CPSC 223
Algorithms & Data Abstract Structures

Lecture 14:
Quicksort exercise
Binary Search Tree Traversal and Deletion

Today …

• Quiz 6
• Sorting Exercise (Quicksort)
• Tree traversals
• Binary Search Tree Deletion
  – Ch 10: pp. 546-563
**Quicksort (based on textbook)**

```c
void Quicksort(Entry theArray[], int first, int last)
{
    if(first < last) {
        int pivotIndex = Partition(theArray, first, last);
        // quicksort first half
        Quicksort(theArray, first, pivotIndex - 1);
        // quicksort second half
        Quicksort(theArray, pivotIndex + 1, last);
    }
}
```

**Quicksort (based on textbook)**

```c
int Partition(Entry theArray[], int first, int last)
{
    Entry pivot = theArray[first]; // pivot value
    int lastP1 = first; // last index of first partition
    for(int i = first + 1; i <= last; i++) {
        if(theArray[i] < pivot) {
            lastP1++;
            Swap(theArray[i], theArray[lastP1]);
        }
    }
    Swap(theArray[first], theArray[lastP1]);
    return lastP1;
}
```
Binary Search Tree Traversal

Traversing binary trees

- When traversing a tree, we typically “visit” every node
- As we visit nodes, apply operations (like print)
- We usually move top-to-bottom (“depth first”) and left-to-right
- Traversal used in various tasks (insert, remove, display, etc.)
Traversing binary trees

Tree traversals are usually implemented using recursion
  – Don’t need to know number of nodes, height, etc.
  – It’s possible to traverse without using recursion (but tricky)

But, when traveling a specific path …
  – Can use recursion or iteration (loops)
  – E.g., lookup and retrieve (typically simple loops)

---

Given a tree $T$, we have the following two cases:
  – $T$ is empty (we have nothing to traverse)
  – $T$ is not empty, and so we:

1. Visit the root of $T$
2. Visit the left subtree of $T$
3. Visit the right subtree of $T$

The basic structure of each type of tree traversal we’ll look at (preorder, inorder, postorder)
Depth-First, Left-to-Right Tree Traversal

```
  D
 /   \
B     G
|     |
A     E
|     |
C     H
```

Let's do a (preorder) traversal of this tree ...

- This means visit the root node **first**
- Once visited, traverse left and then traverse right

1. Visit the **root** D of the (sub) tree
Depth-First, Left-to-Right Tree Traversal

2. Traverse to the left subtree of D

3. Visit the root B of the left subtree of D
Depth-First, Left-to-Right Tree Traversal

4. Traverse to the left subtree of B

Depth-First, Left-to-Right Tree Traversal

5. Visit the root A of the left subtree of B
Depth-First, Left-to-Right Tree Traversal

5. Traverse to the left subtree of A
   - The subtree is empty, so nothing to visit
   - We’re done going left ...

Note this is what is meant by “depth first”

6. Traverse to the right subtree of A
   - The subtree is empty, so nothing to visit
   - We’re done going right ...
Depth-First, Left-to-Right Tree Traversal

7. *Finished* with the subtree rooted at A
   - We traversed both left and right subtrees
   - Now return to the *previous subtree*

Depth-First, Left-to-Right Tree Traversal

8. Traverse to the *right subtree* of B
9. Visit the root C of the right subtree of B

10. Traverse to the left subtree of C
    - The subtree is empty, so nothing to visit
    - We're done going right …
11. Traverse to the right subtree of C
   - The subtree is empty, so nothing to visit
   - We're done going left and right …

12. Finished traversing the right subtree of B
    - Now go back to the previous subtree
Depth-First, Left-to-Right Tree Traversal

previous subtree

D

current subtree

B

A

NULL

NULL

G

E

F

H

NULL

NULL

14. Traverse to the right subtree of D

And so on ...

Traversing binary trees

The basic (recursive) algorithm is:

```c
void traverse(const Node* subTreeRoot) {
    if(subTreeRoot == NULL) {
        return;
    }
    traverse(subtreeRoot->leftChild);
    traverse(subtreeRoot->rightChild);
}
```

Note const argument

... we're assuming we are not modifying the tree

... otherwise we wouldn't use const here
Traversing binary trees

When should we “visit” a node?

• In a “preorder” traversal
  – Visit the node first
  – Then traverse left and right

```cpp
void traverse(const Node* subtreeRoot) {
    if(subTreeRoot == NULL)
        return;
    visit(subTreeRoot->item);
    traverse(subtreeRoot->leftChild);
    traverse(subtreeRoot->rightChild);
}
```

What order will the nodes be visited?

```
D B A C G E F H
```

Traversing binary trees

When should we “visit” a node?

• In an “inorder” traversal
  – Traverse left
  – Visit the node
  – Then traverse right

```cpp
void traverse(const Node* subtreeRoot) {
    if(subTreeRoot == NULL)
        return;
    traverse(subtreeRoot->leftChild);
    visit(subTreeRoot->item);
    traverse(subtreeRoot->rightChild);
}
```

What order will the nodes be visited?

```
A B C D E F G H
```
Traversing binary trees

When should we “visit” a node?

• In a “postorder” traversal
  — Traverse left and right
  — Then visit the node

```c
void traverse(const Node* subtreeRoot) {
  if(subTreeRoot == NULL)
    return;
  traverse(subtreeRoot->leftChild);
  traverse(subtreeRoot->rightChild);
  visit(subTreeRoot->item);
}
```

What order will the nodes be visited?

Homework 6: Pre, Post, and In Order Print

These methods should each …

– Use the appropriate traversal (pre, post, inorder)
– Visit the nodes by printing them to an output stream (out)
– To print, use Entry’s operator<< method
  ```c
  out << subTreeRoot->item;
  ```
– The result should look like a dictionary file

You’ll need to create

– Helper methods for the recursion, e.g.,
  ```c
  void prePrintTraversal(const Node*, ostream&)
  ```
– These methods should be declared protected or private
Binary Search Tree Deletion

Removing (Deleting) Nodes

• Removing nodes is where things get tricky …

• If we could restrict remove to just leaf nodes it would be easy

• For general removal, we need to consider some special cases …
Removing Nodes

Cases to consider …

– The tree is empty!
– The item to be removed is not in the tree
– The node containing the item is a leaf node
– The node containing the item has one child
– The node containing the item has two children

Removing Nodes

• To remove a leaf node
  – We delete the node
  – And set the left or right pointer in the parent to NULL

• To remove a node with one child
  – The parent of the node to be deleted “adopts” the child
Removing Nodes

• **Hardest Case**: Remove a node with *two children*
  – Both children cannot be “adopted” by the parent

![Diagram of a tree with a node B being removed, and a new node D replacing B]

Removing Nodes

• To remove a node with two children
  – one solution is to not delete the node …
  – and instead replace the item in the node with another node’s item

![Diagram of a tree with a node B being removed, and node D replacing B]
Removing Nodes

*Which item do we then use as a replacement?*

- There is more than one we can choose …
- Either the *left-most node of the right* subtree
- or the *right-most node of the left* subtree

Here, the right subtree does not have a left-most node

Removing Nodes

*Which item do we then use as a replacement?*

- There is more than one we can choose …
- Either the *left-most node of the right* subtree
- or the *right-most node of the left* subtree

A more typical example
Removing Nodes

- The left-most node of the right subtree is often referred to as the “inorder successor”
  - The node visited next in an inorder traversal

The inorder successor of B is C

- The node visited next in an inorder traversal