CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

November 15, 2017

Outline

1. C++ Supplement: 1.2
   - A Binary Search Tree

Outline

1. C++ Supplement: 1.2
   - A Binary Search Tree

The Binary Search Property

A Binary Tree is "Sorted"

A Binary Search Tree, or BST, is a binary tree where every node has the following property:

- Each value in the left subtree is less than the value at the node.
- Each value in the right subtree is greater than the value at the node.

The Binary Search Property

Binary Search With A Binary Tree

- Start at the root
- If the value is there, you are done
- If the value is less than the node value, search the left subtree
- If the value is greater than the node value, search the right subtree

The Binary Search Property

Performance (Running Time) To Find A Value

- Average Performance Is $\Theta(\log n)$. If the tree is not too unbalanced, then we divide the number of items to search in half at each node. This is actually a binary search.
- Worst-Case Performance Is $\Theta(n)$. If the tree branches only to one side (left or right) this is the same as linear search.

Implementing a BST

BST.h (Header File) I

```
class BST
{
public:
   BST();
   ~BST();
   void insert(int value);
   int find(int value);
   void delete_(int value);
```

Implementing a BST

BST.h (Header File) II

```
private:
   TreeNode *_root;
   void _deleteNodes(TreeNode *t);
   TreeNode *_insertRec(TreeNode* t, int value);
   TreeNode *_findRec(TreeNode* t, int value);
   TreeNode *_deleteRec(TreeNode* t, int value);
   TreeNode *_deleteMax(TreeNode *pNode, int& item);
         // updates item
}
```

Implementing a BST

BST.cpp (Constructor)

```
BST::BST()
{
   /*
   post:  creates empty binary search tree
   */
   _root = NULL;
}
```

Implementing A BST

BST.cpp (insert)

```
void BST::insert(int value)
{
   _root = _insertRec(_root, value);
}
TreeNode *BST::_insertRec(TreeNode* t, int value)
{
   if (t == NULL)
      t = new TreeNode(value, NULL, NULL);
   else if (value < t->_item)
      t->_left = _insertRec(t->_left, value);
   else if (value > t->_item)
      t->_right = _insertRec(t->_right, value);
   else
      // raise exception since value is already in tree
   return t;
}
```

Implementing a BST

BST.cpp (find)

```
int BST::find(int value)
{
   TreeNode *t = _findRec(_root, value);
   return t->_item;
}
```

Implementing A BST

BST.cpp (_findRec)

```
TreeNode *BST::_findRec(TreeNode *t, int value)
{
   if (t == NULL) //raise exception since value not in tree
   else
   {
      if (value < t->_item)
         return _findRec(t->_left, value);
      else if (value > t->_item)
         return _findRec(t->_right, value);
      else // found value
         return t;
   }
}
```

Implementing a BST

BST.cpp (delete_)

```
void BST::delete_(int value)
{
   /*
   pre:  item is in tree
   post:  item is not in tree
   */
   _root = _deleteRec(_root, value);
}
```

Implementing A BST

BST.cpp (_deleteRec I)

```
TreeNode *BST::_deleteRec(TreeNode *t, int value)
{
   if (t == NULL)
      //raise exception since value not in tree
   else if (value < t->_item) // deletion from left
      t->_left = _deleteRec(t->_left, value);
   else if (value > t->_item) // deletion from right
      t->_right = _deleteRec(t->_right, value);
   else // delete t itself
```

Implementing a BST

BST.cpp (_deleteRec II)

```
  else // delete t itself
     if (t->_left == NULL) //promote right subtree
     {
        TreeNode *new_t = t->_right;
        delete t;
        t = new_t;
     }
     else if (t->_right == NULL) //promote left subtree
     {
        TreeNode *new_t = t->_left;
        delete t;
        t = new_t;
     }
     else // can't delete t--overwrite with max left value
        t->_left = _deleteMax(t->_left, t->_item);
  return t;
}
```

Implementing a BST

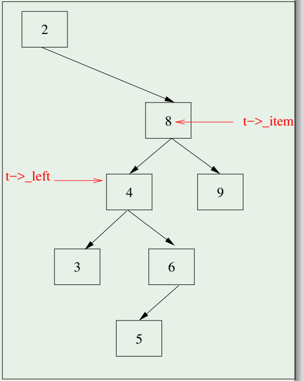
BST.cpp (_deleteMax)

```
// use pass-by-reference to overwrite deleted item value
TreeNode *BST::_deleteMax(TreeNode *t, int& item)
{
   if (t->_right == NULL) // t is the max
   {
      TreeNode *new_t = t->_left;
      item = t->_item;
      delete t;
      return new_t;
   }
   else // max in rt subtree--recursively find and delete it
   {
      t->_right = _deleteMax(t->_right, item);
      return t;
   }
}
```

Deletion Example: DELETE_(8)

Deletion: Neither child is NULL

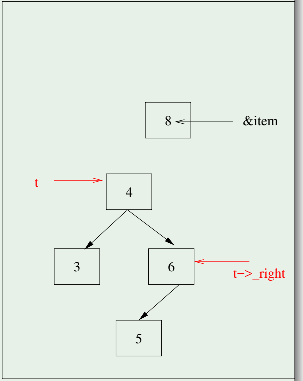
```
  else // 'delete' t
{
   if (t->_left == NULL)
   { //promote right subtree
      TreeNode *new_t = t->_right;
      delete t;
      t = new_t;
   }
   else if (t->_right == NULL)
   { //promote left subtree
      TreeNode *new_t = t->_left;
      delete t;
      t = new_t;
   }
   else // can't delete t-overwrite max
      t->_left = _deleteMax(t->_left,
            t->_item);
return t;
```



Deletion Example: delete_(8)

Deletion: Neither child is NULL

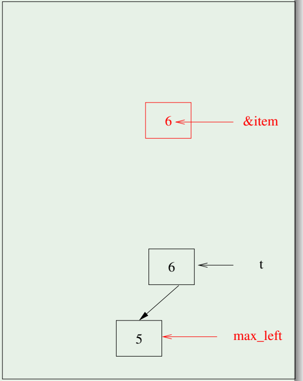
```
 TreeNode *BST::_deleteMax(TreeNode *t,
int& item)
{
   if (t->_right == NULL) // t has the max
   {
      TreeNode *max_left = t->_left;
      item = t->_item;
      delete t;
      return max_left;
   }
   else // max is in right subtree
   {
      t->_right = _deleteMax(t->_right,
         item);
      return t;
   }
```



Deletion Example: delete_(8)

Deletion: Neither child is NULL

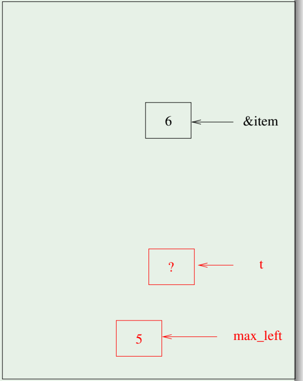
```
 TreeNode *BST::_deleteMax(TreeNode *t,
int& item)
{
   if (t->_right == NULL) // t has the max
   {
      TreeNode *max_left = t->_left;
      item = t->_item;
      delete t;
      return max_left;
   }
   else // max is in right subtree
   {
      t->_right = _deleteMax(t->_right,
         item);
      return t;
   }
```



Deletion Example: delete_(8)

Deletion: Neither child is NULL

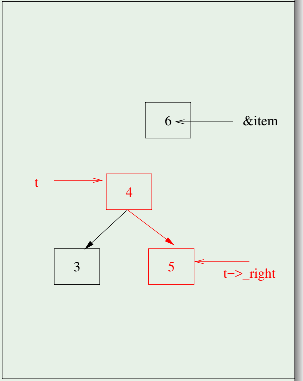
```
TreeNode *BST::_deleteMax(TreeNode *t,
int& item)
{
   if (t->_right == NULL) // t has the max
   {
      TreeNode *max_left = t->_left;
      item = t->_item;
      delete t;
      return max_left;
   }
   else // max is in right subtree
   {
      t->_right = _deleteMax(t->_right,
         item);
      return t;
   }
```



Deletion Example: delete_(8)

Deletion: Neither child is NULL

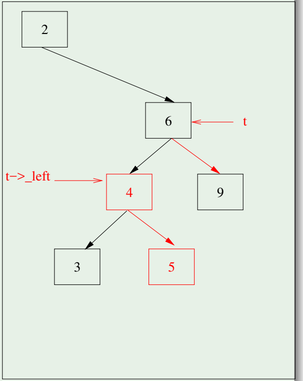
```
  TreeNode *BST::_deleteMax(TreeNode *t,
int& item)
{
   if (t->_right == NULL) // t has the max
   {
      TreeNode *max_left = t->_left;
      item = t->_item;
      delete t;
      return max_left;
   }
   else // max is in right subtree
   {
      t->_right = _deleteMax(t->_right,
         item);
      return t;
   }
```



Deletion Example: delete_(8)

Deletion: Neither child is NULL

```
  else // 'delete' t
{
   if (t->_left == NULL)
   { //promote right subtree
      TreeNode *new_t = t->_right;
      delete t;
      t = new_t;
   }
   else if (t->_right == NULL)
   { //promote left subtree
      TreeNode *new_t = t->_left;
      delete t;
      t = new_t;
   }
   else // can't delete t-overwrite max
      t->_left = _deleteMax(t->_left,
            t->_item);
return t;
```



Traversing A BST

Copy Data Into An Array

- Use inorder traversal to keep items in order.
- Then process the array.
- Disadvantage: uses extra memory for the array.

Run-Time Analysis Of BST Methods

Methods

- visit is $\Theta(n)$.
- insert, delete, find have $\Theta(\log n)$ average behavior.
- insert, delete, find have $\Theta(n)$ worst-case behavior.