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

Binary Search Trees
Admin

- Hwk 2
  - Due today

- Project Proposals
  - Due this week (apparently I said Fri somewhere?)

- Reading
  - Chapter 4
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?
  • C++ vs C
    • OOP: Inheritance, design
    • Sets
    • Performance
    • Big-O
    • Iterators

• Leftover:
  • Maps
Map: the most useful ADT ever

- Sets are incredibly useful
- Maps are even more useful
  - Track mappings from keys to values
  - Probably the most ubiquitous thing we do in programs

- Interface
  - add(key, value)
  - find(key)
  - delete(key) [maybe?]
  - iterators [maybe?]
Maps with LinkedLists

- Could do Linked List implementation
  - add(key, value)       $O(1)$
  - find(key)            $O(N)$
  - delete(key)          $O(1)$

- Is this good?
  - $O(N)$ sounds good, but can actually do better
  - $O(lg N)$: Double input size, add 1 to runtime.
    - Pretty good: go from 2 Billion to 4 Billion, runtime 31->32
  - How can we do such a thing?
O(lg N): cut the problem in half

- Recall that
  - lg N = x means $2^x = N$
  - lg 32768 = 15
  - lg 65536 = 16
  - lg 131072 = 17
  - ...
  - lg 4 billion $\sim 32$

- To have a lg N algorithm, we have to cut the problem size in half with each step
Binary Search on arrays

- Can perform **binary search** on a sorted array
  - Find a particular element in $\lg N$ time
- Look at middle element
  - Rule out half the array
Binary Search on arrays

• Algorithm: setup
  • Maintain low (inclusive) and high (exclusive) range
    • Invariant: low <= goal < hi
    • Goal = index of value we want
      • If its in the array
      • If not, we will know this when impossible equation
        • X <= goal < X
Binary Search on arrays

- Each step
  - mid = (hi + low) / 2
  - array[mid] = target? Done
Binary Search on arrays

• Each step
  • mid = (hi + low) / 2
  • array[mid] = target? Done
  • array[mid] < target? Rule out bottom half of range
    • low = mid
Binary Search on arrays

Each step

- $\text{mid} = (\text{hi} + \text{low}) / 2$
- $\text{array}[\text{mid}] = \text{target}$? Done
- $\text{array}[\text{mid}] < \text{target}$? Rule out bottom half of range
  - low = mid + 1
- $\text{array}[\text{mid}] > \text{target}$? Rule out top half of range
  - hi = mid
Binary Search on arrays

- Example: looking for 319
  - Array[mid] is 723
Binary Search on arrays

- Example: looking for 319
  - array[mid] is 723
  - Rule out top half and repeat
Binary Search on arrays

- Example: looking for 319
  - array[mid] is 723
  - Rule out top half and repeat
  - 123 is too small, rule out bottom half and repeat
Binary Search on arrays

- Example: looking for 319
  - array[mid] is 723
  - Rule out top half and repeat
  - 123 is too small, rule out bottom half and repeat
  - 244 is too small, rule out bottom half and repeat
Binary Search on arrays

- Example: looking for 319
  - If the array has 319, it has to be here
    - Convince yourself of this
  - If the array does not have 319, then what happens?
Binary Search on arrays

- Example: looking for 319
  - If the array has 319, it has to be here
    - Convince yourself of this
  - If the array does not have 319, then what happens?
Binary Search on arrays

- **Example: looking for 319**
  - If the array has 319, it has to be here
    - Convince yourself of this
  - If the array does not have 319, then what happens?
    - We end up with hi = low
    - Know that we have ruled out entire array, and stop
Binary Search on arrays

- Note: if last value were too low, would still stop
  - Just would move low to mid +1, which is hi
Array binary search

- Suppose we implement a Set of ints with a sorted array
  - Want to check contains in \( O(\log N) \) time....

```java
class IntSet {
    int * array;
    int arraySize;

    ...

    bool contains(int x) {
        ...
        // let's write this
    }
};
```
Array binary search

```c
bool contains(int x) {
    int low = 0;
    int hi = arraySize;
}
```

Need low and hi
Could be anywhere to start:
0 <= goal < arraySize
So
low = 0
hi = arraySize
Array binary search

```cpp
bool contains(int x) {
    int low = 0;
    int hi = arraySize;
    while (low < hi) {

        // If low >= hi, we have ruled out the entire Array, and can stop (and return false)

    }
    return false;
}
```
Array binary search

bool contains(int x) {
    int low = 0;
    int hi = arraySize;
    while (low < hi) {
        int mid = (low + hi) / 2;
        // mid is halfway between lo and hi
    }
    return false;
}
Array binary search

bool contains(int x) {
    int low = 0;
    int hi = arraySize;
    while (low < hi) {
        int mid = (low + hi) / 2;
        if (array[mid] == x) { return true; }
    }
    return false;
}

Maybe we found what we are looking for?
Array binary search

bool contains(int x) {
    int low = 0;
    int hi = arraySize;
    while (low < hi) {
        int mid = (low + hi) / 2;
        if (array[mid] == x) { return true; }
        if (array[mid] < x) { low = mid + 1; }
    }
    return false;
}

Too small?
Move up low end of range
Array binary search

bool contains(int x) {
    int low = 0;
    int hi = arraySize;
    while (low < hi) {
        int mid = (low + hi) / 2;
        if (array[mid] == x) { return true; }
        if (array[mid] < x) { low = mid + 1; }
        else { hi = mid; }
    }
    return false;
}

Too big?
Move down top end of range
Array binary search

```cpp
bool contains(int x) {
    int low = 0;
    int hi = arraySize;
    while (low < hi) {
        int mid = (low + hi) / 2;
        if (array[mid] == x) { return true; }
        if (array[mid] < x) { low = mid + 1; }
        else { hi = mid; }
    }
    return false;
}
```

Why is low = mid + 1
But hi = mid
Why not hi = mid – 1?
Or low = mid?
Array binary search

bool contains(int x) {
    int low = 0;
    int hi = arraySize;
    while (low < hi) {
        int mid = (low + hi) / 2;
        if (array[mid] == x) { return true; }
        if (array[mid] < x) { low = mid + 1; }
        else { hi = mid; }
    }
    return false;
}

Invariants:
low is inclusive
hi is exclusive

Note: low = mid could get “stuck”
Array binary search

```cpp
bool contains(T x) {
    int low = 0;
    int hi = arraySize;
    while (low < hi) {
        int mid = (low + hi) / 2;
        if (array[mid] == x) { return true; }
        if (array[mid] < x) { low = mid + 1; }
        else { hi = mid; }
    }
    return false;
}
```

Could work for any type T where < and == are defined...
If we template the class
template <class T> class Set {
    T * array;
    int arraySize;
    ...
    bool contains(T x) {
        ...//from previous slide
    }
};
Sorted array Set implementation

• Sorted array
  • O(lg N) contains [LLs: O(N)]
  • O(N) insert [LLs: O(1)]
    • Have to iterate through array to find spot
    • Then copy everything else down one space
  • Maybe a good tradeoff if we check contains much more than we insert?

• Can do better!
  • Binary Search Trees
Binary Search Trees (BSTs)

- Pointer based data structure
  - Left + right (rather than “next”)

- Invariant:
  - Everything to the left is smaller
  - Everything to the right is greater (or equal)
**BST-based Map**

- Let’s switch from a Set to a Map
  - If you can do a Map you can do a Set

```cpp
template<class K, class V> class BstMap {
    ...
};
```

Now templated over two things:
- K is the type of the Key
- V is the type of the Value
Remember: map keys to values
Maybe different types
BST-based Map

- Let’s switch from a Set to a Map
  - If you can do a Map you can do a Set

```cpp
template<class K, class V> class BstMap {
    ...

};
```

Could have some AbstractMap which we extend for polymorphism, but that is a secondary concern
template<class K, class V> class BstMap {
private:
    class BSTNode {
        public:
            K key;
            V val;
            BSTNode * left; BSTNode * right;
            BSTNode(K k, V v): key(k), val(v), left(NULL), right(NULL) {} 
    };
    BSTNode * root;

...
Add to BST

- Want to start with a way to add
  
  ```
  void add(K key, V val) {
  ...
  }
  ```

- How do we do this?
  - Step 1?
• Suppose we want to add 57
  • Where could it go?
  • With minimal “disruption”
Incorrect idea

- Suppose we want to add 57
  - **Incorrect approach**: always add as root
    - 2 and 43 are on the wrong side of 57 now!
Incorrect idea

- Suppose we want to add 57
  - Where could it go?
  - Can’t be the root... Must go left of 245... Right of 43...
Incorrect idea

- This spot is possible in this tree, but...
  - Need to know what comes below to be sure
Incorrect idea

- This spot is possible in this tree, but...
  - Need to know what comes below to be sure
  - 45 as left child of 119? 57 can’t go here anymore
Binary Search Trees (BSTs)

- Add at “leaves”
  - Bottom of tree, no more children
  - Find place by checking direction at each node
Binary Search Trees (BSTs)

- Add at “leaves”
  - Bottom of tree, no more children
  - Find place by checking direction at each node
  - Then add there
BST add 3 ways

- Can approach in the same 3 ways as LL add

```cpp
void add(K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    Make a new node
}```
BST add 3 ways

- Can approach in the same 3 ways as LL add

```c
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    if (root == NULL) {
        root = b;
    }
}
```
BST add 3 ways

- Can approach in the same 3 was as LL add

```cpp
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    if (root == NULL) {
        root = b;
    }
    BSTNode * curr = root;
    while (1) {
        Write infinite looking loop
        Then return out fo the middle
        Ugly?  Sometimes
    }
```
BST add 3 ways

• Can approach in the same 3 was as LL add

```c
void add (K key, V val) {
    BSTNode * b = new BSTNode(k,v);
    if (root == NULL) {
        root = b;
    }

    BSTNode * curr = root;
    while (1) {
        if (key < curr->key) {
            Go left?
```
BST add 3 ways

- Can approach in the same 3 ways as LL add

```java
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    if (root == NULL) {
        root = b;
    }
    BSTNode * curr = root;
    while (1) {
        if (key < curr->key) {
            if (curr->left == NULL) {
                curr->left = b;
                return;
            }
        } else {
            curr = curr->right;
        }
    }
}
```

Check if end of tree
BST add 3 ways

- Can approach in the same 3 ways as LL add

```c
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    if (root == NULL) {
        root = b;
    }
    BSTNode * curr = root;
    while (1) {
        if (key < curr->key) {
            if (curr->left == NULL) {
                curr->left = b;
                return;
            }
            curr = curr->left;
        } else {  // If not keep going
            curr = curr->right;
        }
    }
}
```
BST add 3 ways

- Can approach in the same 3 ways as LL add

```c
void add (K key, V val) {
    BSTNode * b = new BSTNode(k,v);
    if (root == NULL) {
        root = b;
    }
    BSTNode * curr = root;
    while (1) {
        if (key < curr->key) {
            if (curr->left == NULL) {
                curr->left = b;
                return;
            }
            curr = curr->left;
        } else {//similar for right
    }
}
```

Copy, paste, Replace left with right
Works, but ugly

• Copy/paste = bad
• Previous approach works, but is ugly

• Fortunately, we all have black belts in pointer kung fu
BST add 3 ways

- Can approach in the same 3 ways as LL add

```java
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
}
```

Start out the same
**BST add 3 ways**

- Can approach in the same 3 was as LL add

```c
void add (K key, V val) {
    BSTNode * b = new BSTNode(k,v);
    BSTNode ** ptr = & root;
    But now, point at where we might
    Want to change
```
BST add 3 ways

- Can approach in the same 3 was as LL add

```c
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    BSTNode ** ptr = & root;
    while (*ptr != NULL) {
        Hey now we can easily write
        a loop condition that makes sense
    }
    We could do this before,
    But it would have been complex
    And cumbersome
```

ECE 590.01 (Hilton): BSTs
**BST add 3 ways**

- Can approach in the same 3 ways as LL add

```cpp
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    BSTNode ** ptr = & root;
    while (*ptr != NULL) {
        if (key < (*ptr)->key) {
            ptr = & (*ptr)->left;
        } else {
            // Go left?
        }
    }
}
```
BST add 3 ways

- Can approach in the same 3 ways as LL add

```java
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    BSTNode ** ptr = & root;
    while (*ptr != NULL) {
        if (key < (*ptr)->key) {
            ptr = & (*ptr)->left;
        } else {
            ptr = & (*ptr)->right;
        }
    }
}
```

Else, go right
**BST add 3 ways**

- Can approach in the same 3 ways as LL add

```cpp
void add (K key, V val) {
    BSTNode * b = new BSTNode(k, v);
    BSTNode ** ptr = & root;
    while (*ptr != NULL) {
        if (key < (*ptr)->key) {
            ptr = & (*ptr)->left;
        } else {
            ptr = & (*ptr)->right;
        }
    }
    *ptr = b;  // Change *ptr
    To our new node after the loop
}
```
Trees: recursive data structures

- BSTs are naturally recursive data structures
  - If you have a BST and you go left, you have a BST
  - Same if you go right

- Can do add recursively too
  - Base case?
    - What is the simplest tree to add to?
Trees: recursive data structures

- BSTs are naturally recursive data structures
  - If you have a BST and you go left, you have a BST
  - Same if you go right

- Can do add recursively too
  - Base case? Empty tree
    - What is the simplest tree to add to?
BST add 3 ways

```c
void add (K key, V val, BSTNode * curr) {
    // Write helper (private)
}
```
void add (K key, V val, BSTNode * curr) {
    if (curr == NULL) {
        return new BSTNode (key, val);
    }
}

Base case:
Add to empty tree?
Tree with one node
void add (K key, V val, BSTNode * curr) {
    if (curr == NULL) {
        return new BSTNode (key, val);
    }
    if (key < curr->key) {
        curr->left = add (key, val, curr->left);
    }
    else {
        curr->right = add (key, val, curr->right);
    }
    return curr;
}

Recursive case?
Set left/right to what you get recursing
Left/right
void add (K key, V val, BSTNode * curr) {
    if (curr == NULL) {
        return new BSTNode (key, val);
    }
    if (key < curr->key) {
        curr->left = add (key, val, curr->left);
    } else {
        curr->right = add (key, val, curr->right);
    }
    return curr;
}

public:
void add (K key, V val) {
    root = add (key, val, root);
}
BST-based Map

- Now can add(key, val) in lg N time (*)
- What about finding something?
  - \( V \) find(K key) ?

- (*) As we’ll see later, we need to balance the tree to ensure lg N time.
BST find

V find (K key) {
    BSTNode * curr = root;
}

Start at the root
BST find

V find (K key) {
    BSTNode * curr = root;
    while (curr != NULL) {
        // Keep looking
        // As long as we haven’t
        // Gone off the end of the tree
    }
}
BST find

V find (K key) {
    BSTNode * curr = root;
    while (curr != NULL) {
        if (curr->key == key) {
            return curr->val;
        }
    }
    Maybe we found it?
}
BST find

V find (K key) {
    BSTNode * curr = root;
    while (curr != NULL) {
        if (curr->key == key) {
            return curr->val;
        }
        if (key < curr->key) {
            curr = curr->left;
        }
        else {
            curr = curr->right;
        }
    }
}

If not, cut the tree in half: either left or right (but never both)
V find (K key) {
    BSTNode * curr = root;
    while (curr != NULL) {
        if (curr->key == key) {
            return curr->val;
        }
        if (key < curr->key) {
            curr = curr->left;
        } else {
            curr = curr->right;
        }
    }
    return ???;
}

What do we return if it's not found?

Note: can't be NULL, May not be returning a pointer
BST find

V find (K key) {
    BSTNode * curr = root;
    while (curr != NULL) {
        if (curr->key == key) {
            return curr->val;
        }
        if (key < curr->key) {
            curr = curr->left;
        }
        else {
            curr = curr->right;
        }
    }
    K dummy()
    return dummy;
}
BST find

\[
V \text{ find } (K \text{ key}) \{
    \text{BSTNode } * \text{ curr } = \text{ root};
    \text{while (curr }\neq \text{ NULL}) \{
        \text{if (curr->key }\neq\text{ key) } \{
            \text{return curr->val;}
        \}
        \text{if (key < curr->key) } \{
            \text{curr }= \text{ curr->left;}
        \}
        \text{else } \{
            \text{curr }= \text{ curr->right;}
        \}
    \}
    \text{throw exception();}
\}
Exceptions

- Exceptions: similar to Java...
  - Except can throw anything (throw 22;)

- Unwinds call stack (returns from functions)
  - Until try{...} catch( ...) {...} is found that can handle the exn

```cpp
try {
  someFun();
  anotherFun();
}
catch(std::exception & e) {
  std::cerr << " Exception " << e.what() << endl;
}
```