ECE 590.01
C++ Programming, Data structures, and Algorithms

Binary Search Trees Continued
Admin

- Hwk 3
  - Good to get started on before midterm

- Project Proposals
  - Read and commented on

- Reading
  - Chapter 4
  - Chapter 12.2 for next time (out of order)

- Midterm soon
  - March 1\textsuperscript{st} in recitation
  - Page of notes
  - Be prepared: know how to code
Remind us where we left off last time?

- Who can remind us what we talked about?
Remind us where we left off last time?

• Who can remind us what we talked about?
  • Recursion
    • Very good for BSTs
    • Which is what we talked about before
      • And are talking about again
**Binary Search Trees (BSTs)**

- Refresher from 1 week ago
  - Left: smaller
  - Right: greater (or equal)
- Saw how to add and find
BST traversal

Suppose we want to do *something* to all items in a tree
- E.g., print them out
- How could we do this?
  - Step 1?
• Answer (a): 2, 43, 119, 245, 327, 456
BST traversal

root

245

43

2

119

327

456

- Answer (a): 2, 43, 119, 245, 327, 456
- Answer (b): 245, 43, 2, 119, 456, 327
Answer (a): 2, 43, 119, 245, 327, 456
Answer (b): 245, 43, 2, 119, 456, 327
Answer (c): 2, 119, 43, 327, 456, 245
(could also reverse all of these)
BST traversal

- In-order: 2, 43, 119, 245, 327, 456
- Pre-order: 245, 43, 2, 119, 456, 327
- Post-order: 2, 119, 43, 327, 456, 245
BST Traversal: which one

- Three algorithms—which to choose?
  - If only requirement is “print them all, doesn’t matter”
  - May have other situations where we care
    - In-order: naturally ordered
    - Pre-order: ???
**BST traversal**

- Pre-order: 245, 43, 2, 119, 456, 327
  - Can anyone think why this might be useful?

```
  245
   /   
  43    456
 /     /   
2      119  327
```
BST traversal

• Pre-order: 245, 43, 2, 119, 456, 327
  • Can anyone think why this might be useful?
  • Save, and restore: get exactly the same tree
    • In-order would give highly un-balanced tree
**BST traversal**

- Post-order: 2, 119, 43, 327, 456, 245
  - When is this useful?
BST traversal

- Post-order: 2, 119, 43, 327, 456, 245
  - When is this useful?
    - Maybe not so much for printing (except RPN exp trees)
    - What about doing other things?
      - Freeing the tree’s memory?
Start with in-order

- In-order: 2, 43, 119, 245, 327, 456
  - How do we come up with this?
  - Everyone take a moment to think out an algorithm...
    - Might help to imagine the tree without numbers
Start with in-order

- Can’t just read off the answer
  - Have to think how you would go about printing in order
In-order traversal algorithm

- Check if trying to traverse empty tree?
  - If so, do nothing
  - If not, then...
    - Traverse left sub tree
    - Print out my value
    - Traverse right sub tree
In-order traversal algorithm

```c
void inorder(BstNode * curr) {
    • Check if trying to traverse empty tree?
        • If so, do nothing
        • If not, then...
            • Traverse left sub tree
            • Print out my value
            • Traverse right sub tree
}
```
**In-order traversal algorithm**

```c
void inorder(BstNode * curr) {
    if (curr == NULL) {
        • If so, do nothing
    }
    else {
        • If not, then...
          • Traverse left sub tree
          • Print out my value
          • Traverse right sub tree
    }
}
```
In-order traversal algorithm

```c
void inorder(BstNode * curr) {
    if (curr == NULL) {
        //nothing
    }
    else {
        • If not, then...
            • Traverse left sub tree
            • Print out my value
            • Traverse right sub tree
    }
}
```
### In-order traversal algorithm

```c
void inorder(BstNode * curr) {
    if (curr == NULL) {
        //nothing
    }
    else {
        • Traverse left sub tree
        • Print out my value
        • Traverse right sub tree
    }
}
```

How do we do this?
In-order traversal algorithm

```c
void inorder(BstNode * curr) {
    if (curr == NULL) {
        //nothing
    }
    else {
        inorder(curr->left);  // Recurse!
        • Print out my value
        • Traverse right sub tree
    }
}
```
In-order traversal algorithm

```cpp
void inorder(BstNode * curr) {
    if (curr == NULL) {
        // nothing
    } else {
        inorder(curr->left);
        cout << curr->value << endl;
        // Traverse right sub tree
    }
}
```
In-order traversal algorithm

```cpp
void inorder(BstNode * curr) {
    if (curr == NULL) {
        //nothing
    }
    else {
        inorder(curr->left);
        cout << curr->value << endl;
        inorder(curr->right);
    }
}
```

Now just a little cleanup
In-order traversal algorithm

void inorder(BstNode * curr) {
    if (curr != NULL) {
        inorder(curr->left);
        cout << curr->value << endl;
        inorder(curr->right);
    }
}

What if we wanted to make this more generic?
Allow us to do something else, not just print?
Generic traversal

• Generic traversal
  • Option 1: Implement an iterator
    • Trickier than list iterator
    • Requires explicit stack
      • Converting head recursion to a loop
  • Option 2:
    • Pass in “what we want to do” to the traversal
      • Could be function pointer
      • Or in C++: an object
Generic Traversal

template<class R, class A> class Function {
    public:
        virtual R invoke(A arg) = 0;
};

• Start with an abstract class that has one function
  • Templated so we can have any return type/argument type
Generic Traversal

template<class R, class A> class Function {
public:
    virtual R invoke(A arg) = 0;
};

template<class A> class Print :
    public Function<void, A> {
    public:
        virtual void invoke(A arg) {
            cout << arg << endl;
        }
    }
};

• Can make sub-classes to do particular tasks
  • E.g., print
Generic Traversal

```cpp
template<class R, class A> class Function {
    public:
        virtual R invoke(A arg) = 0;
};

void inorderApp(Function<void,K>* f, BstNode* curr) {
    if (curr != NULL) {
        inorderApp(f, curr->left);
        f->invoke(curr->value);
        inorderApp(f, curr->right);
    }
}
```

- Now traversal code can use polymorphism
  - f->invoke dynamically dispatched to whatever we pass in
  - f can also keep state, and we could return values if we wanted
What if we want to pass two things?

template<class R, class A> class Function {
    public:
        virtual R invoke(A arg) = 0;
};

• Suppose I want to be able to iterate and pass key and value for a map

• Can use pair<K,V> type
  • E.g., Function<void, pair<K, V> > * f

    f->invoke(pair<K,V>(curr->key, curr->value));
Binary Search Trees (BSTs)

• Next question: how do we delete?
  • What would you get if you deleted 2?
Next question: how do we delete?
- What would you get if you deleted 2?
- Ok, that was pretty easy, but...
Binary Search Trees (BSTs)

- How do you delete 245?
  - Also note, this is a small tree: what if it were more complex?
• How do you delete 245?
  • Think about it…
• How do you delete 245?
  • Think about it...
  • Removing a node with one child is easy:
    • Maybe we can pick a one child node to remove instead
    • Then put it in 245’s place (getting rid of 245, keeping the leaf)
Binary Search Trees (BSTs)

- Could bring either 200 or 298 up to the top
  - Let’s see these
Binary Search Trees (BSTs)

- Could bring either 200 or 298 up to the top
  - Let’s see these
    - Everything to the right is larger (200 < 245 anyways)
    - Everything to the left is smaller (200 was largest on lhs)
Binary Search Trees (BSTs)

• Could bring either 200 or 298 up to the top
  • Let’s see these
    • Everything to left is smaller (298 > 245)
    • Everything to right is larger (298 was smallest on rhs)
Binary Search Trees (BSTs)

- Cold bring either 200 or 298 up to the top
  - Let’s see these
    - Everything to left is smaller (298 > 245)
    - Everything to right is larger (298 was smallest on rhs)
Binary Search Trees (BSTs)

- Two “most similar” one-child nodes
  - Min from the right
  - Max from the left
  - Guaranteed to have one child, easy to find, “fit” in this place
BST Delete

```java
void remove(K k) {
    root = remove(root, k);
}

BstNode * remove(BstNode * curr, K k) {

    Write recursive helper
```
BST Delete

BstNode * remove(BstNode * curr, K k) {
    if (curr == NULL) { return NULL; }  // NULL?
    if (curr == NULL) { return NULL; }  // Not in tree
    if (curr == NULL) { return NULL; }  // That’s fine, we’ll leave I unchanged

BST Delete

BstNode * remove(BstNode * curr, K k) {
    if (curr == NULL) { return NULL; }
    if (curr->k == k) {
        // Either we found what we are looking for, or its to the left or to the right...
        
        // Recursive cases are easy: update left/right with recursive result
    }
    else if (k < curr->key) {
        curr->left = remove(curr->left, k);
    }
    else {
        curr->right = remove(curr->right, k);
    }
    return curr;
}
BST Delete

BstNode * remove(BstNode * curr, K k) {
    if (curr == NULL) { return NULL; }
    if (curr->k == k) {
        if (curr->left == NULL) { return curr->right; }
        if (curr->right == NULL) { return curr->left; }
    }
    else if (k < curr->key) {
        curr->left = remove(curr->left, k);
    }
    else {
        curr->right = remove(curr->right, k);
    }
    return curr;
}

If either left or right is NULL, deletion is easy: return the other side (both may be NULL—that’s fine)
BST Delete

BstNode * remove(BstNode * curr, K k) {
    if (curr == NULL) { return NULL; }
    if (curr->k == k) {
        if (curr->left == NULL) { return curr->right; }
        if (curr->right == NULL) { return curr->left; }
        curr->left = twoChildRm(curr->left, curr);
        return curr;
    }
    else if (k < curr->key) {
        curr->left = remove(curr->left, k);
    }
    else {
        curr->right = remove(curr->right, k);
    }
    return curr;
}
BST Delete

BstNode * twoChildRm(BstNode * curr, BstNode * replace) {
    if (curr->right == NULL) {

    }

    }

    Right is NULL?  Max of that sub-tree
BST Delete

BstNode * twoChildRm(BstNode * curr, BstNode * replace) {
    if (curr->right == NULL) {
        replace->key = curr->key;
        replace->value = curr->value;
        return curr->left;
    }
}

Replace the key (and value if it’s a map, or whatever else we might need) in the node we are deleting.

Then remove this one from the tree
BST Delete

BstNode * twoChildRm(BstNode * curr, BstNode * replace) {
    if (curr->right == NULL) {
        replace->key = curr->key;
        replace->value = curr->value;
        return curr->left;
    }
    curr->right = twoChildRm(curr->right, replace);
    return curr;
}

Otherwise, do recursion on the right child
Binary Search Trees (BSTs)

• Let’s see how this deletes 245
Binary Search Trees (BSTs)

Let’s see how this deletes 245
if (curr->k == k) {
    if (curr->left == NULL) { return curr->right; }
    if (curr->right == NULL) { return curr->left; }
    curr->left = twoChildRm(curr->left, curr);
    return curr;
}
BstNode * twoChildRm(BstNode * curr, BstNode * replace) {
    if (curr->right == NULL) {
        ...
    }
    curr->right = twoChildRm(curr->right, replace);
}
BstNode * twoChildRm(BstNode * curr, BstNode * replace) {
    if (curr->right == NULL) {
        ...
    }
    curr->right = twoChildRm(curr->right, replace);
}
Binary Search Trees (BSTs)

```c
BstNode * twoChildRm(BstNode * curr, BstNode * replace) {
    if (curr->right == NULL) {
        replace->key = curr->key;
        replace->value = curr->value;
        return curr->left;
    }
    // Other cases...
}
```
BstNode * twoChildRm(BstNode * curr, BstNode * replace) {
    if (curr->right == NULL) {
        replace->key = curr->key;
        replace->value = curr->value;
        return curr->left;
    }
    return nullptr;
}
curr->right = twoChildRm(curr->right, replace);
return curr;
curr->right = twoChildRm(curr->right, replace);
return curr;
curr->left = twoChildRm(curr->left, curr);
return curr;
Binary Search Trees (BSTs)

What is wrong with what we did?
Binary Search Trees (BSTs)

What is wrong with what we did?
Leaked memory!
BST Delete

BstNode * twoChildRm(BstNode * curr, BstNode * replace) {
    if (curr->right == NULL) {
        replace->key = curr->key;
        replace->value = curr->value;
        BstNode * temp = curr->left;
        delete curr;
        return temp;
    }
    curr->right = twoChildRm(curr->right, replace);
    return curr;
}

Fix the memory leak here by deleting the node we need to
BstNode * remove(BstNode * curr, K k) {
    if (curr == NULL) { return NULL; }
    if (curr->k == k) {
        BstNode * temp;
        if (curr->left == NULL) {
            temp = curr->right;
            delete curr;
            return temp;
        }
        if (curr->right == NULL) {
            temp = curr->left;
            delete curr;
            return temp;
        }
        // Also need to delete in these two cases
    }
    if (curr->right == NULL) {
        temp = curr->left;
        delete curr;
        return temp;
    }
}

...
BST Delete

- I recommend taking the BST delete code and tracing it
  - Start with the tree on slide 49
  - Remove 2
  - Remove 456
  - Remove 800
  - Remove 900
- For each trace, make sure you understand why we deleted where we did
- This should help your understand of
  - BST delete
  - Recursion
  - Pointers
  - When to delete and why
BSTs: other functionality

• Maps and Sets are great, but BSTs can implement other functionality
  • Ordering property let’s us find things by ranges

• Efficiently find/count how many things in a range
  • int countBetween(K min, K max) {...}

• Can also do things like
  • V findLargestNotLargerThan(K limit) { ...}
    • (or smallestNotSmallerThan)

• Good practice to write these
Wrap-up

• That wraps up today
  • End of material on midterm

• Next time: balancing trees
  • Really ensuring \( \lg N \) access time