Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski
Topics:

- This time:
  - Starting with a dream: binary search in a linked list?
  - How our dream provided the inspiration for the BST
  - Map implemented as a Binary Search Tree (BST)
  - BST insert
  - Big-O analysis of BST

- Next time:
  - BST balance issues
  - Tree traversals
    - Pre-order, In-order, Post-order, Breadth-first
  - Applications of tree traversals
From last time:
Binary Search in a Linked List?

EXPLORING A GOOD IDEA, FINDING WAY TO MAKE IT WORK
Recall our beautiful algorithm: binary search!

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>51</td>
<td>89</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>

- How long does it take us to find data in a sorted array?
  - **Use binary search!**
  - **O(logn):** awesome!!

- Big downside: O(N) insert, to keep the array sorted
Q. Can we do binary search on a linked list?

A. No.

- The nodes are spread all over memory, and we must follow “next” pointers one at a time to navigate (the treasure hunt).
- Therefore cannot jump right to the middle.
- Therefore cannot do binary search.
- Find is $O(N)$: not terrible, but pretty bad compared to $O(\log n)$ or $O(1)$

Let’s brainstorm a wild idea and then see if we can make it work
“What if…?”
The inspiration for Binary Search Trees

- What if...
- ...instead of having a _front pointer in our linked list, we had a pointer to the element we want to look at first in binary search: the exact median/middle element?

That would make the first step of our binary search **really** fast/easy!

- What about the next step? (and the front half of our list, lol)
“What if...?”
The inspiration for Binary Search Trees

- What about the next step? (and the front half of our list, lol)
- Well, we could have the middle element point to the middle element of both the left half and the right half, so the 2nd step of our binary search is easy/fast too!

- Keep doing this until all elements have pointers to the middle of what remains to their left/right sides...voila!
An Idealized Binary Search Tree

- Our class will have a pointer to the median element*, and each element has pointers to the medians of everything to their left and right
  - * actually it's hard to guarantee it will be the exact middle element, more on this, and lots more about Binary Search Trees, next time!
Binary Search Trees

IMPLEMENTING THE MAP INTERFACE WITH BINARY SEARCH TREES
The centrality of search for Map interface

- The Map implementation should be **highly optimized** for finding a key amongst its collection of keys (to access the associated value)
  - Hence looking at binary search a moment ago—very fast search!
- One difference: a map has **keys** and **values**
- So remember our binary search array?

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>51</td>
<td>89</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>

- Now imagine each number stored in here is a key, and has a value attached to it:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>51</td>
<td>89</td>
<td>90</td>
<td>95</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;B+&quot;</td>
<td>&quot;A-&quot;</td>
<td>&quot;B&quot;</td>
<td>&quot;C+&quot;</td>
<td>&quot;A+&quot;</td>
<td>&quot;A-&quot;</td>
<td>&quot;A&quot;</td>
<td>&quot;B-&quot;</td>
<td>&quot;A&quot;</td>
<td>&quot;A&quot;</td>
</tr>
</tbody>
</table>
The centrality of search for Map interface

- And each number stored in here is a key, and has a value attached to it (not all pictured)
Implementing Map interface with a Binary Search Tree (BST)

- Binary Search Tree is one option for implementing Map
  - C++’s Standard Template Library (STL) uses a Red-Black tree (a type of BST) for their map
  - Stanford library also uses a BST

- Another Map implementation is a hash table
  - We will talk about this later!
  - This is what Stanford’s HashMap uses
TreeMap

This is basically the same as Stanford Map. Here in class we’ll call it TreeMap just to be explicit about its implementation.
tree-map.h

template <typename Key, typename Value>
class TreeMap {
public:
    TreeMap();
    ~TreeMap();

    bool isEmpty() const;
    int size() const;
    bool containsKey(const Key& key) const;
    void put(const Key& key, const Value& value);
    Value get(const Key& key) const;
    Value& operator[](const Key& key);

    //...(continued on next slide)
// class TreeMap continued...
private:
    struct node {
        Key key;
        Value value;
        node* left;
        node* right;
    };
    int _size;
    node* _root;
};
BST put()

Pretty simple!
- If key > node’s key
  › Go right!
- If key < node’s key
  › Go left!
- If there is nothing currently in the direction you are going, that’s where you end up
- Example: put(23, value)
Question about our BST `put()` algorithm:

Pretty simple!
- If key > node’s key
  - Go right!
- If key < node’s key
  - Go left!

FAQ. What do we do if the key is equal to the node’s key?

Stanford Map example:

```cpp
Map<int, string> mymap;
mymap.put(5, "five");
mymap.put(5, "cinco"); // what should happen?
cout << mymap.get(5) << endl; // what should print?
```
**BST put() algorithm:**

- If key > node’s key
  - Go right! (if doesn’t exist—place here)
- If key < node’s key
  - Go left! (if doesn’t exist—place here)
- If key is equal, update value here.
**BST** \texttt{put()}

Insert: 22, 9, 34, 18, 3

Your Turn: How many of these result in the same tree structure as above?

- 22, 34, 9, 18, 3
- 22, 18, 9, 3, 34
- 22, 9, 3, 18, 34

A. None of these
B. 1 of these
C. 2 of these
D. All of these

If key > node’s key
  Go right! (if doesn’t exist—place here)
If key < node’s key
  Go left! (if doesn’t exist—place here)
If key is equal, update value here.
BST Big-O Performance

WHAT CAN WE EXPECT FROM A BST-BASED MAP?
Your Turn: What is the worst case cost for doing `containsKey()` in a BST?

A. \(O(1)\)
B. \(O(\log n)\)
C. \(O(n)\)
D. \(O(n \log n)\)
E. \(O(n^2)\)
What is the worst case cost for doing `containsKey()` in a BST *if the BST is balanced*?

O(logN)—awesome!

---

BSTs are **great** when balanced

BSTs are **bad** when unbalanced

- …and **Balance depends on order of insert** of elements…
- …but user controls this, not “us” (author of the Map class)…
- …no way for “us” (author of Map class) to ensure our Map doesn’t perform terribly 😞 😞
Your Turn: how many worst-case BSTs are there?

One way to create a bad BST is to insert the elements in *decreasing* order: 34, 22, 9, 3
That’s not the only way...

How many **distinctly structured** BSTs are there that exhibit the worst case height (worst case is where height equals number of nodes) for a tree with the 4 nodes listed above?

A. 1-3  
B. 4-5  
C. 6-7  
D. 8-9  
E. More than 9

*Bonus question: general formula for any BST of size n?*  
*Extra bonus question (CS109): what is this as a fraction of all trees (i.e., probability of worst-case tree).*
BST and Heap quick recap/cheat sheet

It can be easy to get confused between BST and heap—here’s a quick guide!
BST and Heap Facts (cheat sheet)

**Heap (Priority Queue)**
- **Structure:** must be “complete”
- **Order:** parent priority must be <= both children
  - This is for min-heap, opposite is true for max-heap
  - No rule about whether left child is > or < the right child
- **Big-O:** guaranteed log(n) enqueue and dequeue
- **Operations:** always add to end of array and then “bubble up”; for dequeue do “trickle down”

**BST (Map)**
- **Structure:** any valid binary tree
- **Order:** leftchild.key < self.key < rightchild.key
  - No duplicate keys
  - Because it’s a Map, values go along for the ride w/keys
- **Big-O:** log(n) if balanced, but might not be balanced, then O(n)
- **Operations:** recursively repeat: start at root and go left if key < root, go right if key > root