Lecture 7b: Trees in C++

CS 70: Data Structures and Program Development

Thursday, March 5, 2020

Recall: BST lookup pseudocode

```
lookup(tree, x):
   if tree is empty:
      return false

   else if x == tree's root:
      return true

   else if x < tree's root:
      return lookup(left subtree, x)

   else if tree's root < x:
      return lookup(right subtree, x)
```

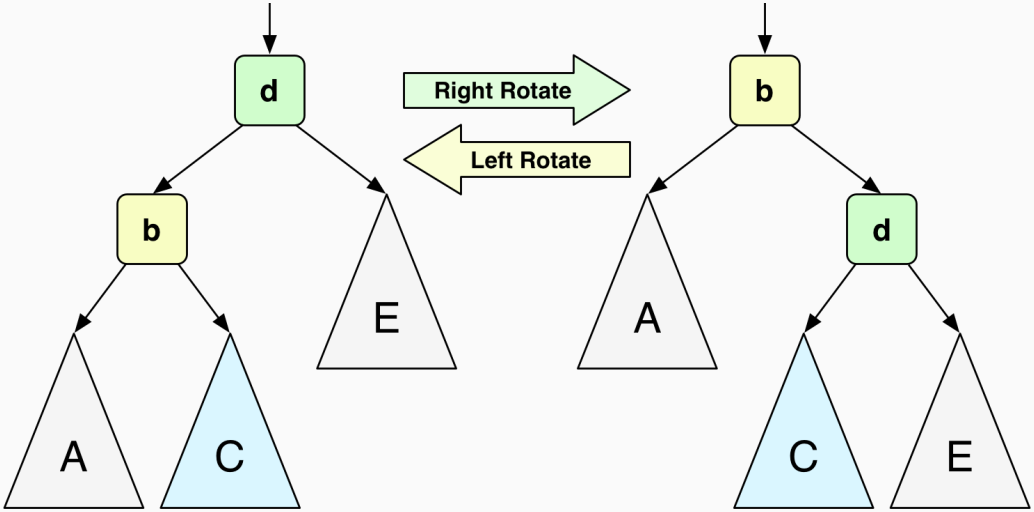
Recall: `insert` pseudocode

```
insert(tree, x):
   if tree is empty:
      make x its new root.

   else if x < tree's root:
      insert(left subtree, x)

   else if tree's root < x:
      insert(right subtree, x)
```

Recall: Tree Rotations



Practice

Do #1 and #2 on the exercise sheet.

Recall: `insertAtRoot` pseudocode

```
insertAtRoot(tree, x):
   if tree is empty:
      make x its new root.

   else if x < tree's root:
      insertAtRoot(left subtree, x)
      do right rotation at tree's root.

   else if tree's root < x:
      insertAtRoot(right subtree, x)
      do left rotation at tree's root.
```

Practice

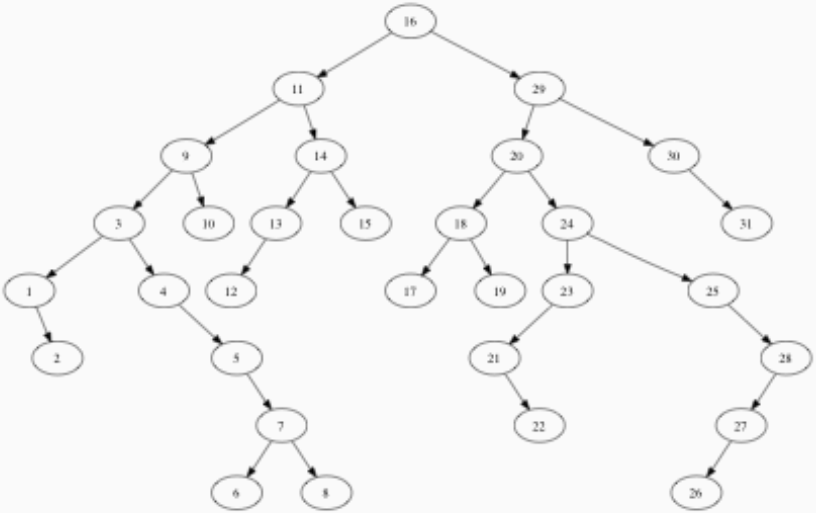
Do #3 on the exercise sheet.

Suppose we have a BST with n nodes.

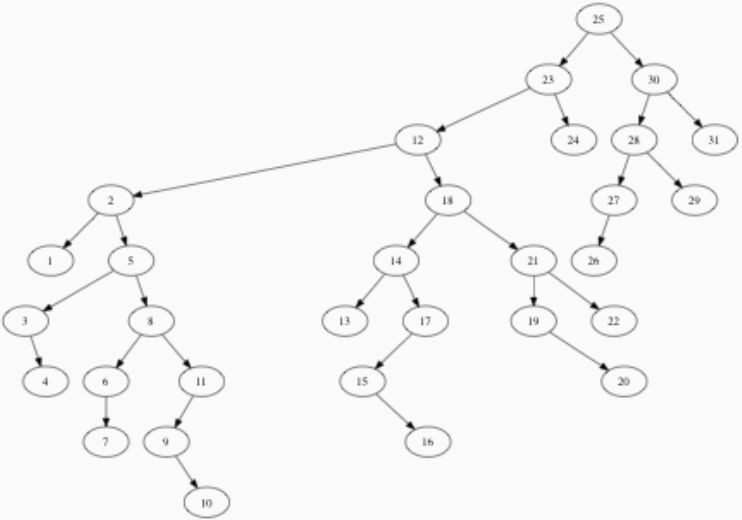
What is the worst-case running time for `find` (and `insert` or `insertAtRoot`)

- if we have a really terrible tree?
- if we have a really nice tree?
- if we have a "random" tree?

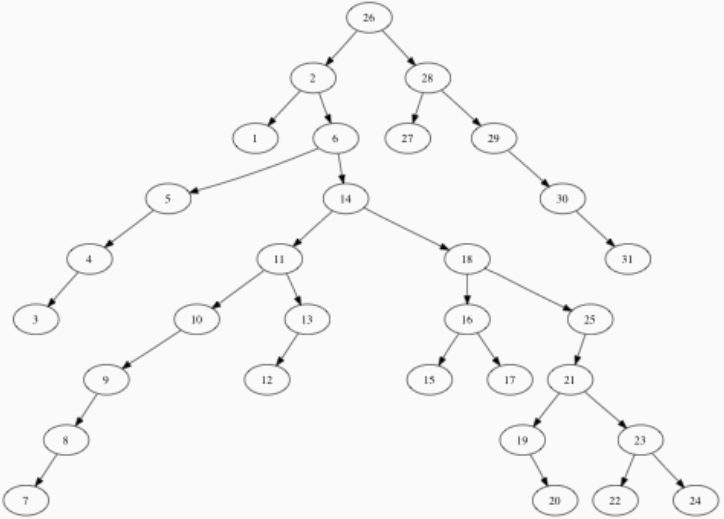
Random trees average 39% worse than perfect



Random trees average 39% worse than perfect



Random Tree average 39% worse than perfect



Building better trees: Off-line algorithm

1. Take the inputs we want to put in the tree.
2. Randomly shuffle them.
3. Build tree by inserting in *shuffled* order.

Building better trees: Randomized Binary Trees

Idea: emulate the nice behavior of pre-shuffled input sequences by inserting each new input at a random level of the tree.

- maybe the root
- maybe the second level (root of a subtree)
- maybe the third level (root of a subsubtree), etc.

Building better trees: Randomized Binary Trees

Idea: emulate the nice behavior of pre-shuffled input sequences by inserting each new input at a random level of the tree.

- maybe the root
- maybe the second level (root of a subtree)
- maybe the third level (root of a subsubtree), etc.

Algorithm: to randomly insert x into a tree already having n nodes

- with probability $1/(n+1)$, do insertAtRoot
- otherwise, *randomly* insert x somewhere in the correct subtree.

Building better trees: Randomized Binary Trees

Idea: emulate the nice behavior of pre-shuffled input sequences by inserting each new input at a random level of the tree.

- maybe the root
- maybe the second level (root of a subtree)
- maybe the third level (root of a subsubtree), etc.

Algorithm: to randomly insert x into a tree already having n nodes

- with probability $1/(n+1)$, do insertAtRoot
- otherwise, *randomly* insert x somewhere in the correct subtree.

Important: lookup in randomized BSTs doesn't change!

BST randomized insert pseudocode

```
randomizedInsert(tree, x):
   if tree currently has n elements,
   with probability 1/(n+1):
      insertAtRoot(tree, x)

   else if x < tree's root:
      randomizedInsert(left subtree, x)

   else if tree's root < x:
      randomizedInsert(right subtree, x)
```

Practice

Do #4 and #5 on exercise sheet.

Practice

Do #4 and #5 on exercise sheet.

Probabilities for insertAtRoot

- 1/6: if rolled die = 1
- 1/4: if rolling die twice gives two even numbers
- 1/2: if rolling die is even
- 1/1: no need to roll

Representing Trees

"A binary tree is empty, or has a root and two (possibly empty) subtrees"

```
class IntTree {
 public: ...
 private:
    // "struct" == ("class" + public by default)
    struct Node {
       int value_;
       Node* left_;
       Node* right_;
    };

    Node* root_;
};
```

Writing `insert`

```
class intTree {
 public:
    void insert(int m);
 private:
    struct Node { ... };
    Node* root_;
};
```

Recall: `insert` pseudocode

```
insert(tree, x):
   if tree is empty:
      make x its new root.

   else if x < tree's root:
      insert(left subtree, x)

   else if tree's root < x:
      insert(right subtree, x)
```

Writing `insert` with a helper function

```
class intTree {
 public:
    void insert(int m);
 private:
    struct Node { ... };
    Node* root_;

    void insertHelper(Node*& nd, int m);
};
```

Insertion Code

```
void IntTree::insert(int m) {
   insertHelper(root_, m);
}

void IntTree::insertHelper(Node*& nd, int m) {
   if (nd == nullptr)
      nd = new Node{m}; // assumes we wrote a constructor
   else if (m < nd->value_)
      insertHelper(nd->left_, m);
   else
      insertHelper(nd->right_, m);
}
```

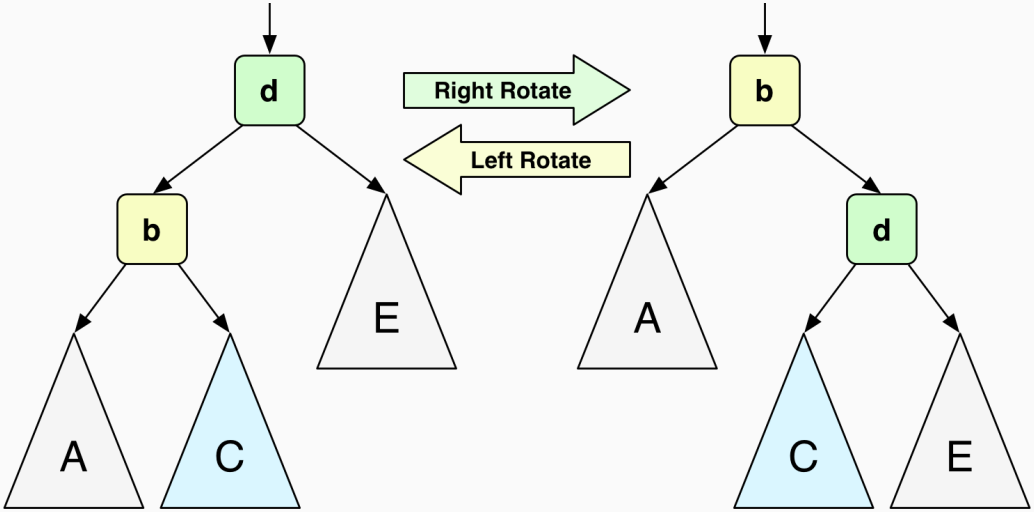
How would we add `exists` ?

```
class intTree {
 public:
    void insert(int m);
 private:
    struct Node { ... };
    Node* root_;

    void insertHelper(Node*& nd, int m);
};
```

Tree Rotations

Recall: Tree Rotations



Just a few pointer updates

```
void rotateRight(Node*& top) {
  Node* b = top->left_;      // b is d's left child
  top->left_ = b->right_;    // C becomes left child of d
  b->right_ = top;           // d becomes right child of b
  top = b;                   // top is now b
}

void rotateLeft(Node*& top) {
  Node* d = top->right_;     // d is b's left child
  top->right_ = d->left_;    // C becomes right child of b
  d->left_ = top;            // b becomes left child of d
  top = d;                   // top is now d
}
```