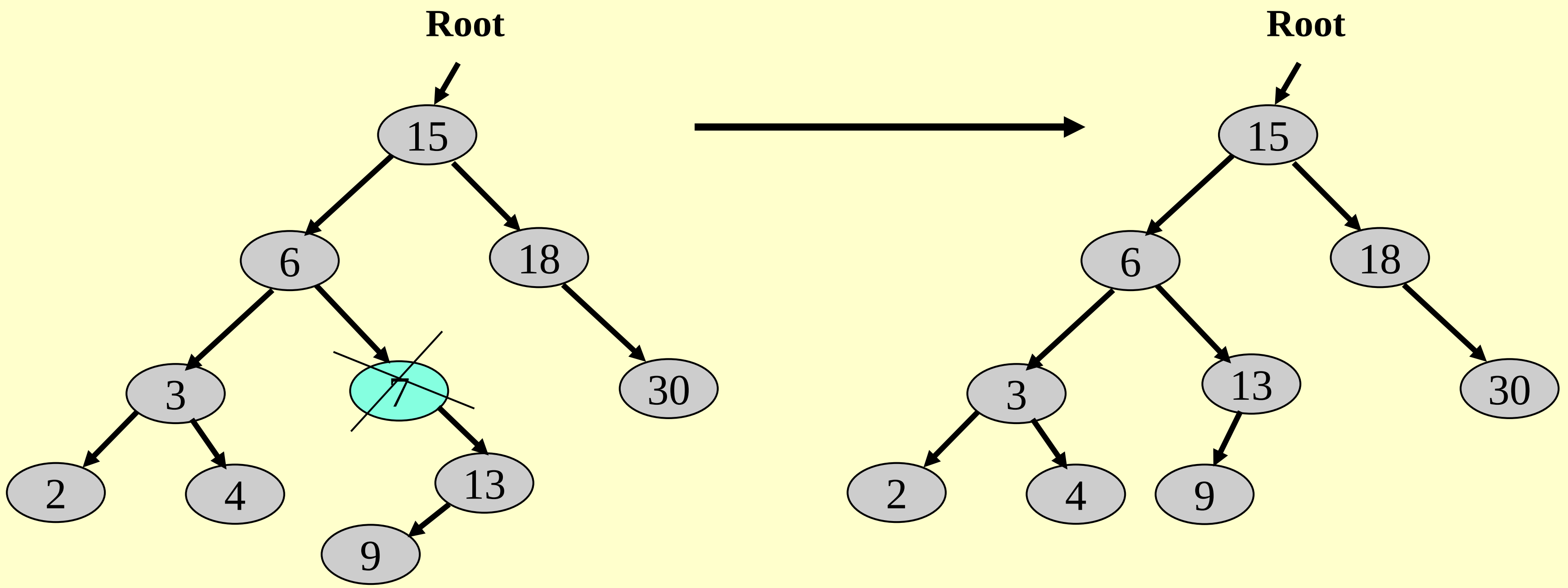
Deletion: Case 1 – Deleting a leaf Node



Deleting 9: Simply remove the node and adjust the pointers

After 9 is deleted

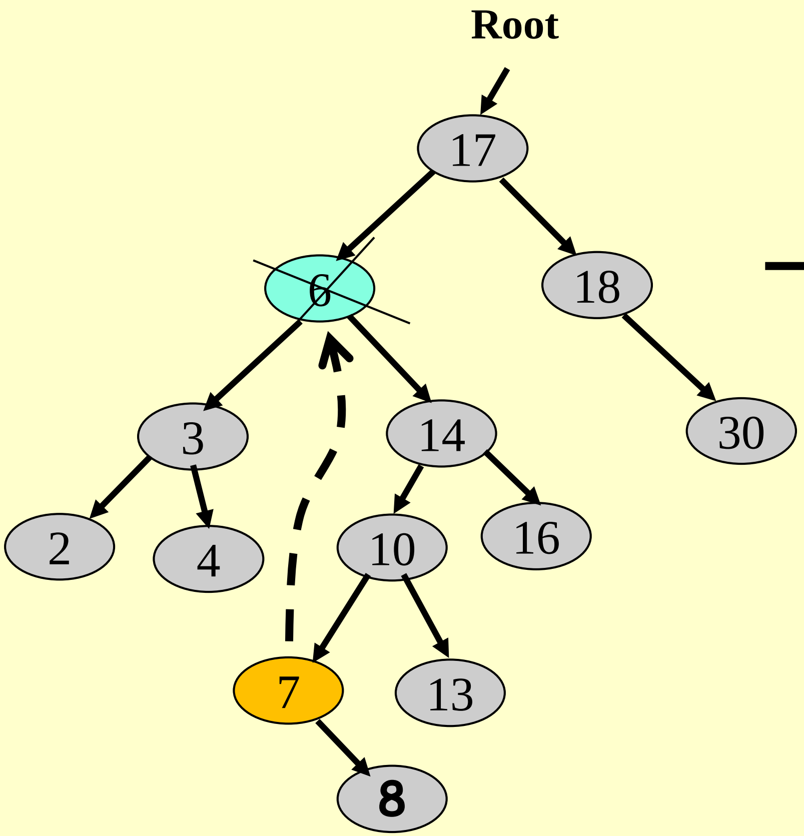
Deletion: Case 2 – A node with one child

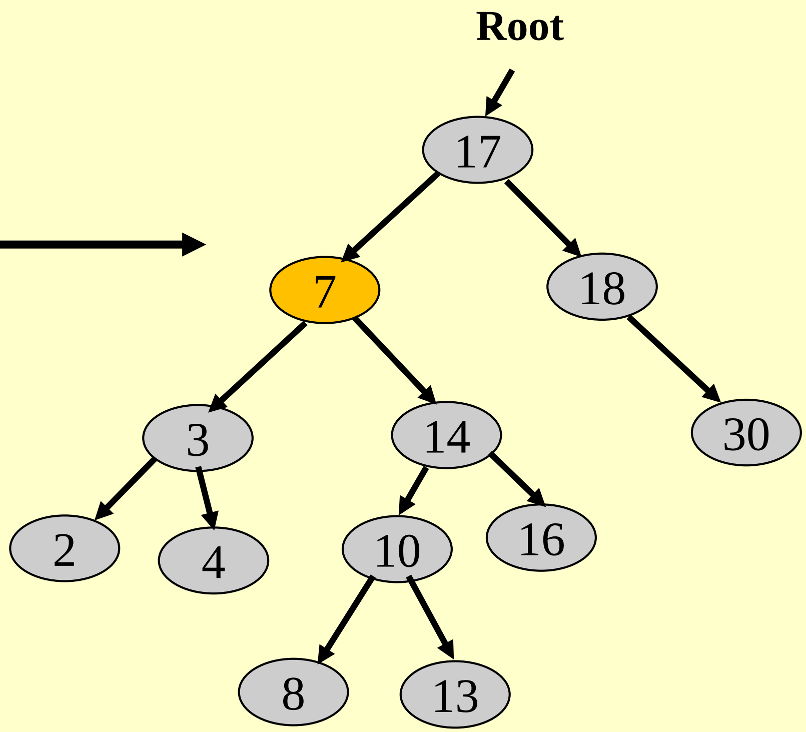


After 7 is deleted

Deleting 7: "Splice out" the node
By making a link between
its child and its parent

Deletion: Case 3 – Node with 2 children





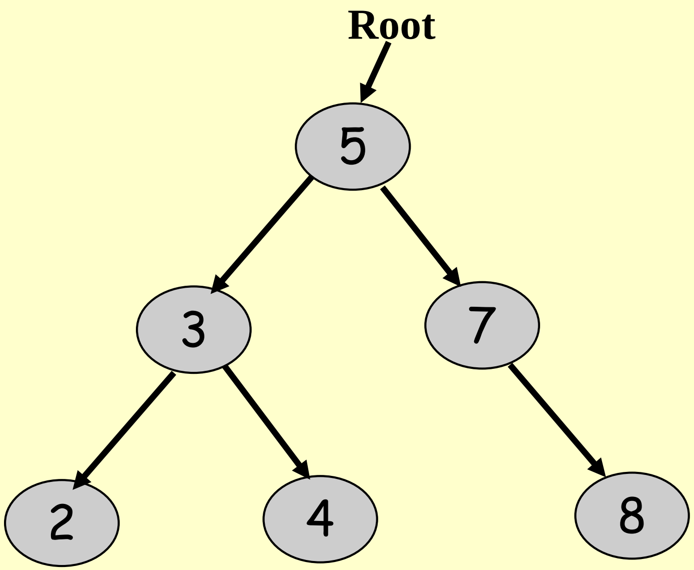
After 6 is deleted

Deleting 6: "Splice out" 6's successor 7, which has no left child, and replace the contents of 6 with the contents of the successor 7

Note: Instead of z's successor, we could have spliced out z's predecessor

Sorting by inorder traversal of a BST

- BST property allows us to print out all the keys in a BST in sorted order by an inorder traversal



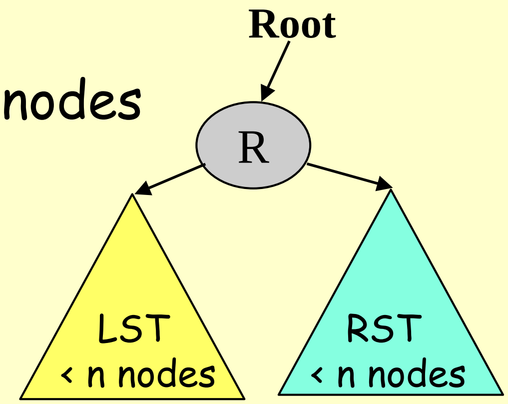
Inorder traversal results

2 3 4 5 7 8

- Correctness of this claim follows by induction in BST property

Proof of the Claim by Induction

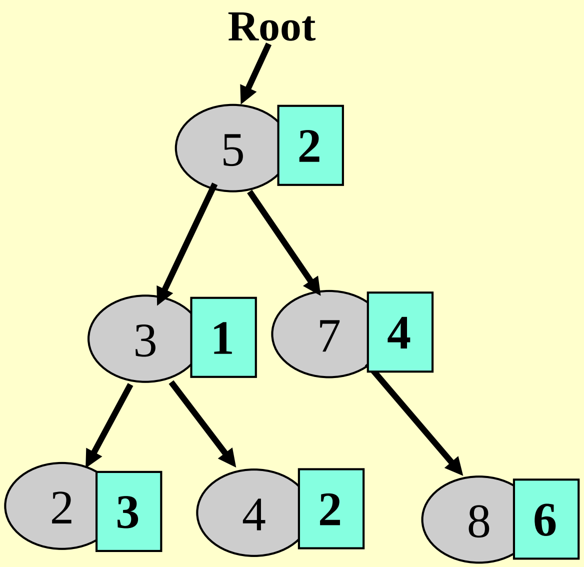
- Base: One node 5 ➔ Sorted
- Induction Hypothesis: Assume that the claim is true for all tree with < n nodes.
- Claim Proof: Consider the following tree with n nodes

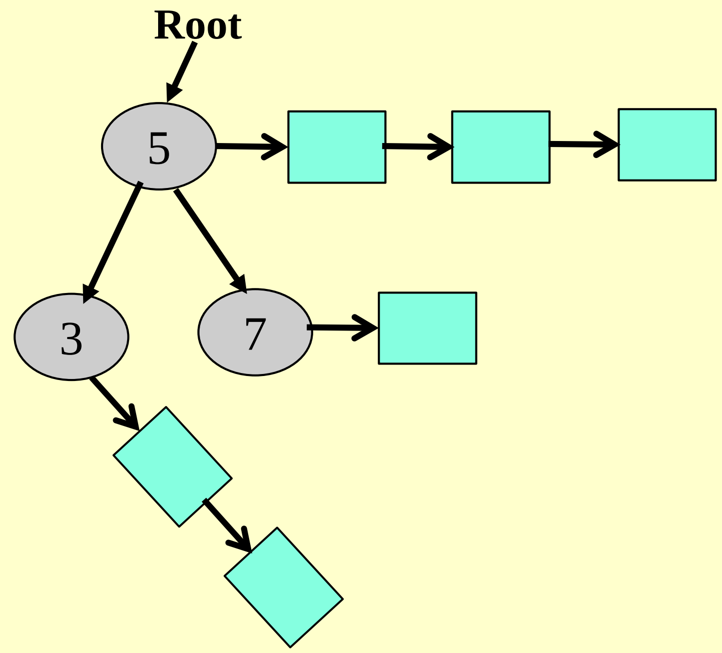


1. Recall Inorder Traversal: LST – R – RST
2. LST is sorted by the Induction hypothesis since it has < n nodes
3. RST is sorted by the Induction hypothesis since it has < n nodes
4. All values in LST < R by the BST property
5. All values in RST > R by the property
6. This completes the proof.

Handling Duplicates in BSTs

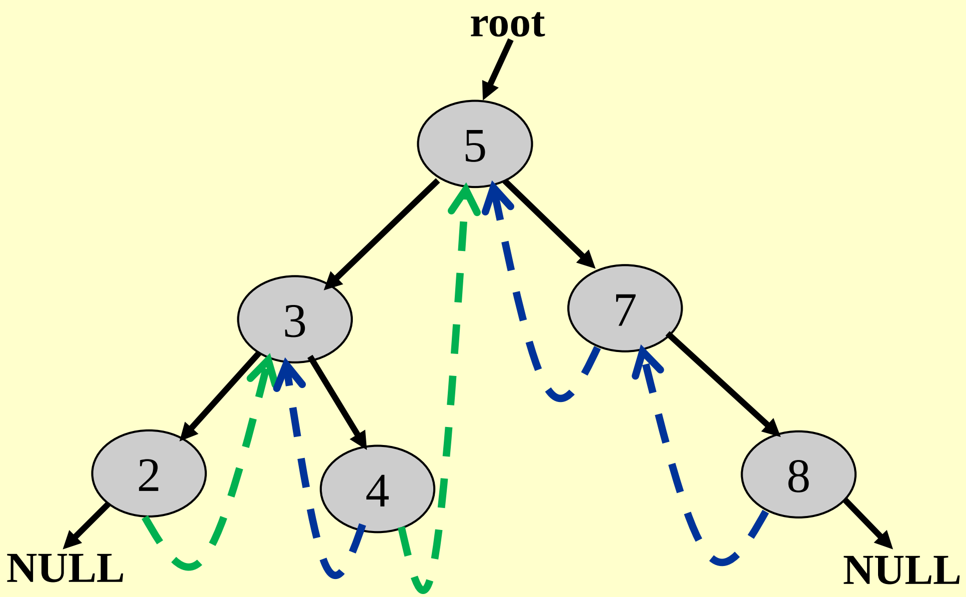
- Handling Duplicates:
  - Increment a counter stored in item's node
- Or
  - Use a linked list at item's node





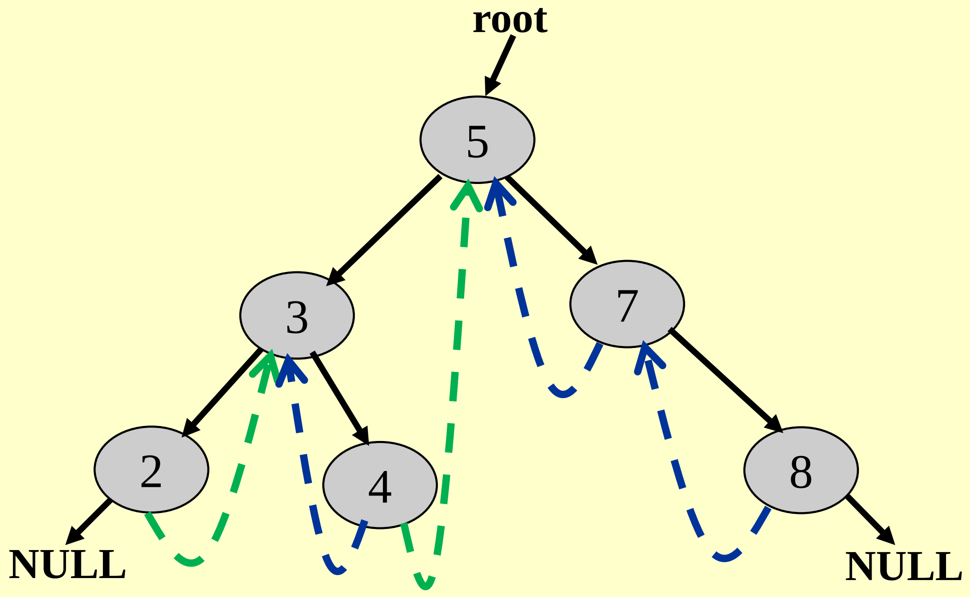
Threaded BSTs

- A BST is threaded if
  - all right child pointers, that would normally be null, point to the inorder successor of the node
  - all left child pointers, that would normally be null, point to the inorder predecessor of the node



Threaded BSTs - More

- A threaded BST makes it possible
  - to traverse the values in the BST via a linear traversal (iterative) that is more rapid than a recursive inorder traversal
  - to find the predecessor or successor of a node easily



Laziness in Data Structures

A "lazy" operation is one that puts off work as much as possible in the hope that a future operation will make the current operation unnecessary

Lazy Deletion

- Idea: Mark node as deleted; *no need to reorganize tree*
  - Skip marked nodes during Find or Insert
  - Reorganize tree only when number of marked nodes exceeds a percentage of real nodes (e.g. 50%)
  - Constant time penalty only due to marked nodes – depth increases only by a constant amount if 50% are marked undeleted nodes (N nodes max N/2 marked)
  - Modify Insert to make use of marked nodes whenever possible e.g. when deleted value is re-inserted
- Gain:
  - Makes deletion more efficient (Consider deleting the root)
  - Reinsertion of a key does not require reallocation of space

- Can also use lazy deletion for Linked Lists

Application of BSTs (1)

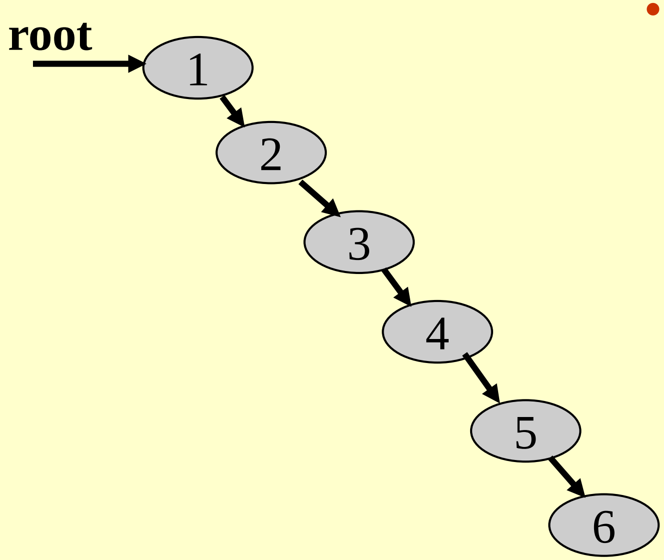
- BST is used as "Map" a.k.a. "Dictionary", i.e., a "Look-up" table
  - That is, BST maintains (key, value) pairs
  - E.g.: Academic records systems:
    - Given SSN, return student record (SSN, StudentRecord)
  - E.g.: City Information System
    - Given zip code, return city/state (zip, city/state)
  - E.g.: Telephone Directories
    - Given name, return address/phone (name, Address/Phone)
    - Can use dictionary order for strings – lexicographical order

Application of BSTs (2)

- BST is used as "Map" a.k.a. "Dictionary", i.e., a "Look-up" table
  - E.g.: Dictionary
    - Given a word, return its meaning (word, meaning)
  - E.g.: Information Retrieval Systems
    - Given a word, show where it occurs in a document (word, document/line)

Taxonomy of BSTs

- O(d) search, FindMin, FindMax, Insert, Delete
- BUT depth "d" depends upon the order of insertion/deletion
- Ex: Insert the numbers 1 2 3 4 5 6 in this order. The resulting tree will degenerate to a linked list-> All operations will take O(n)!



- Can we do better? Can we guarantee an upper bound on the height of the tree?
  1. AVL-trees
  2. Splay trees
  3. Red-Black trees
  4. B trees, B+ trees