Search Structures

```
interface SearchStructure {
  void insert(Object o); //stick into search structure
  void delete(Object o); //remove objects equal to o from search
  boolean search(Object o);
  int size();
 }
```

Using arrays to implement a Search Structure

- Keep items in an array, and track number of items in array.
- To locate an item l in the array, search the array using binary search. If you find an item that is equal to item l, return true; otherwise return false. Time: $O(log(n))$.
- To insert a new item to the structure, search the array as above. If you find the item is already there, there is nothing to do. Otherwise, if the search procedure says the item ought to be in index i, shuffle all items in array from index i onwards one slot to the right to make room for the new item, and stick the item into index i. Time: $O(n)$
- Deletion is similar: do a search. If item is not in array, there is nothing to do. Otherwise, if item is in index i, shuffle all items from index i+1 onwards one slot to the left to squeeze out the item to be deleted. Time: $O(n)$

See code in SSAsSortedArray.

Lists can also be used to implement Search Structures

Maintain entries in search structure as a sorted list.

Intuitive idea:

- `search`: do linear search on list
- `insert`: as in sorted list code
- `delete`: as in sorted list code

See class SSAsList.

Binary Search Trees

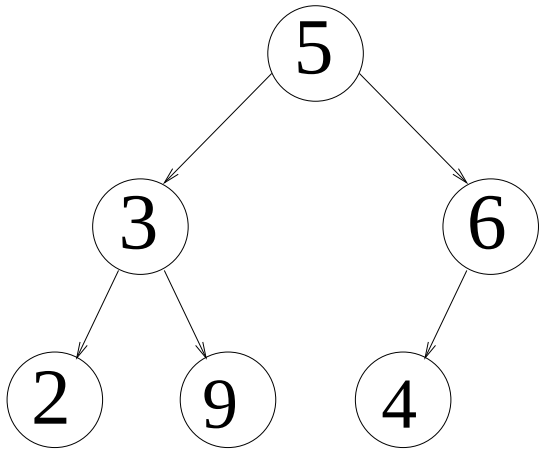
Let us now see how to use trees to implement a fast Search Structure.

Binary tree: contains integers in some order

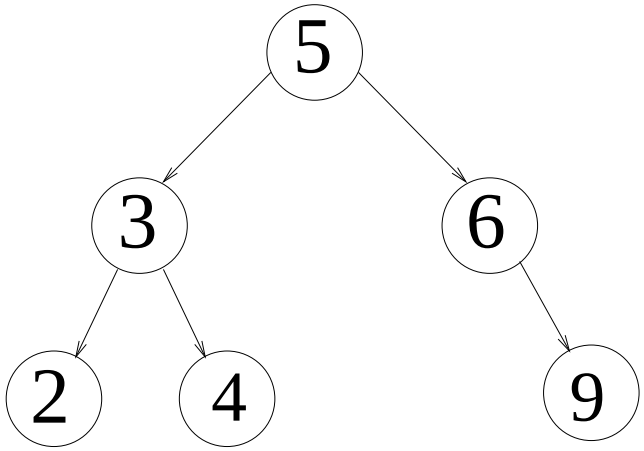
Binary search tree: special case of binary tree

At any node n in the tree,

- all integers smaller than integer at node n are stored in the left subtree
- all integers larger than integer at node n are stored in the right subtree



Not a binary search tree



Binary search tree

Intuition behind binary search trees:

- start with sorted list
- for efficient search, we want access middle of list
- "pick up" list by the scruff of its neck at some internal element (this will be the root of the tree)
- sub-lists to left and right of this element will flop down
- detach these sub-lists
- repeat process recursively with these sublists, hooking their roots to previous root etc.

Algorithm for searching in binary search tree:

- If tree is empty, return false;
- If ((object at root) = (search object)) return true.
- If ((object at root) < (search object)) search in right subtree
- If ((object at root) > (search object)) search in left subtree.

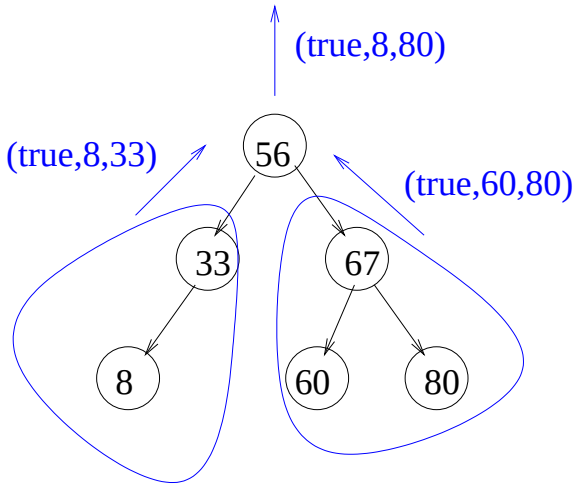
Algorithm for returning largest value in binary search tree:

```
public static Object getMax(TreeCell t) {
    if (t == null) return null;
    if (t.getRight() == null)//t is it
       return t.getDatum();
    else
       return getMax(t.getRight());
}
```

Note: node containing max value will either be a leaf or an internal node that does not have a right child. Similarly, node containing min value will be a leaf or an internal node without a left child.

Algorithm for determining if a tree is a BST

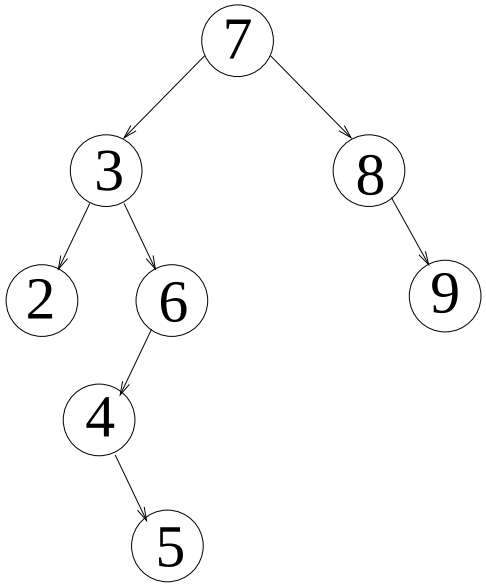
```
empty tree or leaf node: is a bst.
internal node:
  - compute smallest and largest values in left and right subtrees
    and also if both subtrees are bst's themselves
  - if both subtrees are bsts, and
    largest object in left subtree is < object in node and
    smallest object in right subtree is > object in node,
 we have a bst!
```



- perform a "post-order walk" of tree
- visit both subtrees to gather information about these subtrees
  then visit node to process all information about tree

- easy way to remember "post-order": think of postfix expressions
  (operator written after both operands)

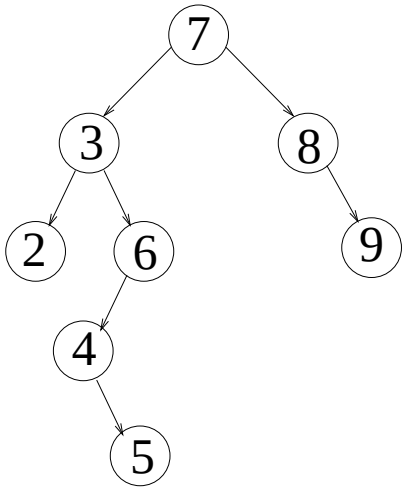
Insertion



Algorithm to insert V into BST:

- Search for V in data structure.
- If V is not there, you'll drop out of BST at some node N.
- Create a new TreeCell T containing V; if V is less than the contents of node N, make T the left child of N; otherwise, make it the right child of N.

Deletion Example



Deleting a node N is easy if

- N is leaf (such as node 9): change reference in parent node of N (node 8) to null
- N has only one child C (such as node 6): change reference in parent to point to C, rather than N (node 3 will point to node 4)
- N has two children (such as node 7): a little tougher...

Helper function `extractMax` : remove largest element in tree

Algorithm:

- Traverse Right tree edges till you reach node (n) for which Right = null
- Value stored at this node n is maximum. Delete this node, and make left subtree of n the right subtree of parent of n.

Note:

Algorithm for deletion: delete integer i

- Walk down tree till you find node N that contains i.
- Let p be the parent node of N.
- If left subtree of N is empty, make right subtree of N into subtree of p.
- If left subtree of N is not empty, extract maximum value from left subtree of N and stick that into N.

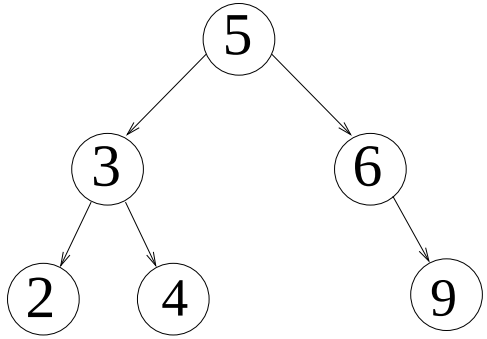
This works, but a more elaborate algorithm might also look to see if right subtree of N is empty before going to extract max from the left subtree.

Intuition behind algorithm: think of tree as a representation of sorted list obtained by picking up list by scruff of its neck.

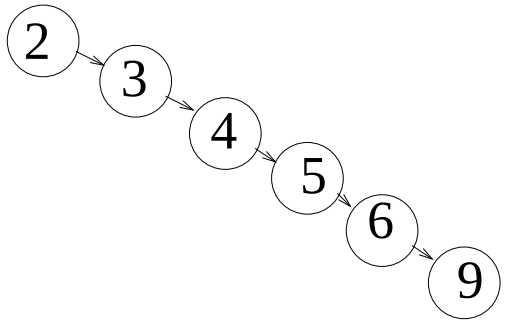
See class BST for code for search/insertion/deletion in binary search trees.

Balanced Binary Search Tree

To get fast search, we would like search tree to be 'bushy' rather than 'skinny'.



Balanced Tree



Unbalanced Tree

Balanced tree: for every node,
height of left subtree = height of right subtree +/- 1

Unfortunately, our trees are not necessarily balanced!

This means search in our bst can sometimes take as long as search in a list!

If tree is balanced, search becomes much more efficient.

Self-balancing Trees

Large body of research on how to 'update' trees on insertion/deletion to guarantee that they are balanced

Options: red-black trees, AVL trees, ...

If you are interested, take CS 410.