Binary Search Trees (BSTs)

• Last time: BSTs
  • 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
• 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," then it doesn't matter
  - May have other situations where we care
    - In-order: naturally ordered
    - Pre-order: ??

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

- Post-order: 2, 119, 43, 327, 456, 245
  - When is this useful?
Start with in-order

In-order traversal algorithm

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

How do we do this?

Recurse!

Now just a little cleanup

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
      • Function object as we saw before
Iterators

- Need more than just "current" in iterator
  - After 2, would need to go "back up" to 43
  - Then right to 119
  - Iterator could keep a stack

- Or tree could have a list threaded through it

Generic Traversal

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

- Easier/cleaner: use generic traversal with Function objects
  - Same basic abstract template we had before

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
  - E.g., Function<void, pair<K, V>> * f

  f->invoke(pair<K,V>({curr->key,curr->value}));
Wrap-up

• Tree Traversals
  • Three kinds:
    • Inorder
    • Pre-order
    • Post-order
  • Could make generic two ways
    • Iterators: need more state than List iterators
    • Function objects