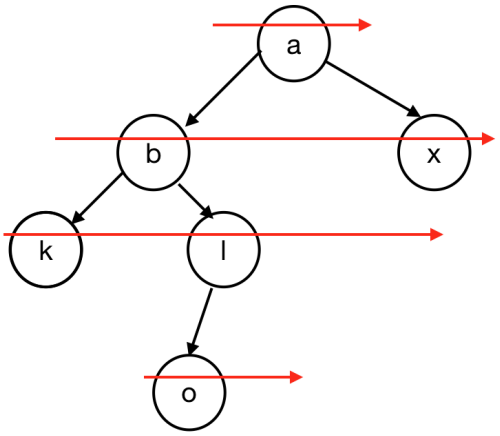
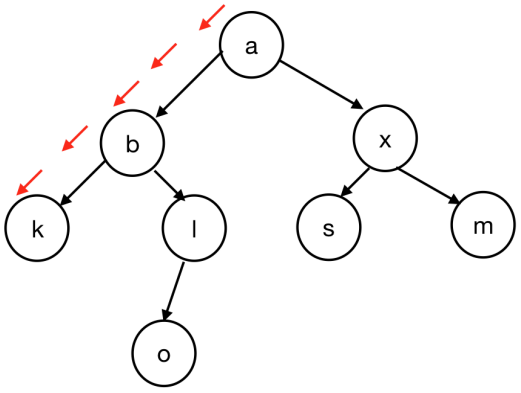
TA Lecture Notes: Lecture #16 - Binary Search Tree
February 21, 2018 • by *Milica Hadzi-Tanovic*

In the previous lecture, we said that traversal means visiting each node in the tree exactly once, while search refers to finding a specific node in the tree (we stop traversing when we find the node). We argued that search is at least as fast as traversal.

- The following are the common ways to search through the tree:
  - **Breadth First Search (BFS)** → the idea is to visit the current node, then all its children, and then the children of the children. In other words, visit the direct descendants quicker. This is actually what we called level-order traversal.



  - **Depth First Search (DFS)** → the idea is to move deeper into the tree as quickly as possible. In other words, visit the leaves as soon as possible. This is similar to preorder traversal.



- In a binary tree, find function takes O(1) time in the best case scenario (the node we are looking for is at the root) and O(n) in the worst case scenario (the node we are looking for is the very last node we visit).

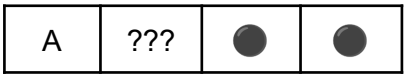
- **Dictionary ADT**
  - Characterized by key-value pairs. We can draw a parallel with lists which can be thought of as set of index-value pairs, so in dictionaries keys are like indices in the arrays.
  - Keys are always unique.
  - Keys and values don't have to be of the same type, but all keys must be of the same type and all values have to be of the same type.
    - For example, a key can be of type string (all keys are of type string) and a value can be a list (all values are lists - all keys map to lists).
  - We want to make dictionary a templated class with two generics:
    - template<typename K, typename V>
  - Basic operations for dictionaries:

Dictionary.cpp

```
1  template<typename K, typename V>
2  class Dictionary {
3      public:
4          V & find(K & key);
5          void insert(K & key, V & value)
6          V & remove(K & key)
7          // We also need an iterator.
8  }
```

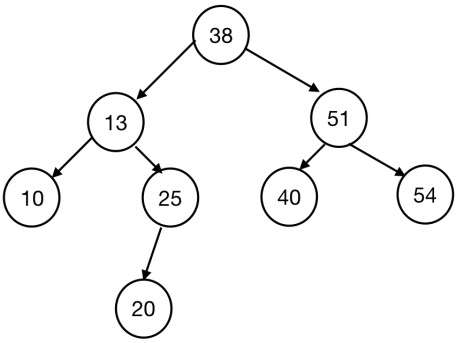
- We said that trees are useful for making certain computation more efficient. Currently the plane binary tree is not very efficient. However, we can make a binary tree into a search structure using dictionaries.

The TreeNode would look like this:



key value left-ptr right-ptr

- **Binary Search Tree (BST)**
  - Properties:
    - Everything to the left of the root is less than the root.
    - Everything to the right of the root is greater than the root.
  - The properties are recursive → true for every node.
  - Formal definition:
    - T = {}
    - T = {d, T_L, T_R}, where $\forall x, x \in T_L , x < d$
      $x \in T_R , x > d$



    - What if x already exists? We cannot have duplicates, keys are unique.
        - We can throw an exception.
        - We can replace the data.
        - Or just do nothing (and possibly return false).

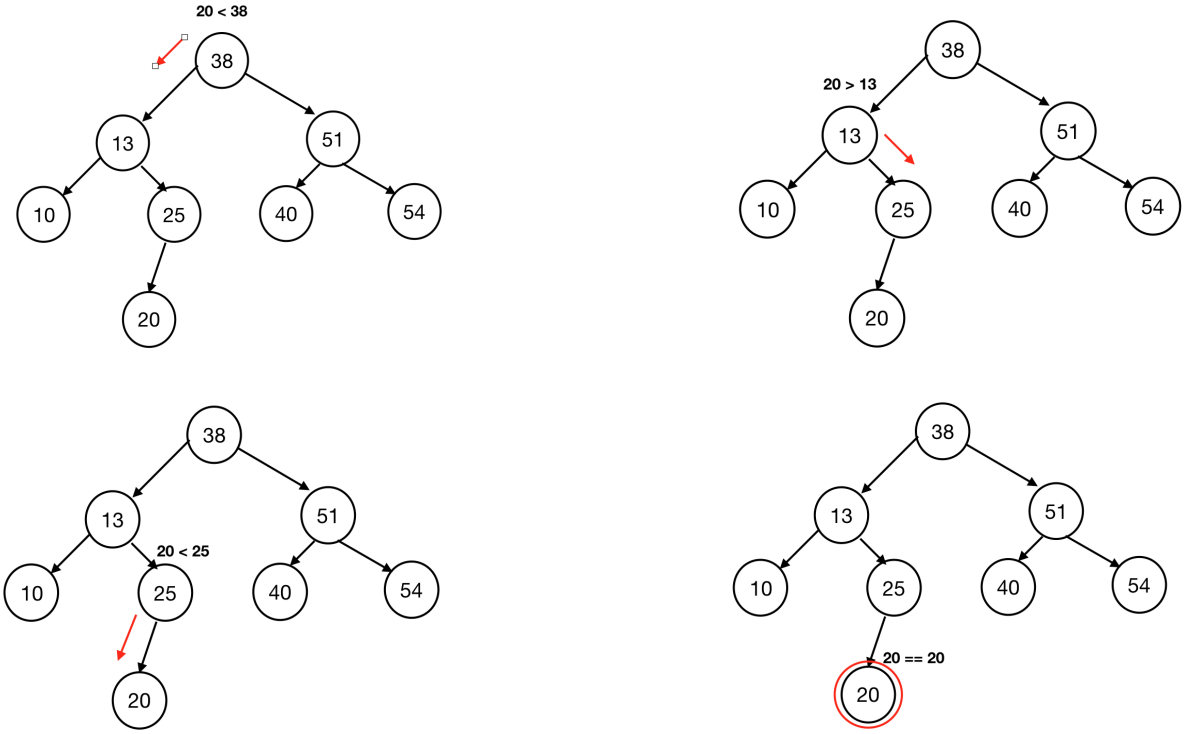
- What happens if we add the smallest number to the tree every time?
    - We are going to get a bad tree, like a linked list.
    - However, there are techniques to fix this.
- Basic operations for BST are find, insert, and remove.
- BST find logic:
    - Starts at the root.
    - Look at the value at the current node.
    - If the value we are looking for is smaller, go to the left.
    - If the value we are looking for is bigger, go to the right.

BST.cpp

```
5   TreeNode * BST::_find(TreeNode *& root, const K & key) const {
6       If (root == NULL || key == root→key) {
7           return root;  // cannot return NULL because we are returning by reference
8       }
9       If (key < root→key) {
10          return _find(root→left, key);
11      }
12      If (key > root→key) {
13          return _find(root→right, key);
14      }
15  }
```

TA Lecture Notes: Lecture #16 - Binary Search Tree

February 21, 2018 • by *Milica Hadzi-Tanovic*



- BST insert logic:
  - Find a position to insert → _find(root, key).
  - Create a new node and insert at the position.
  - Note: Every time we insert, it will always be inserted at the leaf.