A Quick and Dirty Review of Binary Search Trees

The Basic Problem To maintain a dynamic ordered set S, supporting the following *dictionary* operations

- **Search(x):** Find if $x \in S$.
- **Insert(x):** Add x to S.
- **Delete(x):** Delete x from S
- **Pred(x):** Find the predecessor in S of x. x is assumed to be in S
- **Succ(x):** Find the successor of in S of x in S. x is assumed to be in S
- **MIN** and **MAX:** Finding the minimum and maximum items in S.

In what follows we let $n = |S|$.

The Binary Search Tree Solution
A BST is a data structure in which each node is an object that contains three fields: **Key**, **Left**, and **Right**. **Key** is the key of the item being stored at the node (which might also contain a record or pointer to a record associated with the key). **Left** and **Right** are pointers pointing to to the left/right children of the node. These fields contain **NIL** if the associated child doesn't exist.

A BST contains a specified *root node*; *the top of the tree*. The keys in S are stored satisfying the *Binary Search Tree Property:* Let r be a node in the tree. If L is its left child and R its right child, then $key[L] \leq key[r]$ and $key[r] \leq key[R]$.

Notes:

- *Internal nodes* in a BST may have one or two children. Nodes with no children are called *leaves.*
- We emphasize storing a *key* in the tree. In applications, the key might be the key associated with a *record.* The tree node might contain an extra pointer to the actual record or, alternatively, might store the extra data in the tree node itself.
- For convenience, we also provide each node one more field $p[r]$, which points to the *parent* of r. By convention, the parent of the root is itself.

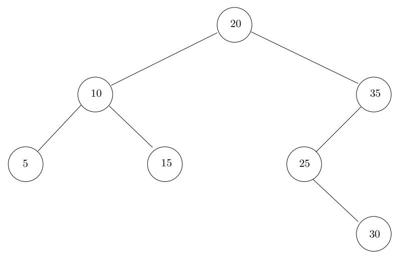

Figure 1: A BST

Walking In Trees There are three well-known types of walks in trees. Each can be implemented in $O(n)$ time. All the walks terminate when they reach a **NIL** node.

- **Inorder(x)**. Inorder (Left[x]); Print key[x]; Inorder(Right[x]).
- **Preorder(x).** Print key[x]; Preorder (Left[x]); Preorder(Right[x]).
- **Postorder(x).** Postorder (Left[x]); Postorder(Right[x]); Print key[x];

Note that if r is the root of the tree, then **Inorder**(r) prints out the elements in S is sorted order. **Preorder**(r) recursively prints the root before the preordering of its left and then right subtrees; **Postorder**(r) recursively prints the left and then right subtrees before the root.

Searching for x in a BST rooted at r

- If $key[r] == NIL$ then x is not in the tree
- If $x == key[r]$ then return r (successful search)
- If $x < key[r]$ then **Search(x)** in $Left[r]$
- Otherwise, **Search(x)** in $Right[r]$.

Note that this walks a path *down* the tree until either x is found or the bottom of the tree is reached. So, the running time of the algorithm is $O(h)$ where h is the *height* of the tree.

Finding Min and Max Let r be a node. Consider $S(r)$ the subset of items in the subtree rooted at r. To find the minimum item in $S(r)$ just follow the left pointers down from r until reaching the last non-NIL item (this could be r itself). To find the maximum item in $S(r)$ follow the right pointers similarly. Note that if r is the root of the actual tree then $S(r) = S$ and this finds **MIN** and **MAX**.

Note that these two operations also run in $O(h)$ time.

Searching for Succ(x) in a BST rooted at r. First Search for X in S. Suppose this is node r.

If $Right[r] \neq NIL$,
$\Rightarrow$ return the Minimum item in the subtree rooted at $Right[r]$.
If $Right[r] == NIL$ and r is the left child of $p[r]$,
$\Rightarrow$ return $key[p[r]]$.
If $Right[r] == NIL$ and r is the right child of $p[r]$
$\Rightarrow$ walk *up* the tree from x until traversing a left edge, and return its parent.

The above can only fail if x is the largest item in the tree, in which case there is no successor. Alternatively, one can add an artificial value ∞ to S to guarantee that every item will always have a successor

Finding $Pred(x)$ is symmetric.

Note that $Succ(x)$ and $Pred(x)$ also require $O(h)$ time.

Question: The above assumed that x was in S. How could we modify the operations to work properly even if $x \notin S$.

Insertion Inserting x is very simple. *Search* for x and find the location where x would be if it had existed. As you are moving down the tree keep track of the last pointers seen so that you know (i) who the parent p of x would have been and (ii) whether x would be the left or right child of p. Create the new node containing x and set the appropriate left/right pointer from p to point to the new node.

Note that this also runs in $O(h)$ time.

Deletion This is the most complicated operation. Let z be the node to be delete with $key[z] = x$.

- If z has no children at all, delete z.
- If z has only one child, set z's parent p to point to z's child and delete z
  (Note the special case when z is the root)

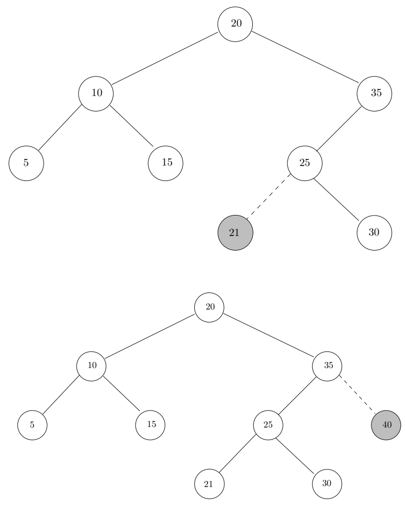

Figure 2: Figure 1, after adding "21" and then "40".

- If z has two children. Let y be the node containing x's successor.
  First note that y has no left child (why?).
  - if y is z's immediate right child then just replace z by y, keeping y's pointers.
  - if y is not z's right child then y is the left child of y's parent p Make the left child of y the new left child of p
    Set $key[z] = key[y]$ and delete the old node y.

This too runs in $O(h)$ time.

Odds and Ends

- All the operations discussed require $O(h)$ time. Unfortunately, h can be as bad as $\Theta(n)$ for regular BSTs. *Balanced* BSTs maintain $h = O(\log n)$ which immediately implies $O(\log n)$ time for all of the operations. The balanced trees we will see are AVL trees but there are many others, e.g., red-black trees, 2-3-4 trees and treaps.
- In our description, information was kept in the internal nodes. There are variations in which all data is kept in the leaves and the internal nodes only contain routing information (to move left or right).
- There are also exist other non-BST data structures structures that can implement $O(\log n)$ dictionary functionality, e.g., skip-lists.

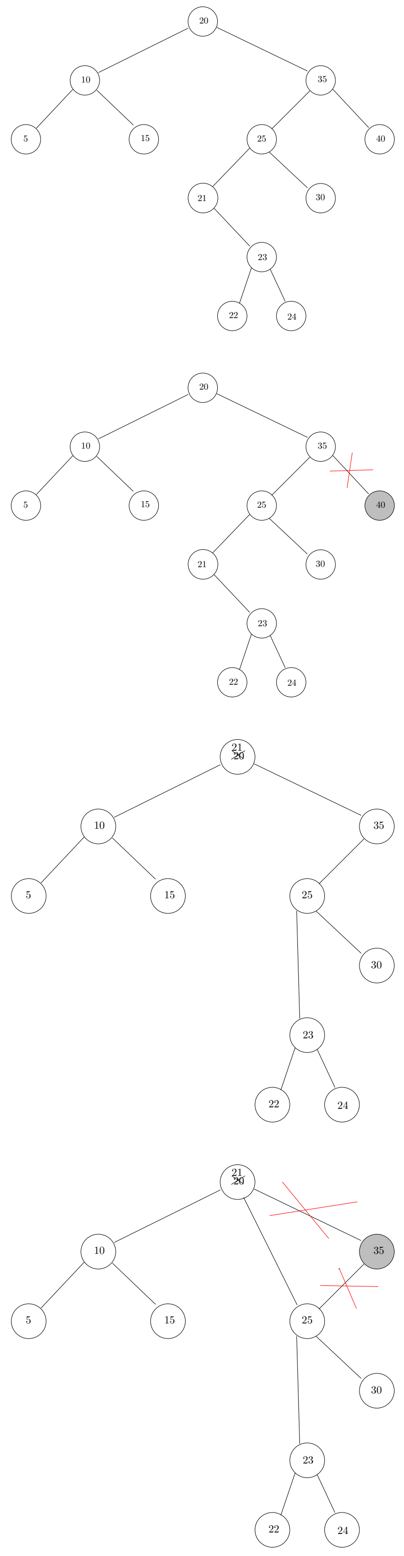

Figure 3: Removing “40” from the first tree is the first case. Removing “20” is the 3rd case with “20” being replaced by its successor “21” and “21”s parent, “25” repointed to “21”s child, “23”. Removing “35” is the 2nd case, with “35”s parent “21” repointing to “35”s unique child, “21”.

- BST’s as discussed were comparison based. Similar to sorting, where we saw both comparison based sorts and representation dependent sorts, e.g., radix sort, there are also representation based search trees. These are called *tries* or *digital search trees*. In binary tries, the root represents the entire set. As you walk down the trie, subsets at level i are split into left and right subtrees depending upon whether their i’th bit is zero or one.