Programming Abstractions

CS106X

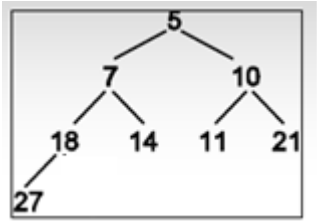
Cynthia Lee

Topics:

- Finish up heap data structure implementation
  - Enqueue (“bubble up”)
  - Dequeue (“trickle down”)
- Binary Search Tree (BST)
  - Starting with a dream: binary search in a linked list?
  - How our dream provided the inspiration for the BST
    - Note: we do NOT actually construct BSTs using this method
  - BST insert
  - Big-O analysis of BST
  - (if we have time) BST balance issues

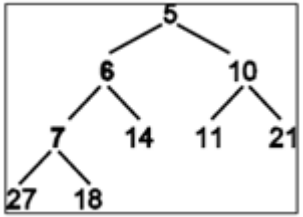
Binary heap insert and delete

Binary heap insert + “bubble up”



Size=8, Capacity=15

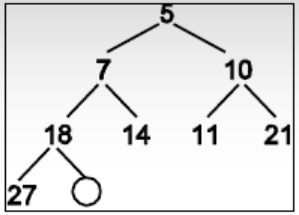
0	1	2	3	4	5	6	7	8	9	…	14
5	7	10	18	14	11	21	27	?	?	…	?



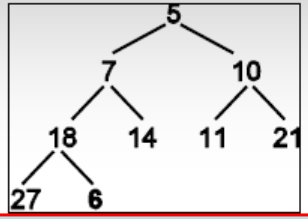
Size=9, Capacity=15

0	1	2	3	4	5	6	7	8	9	…	14
5	6	10	7	14	11	21	27	18	?	…	?

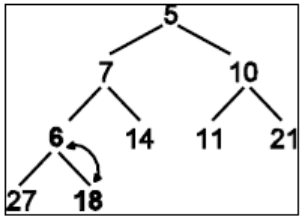
[Binary heap insert reference page]



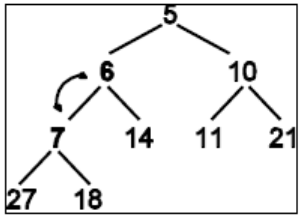
(a) A minheap prior to adding an element. The circle is where the new element will be put initially.



(b) Add the element, 6, as the new rightmost leaf. This maintains a complete binary tree, but may violate the minheap ordering property.

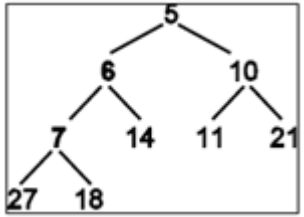


(c) "Bubble up" the new element. Starting with the new element, if the child is less than the parent, swap them. This moves the new element up the tree.



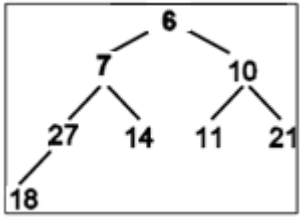
(d) Repeat the step described in (c) until the parent of the new element is less than or equal to the new element. The minheap invariants have been restored.

Binary heap delete + “trickle down”



Size=9, Capacity=15

0	1	2	3	4	5	6	7	8	9	…	14
5	6	10	7	14	11	21	27	18	?	…	?

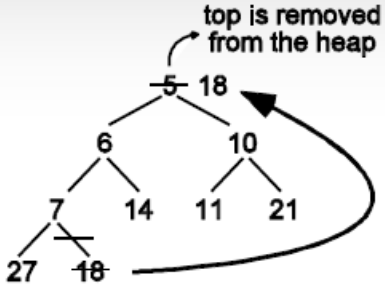


Size=8, Capacity=15

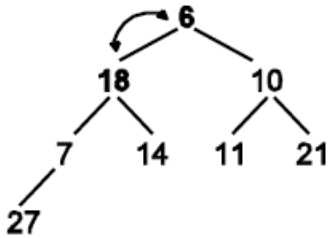
0	1	2	3	4	5	6	7	8	9	…	14
6	7	10	27	14	11	21	18	?	?	…	?

[Binary heap delete + “trickle-down” reference page]

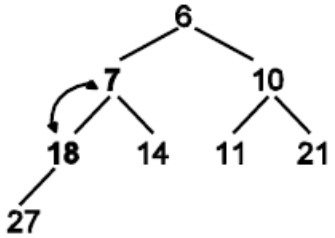
(a) Moving the rightmost leaf to the top of the heap to fill the gap created when the top element (5) was removed. This is a complete binary tree, but the minheap ordering property has been violated.



(b) “Trickle down” the element. Swapping top with the smaller of its two children leaves top’s right subtree a valid heap. The subtree rooted at 18 still needs fixing.



(c) Last swap. The heap is fixed when 18 is less than or equal to both of its children. The minheap invariants have been restored



Binary Search Trees

Implementing the **Map interface** with Binary Search Trees

Implementing Map interface with a Binary Search Tree (BST)

- Often we think of a hash table as the go-to implementation of the Map interface
  - Will talk about this on Thursday!
- Binary Search Trees are another option
  - C++'s **Standard Template Library (STL)** uses a Red-Black tree (a type of BST) for their map
  - Stanford library also uses a BST
    - Use hash-map.h to get a hashing version

Binary Search in a Linked List?

Exploring a good idea, finding way to make it work

Imagine storing sorted data in an array

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- How long does it take us to find?
  - **Binary search!**
  - **O(logn):** awesome!

- But slow to insert
  - **Scoot everyone over to make space**
  - **O(n):** not terrible, but pretty bad compared to log(n) or O(1)
  - In contrast, linked list stores its nodes scattered all over the heap, so it doesn't have to scoot things over

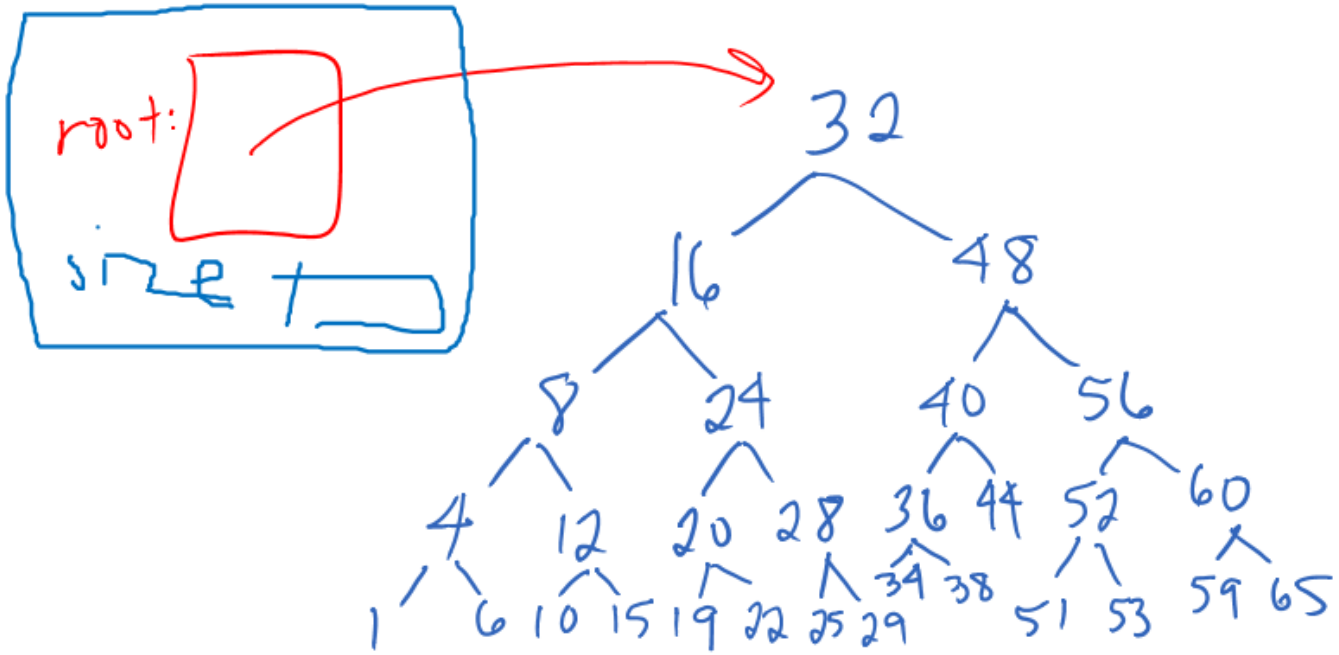
Q. Can we do binary search on a linked list to implement a quick insert?

A. No.

- The nodes are spread all over the heap, and we must follow "next" pointers one at a time to navigate.
- **Therefore cannot jump right to the middle.**
- Therefore cannot do binary search.
- **Find is O(N):** not terrible, but pretty bad compared to O(logn) or O(1)

Binary Search Tree can be thought of as a linked list that has pointers to the middle, again and again (recursively), to form a tree structure

An Idealized Binary Search Tree



TreeMap

An implementation of the Map interface

tree-map.h

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

  bool isEmpty() const { return size() == 0; }
  int size() const { return count; }
  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);
...
```

tree-map.h

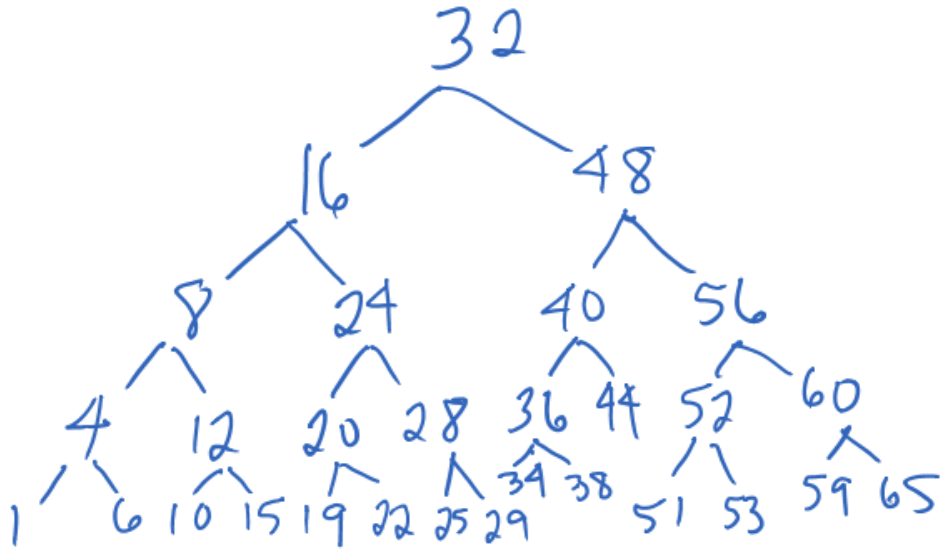
```
...
private:
  struct node {
    Key key;
    Value value;
    node *left, *right;
  };
  int count;
  node *root;

};
```

BST put()

Pretty simple!

- If key > node's key
  - Go right!
- If key < node's key
  - Go left!
- If equal, do *update* of value—no duplicate keys!
- If there is nothing currently in the direction you are going, that's where you end up

Example: insert 2



BST put()

Insert: 22, 9, 34, 18, 3

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

What is the WORST CASE cost for doing containsKey() in 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 BST ***if the BST is balanced?***

O(logN)—awesome!

BSTs is that they are great when balanced

BST is bad when unbalanced

Balance depends on order of insert of elements