CSE 143

Binary Search Trees

[Chapter 10]

A Problem

- Finding a value in a binary tree potentially means visiting *every node*
- Searching a sorted array would still be faster (via binary search)
- If we imposed some ordering on the tree, maybe we could speed things up...
- Leads to the concept of a binary search tree (BST)

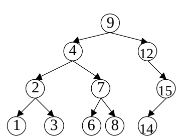
Binary Search Trees (BST)

- Ordering constraints: for every node v,
  - All data in left subtree of v < value of v
  - All data in right subtree of v > value of v
  - Note: no duplicate values
- A binary tree with these constraints is called a *binary search tree* (BST)
- Prerequisite: The items must have a concept of “<” and “>”
  - Does this limit us to ints, doubles, etc.?
  - No! Just need to be able to compare two items
    - In C++, we can even use operator overloading to define <, > etc. for any class.

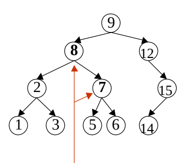
BSTs May Not Be Unique

- Given a set of values, there could be many possible BSTs

Examples and Non-Examples



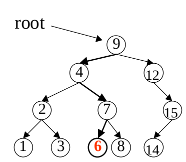
A Binary Search Tree



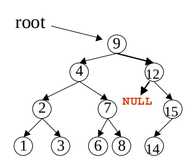
Not a Binary Search Tree

Finding an item in a BST

`find(root, 6)`



`find(root, 10)`



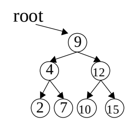
Code For Finding an Item

If we have a binary search tree, we can locate an item like this:

```
// true iff "item occurs in tree with given root"
bool find(BTreeNode *root, int item)  {
  if ( root == NULL )
    return false;
  else if (item == root->data)
    return true;
  else if (item < root->data)
    return find(root->left, item);
  else
    return find(root->right, item);
}
```

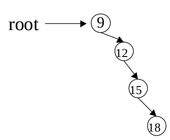

Running time of BST `find`

- Best case: `O(1)`, item is at root
- Worst case: `O(h)`, where `h` is *height* of tree
- Leads to a question:
  - What is the height of a binary search tree with `N` nodes?
- "Full" tree (2^d nodes at each depth `d`) is "shallowest" case:
  - $N = 2^{h+1} - 1$
  - $h = \log_2(N+1) - 1 = O(\log N)$
  - logarithmic running time for find



Running time of `find` (2)

- What if tree isn't balanced?
- Worst case is *degenerate* tree
  - Height = `N`, the number of nodes
- Running time of `find`, worst-case, is `O(N)`



Inserting in a BST

To insert a new key:

- Two base cases:
  - If tree is empty, create new node for item
  - If root holds key, return (no duplicate keys allowed)
- Recursive case:
  - If key < root's value, (recursively) insert in left subtree, otherwise insert in right subtree

Example

Add 8, 10, 5, 1, 7, 11 to an initially empty BST, <u>in that order</u>:

Code For Inserting in a BST

```
// Add data to tree with given root
void insert(BTreeNode *&root, int data) {
  if ( root == NULL ) {
    root = new BTreeNode;
    root->left = NULL;
    root->right = NULL;
    root->item = data;
    return;
  }
  if (data < root->item)
    insert(root->left, data);
  if (data > root->item )
    insert(root->right, data);
}
```

Example (2)

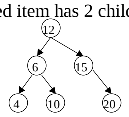
- What if we change the order in which the numbers are added?
- Add 1, 5, 7, 8, 10, 11 to a BST, in that order (following the algorithm):

Complexity of `Insert`

- Base case: `O(1)`
- How many recursive calls?
  - For each node added, takes `O(H)`, where `H` is the height of the tree
- Again, what is height of tree?
  - Balanced trees yields best-case height of `O(log N)` for `N` nodes
  - Degenerate trees yield worst-case height of `O(N)` for `N` nodes
  - For random insertions, expected height is `O(log N)` -- true, but not simple to prove

Deleting an Item from a BST

- An easy strategy: "lazy" deletion
  - have a special bool in the node to mark the node as "deleted" or not
  - leave the node in the tree
- The hard way. Must deal with 3 cases
  - 1. The deleted item has no children (easy)
  - 2. The deleted item has 1 child (harder)
  - 3. The deleted item has 2 children (way hard)



Deletion Algorithm

- First find the node (call it N) to delete.
  - Will also need a pointer to N's parent
- If N is a leaf, just delete it.
- If N has just one child, have N's parent bypass N and point to N's child.
- If N has two children:
  - Replace N's item with the *smallest item K of the right subtree*
  - (Recursively) delete the node that had K (this node is now useless)
    - Note: The smallest item always lives at the leftmost "corner" of a subtree (why?)

Code for Delete

Use two mutually recursive functions:

- void **deleteItem**(int item, BTreeNode *&t);
  - find and delete the node containing "item"
- void **deleteNode**(BTreeNode *&t);
  - delete the root node (only)
    - precondition: t != NULL

Deletion (3): Finding the Node

- This is the "easy" part:

```
void deleteItem(int item,BTreeNode*&t) {
  if (t != NULL) {
    if (item == t->data)
      deleteNode(t);
    else if (item > t->data)
      deleteItem(item, t->right);
    else
      deleteItem(item, t->left);
  }
}
```

Deletion (4): Deleting the Node

```
void deleteNode(BTreeNode*&t) {
  if (t->left && t->right) {    // 2 children
    t->data = findMin(t->right);
    deleteItem(t->data, t->right);
  } else {                      // 0 or 1 child
    BTreeNode* oldVal = t;
    if (t->left)                // left child only
      t = t->left;
    else if (t->right)          // right child only
      t = t->right;
    else                        // no children
      t = NULL;
    delete oldVal; //delete this node
  }
}
```


Deletion (5): Finding Min

- All that remains is to figure out how to find the minimum value in a BST
- Remember, the minimum element lives at the leftmost "corner" of a BST

```
// PRECONDITION:  t is non-NULL
int findMin(BTreeNode* t)
{
  assert(t != NULL);
  while (t->left != NULL)
    t = t->left;
  return t->data;
}
```


Magic Trick

- Suppose you had a bunch of numbers, and inserted them all into an initially empty BST.
- Then suppose you traversed the tree in-order.
- The nodes would be visited in order of their values. In other words, the numbers would come out sorted!
- This is **TreeSort**: another sorting algorithm.
  - O(N log N) most of the time
  - not an "in-place" sort
- Trivial to program if you already have a BST ADT.

Preview of CSE326/373: Balanced Search Trees

- BST operations are dependent on tree height
  - `O(log N)` for `N` nodes if tree is balanced
  - `O(N)` if tree is not
- Can we ensure tree is always balanced?
  - Yes: `insert` and `delete` can be modified to keep the tree pretty well balanced
    - Actually there are several different balanced tree data structures
  - Exact details are complicated
  - Results in `O(log N)` "find" operations, even in worst case

BST Summary

- BST = Binary Trees with ordering invariant
- Recursive BST search
- Recursive `insert`, `delete` functions
- `O(H)` operations, where `H` is height of tree
- `O(log N)` for `N` nodes in balanced case
- `O(N)` in worst case