Programming Abstractions

CS106B

Cynthia Lee

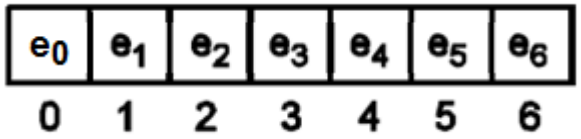
Topics:

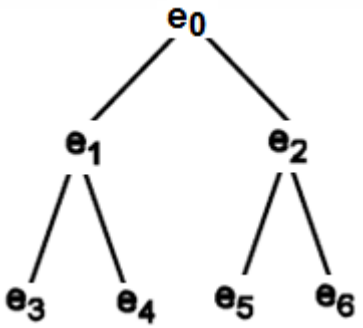
- Finish up **Priority Queue** ADT
  - Heap data structure implementation
    - How do we do insert/remove operations on heaps?
- **Map** implemented as a Binary Search Tree (BST)
  - Starting with a dream: binary search in a linked list?
  - How our dream provided the inspiration for the BST
  - BST insert
  - Big-O analysis of BST
- Friday: more BST!

Priority Queue

Emergency Department waiting room operates as a priority queue: patients are sorted according to priority (urgency), not “first come, first serve” (in computer science, “first in, first out” or FIFO).

Heap in an array

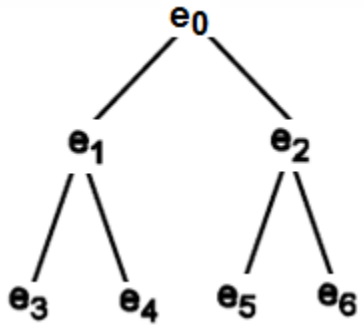




For a node in array index i:

- Q: The parent of that node is found where?
- A: at index:
  - A. i – 2
  - B. i / 2
  - C. (i – 1)/2
  - D. 2i

Fact summary: Binary heap in an array

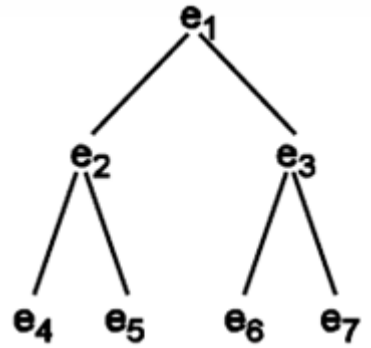


0-based:

For tree of height h, array length is 2^h-1

For a node in array index i:

- Parent is at array index: (i – 1)/2
- Left child is at array index: 2i + 1
- Right child is at array index: 2i + 2



1-based:

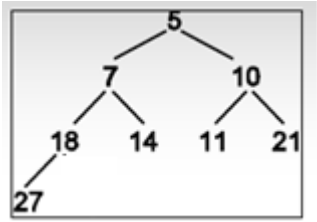
For tree of height h, array length is 2^h

For a node in array index i:

- Parent is at array index: i /2
- Left child is at array index: 2i
- Right child is at array index: 2i + 1

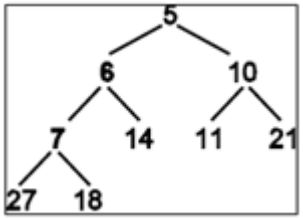
Binary heap enqueue and dequeue

Binary heap enqueue (insert + “bubble up”)



Size=8, Capacity=15

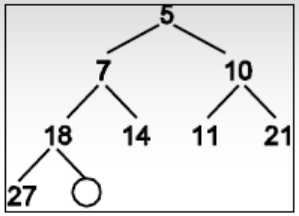
0	1	2	3	4	5	6	7	8
5	7	10	18	14	11	21	27	?



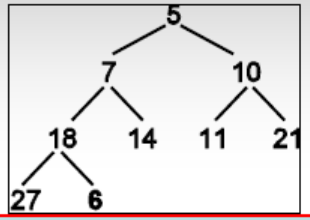
Size=9, Capacity=15

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

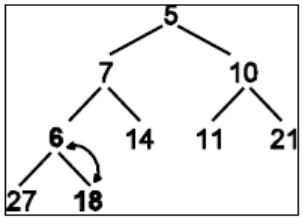
[Binary heap insert reference page]



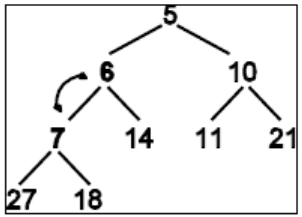
(a) A minheap prior to adding an element. The circle is where the new element will be put initially.



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

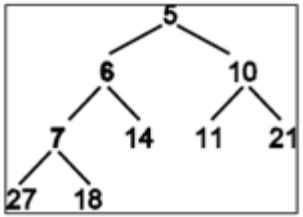


(c) "Bubble up" the new element. Starting with the new element, if the child is less than the parent, swap them. This moves the new element up the tree.



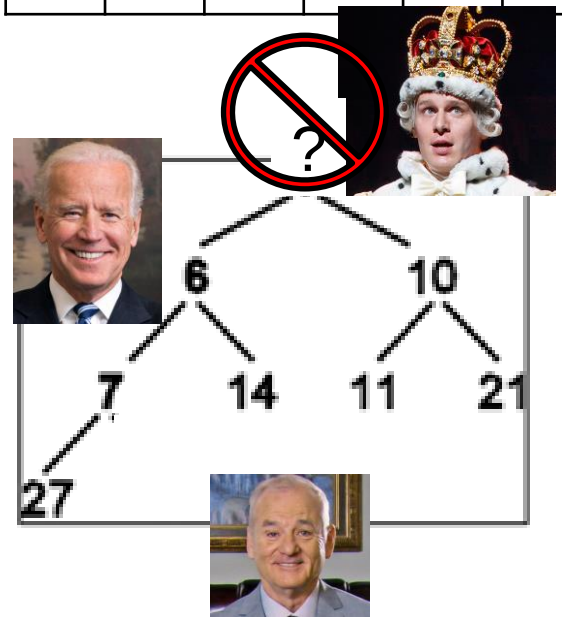
(d) Repeat the step described in (c) until the parent of the new element is less than or equal to the new element. The minheap invariants have been restored.

Binary heap dequeue (delete + “trickle down”)

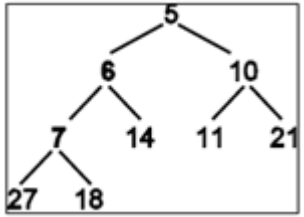


Size=9, Capacity=15

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

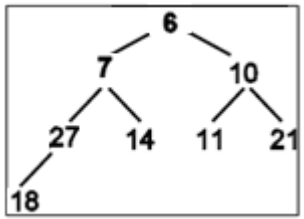


Binary heap dequeue (delete + “trickle down”)



Size=9, Capacity=15

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

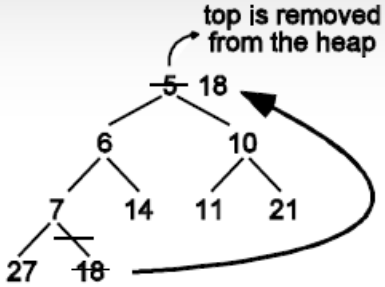


Size=8, Capacity=15

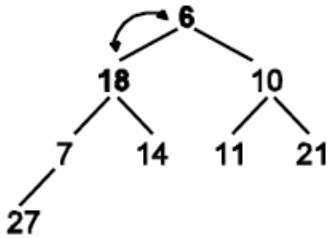
0	1	2	3	4	5	6	7	8	9	…	14
6	7	10	27	14	11	21	18	?	?	…	?

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

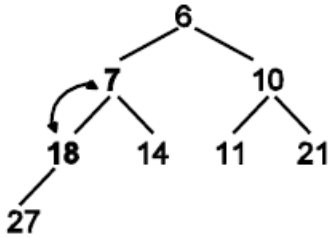
(a) Moving the rightmost leaf to the top of the heap to fill the gap created when the top element (5) was removed. This is a complete binary tree, but the minheap ordering property has been violated.



(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.



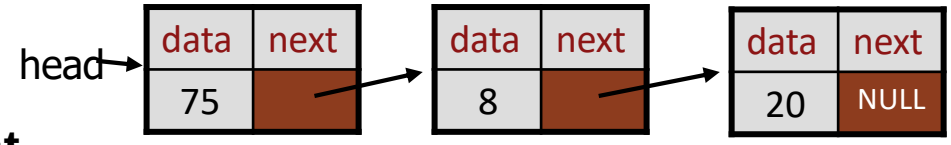
(c) Last swap. The heap is fixed when 18 is less than or equal to both of its children. The minheap invariants have been restored



Summary analysis

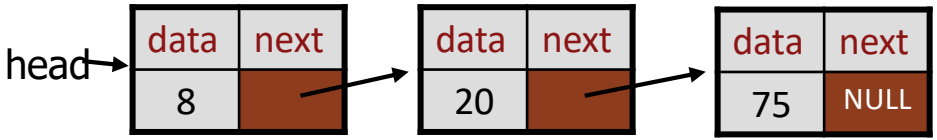
Comparing our priority queue options

Some priority queue implementation options



Unsorted linked list

- Insert new element in front: O(1)
- Remove by searching list: O(N)



Sorted linked list

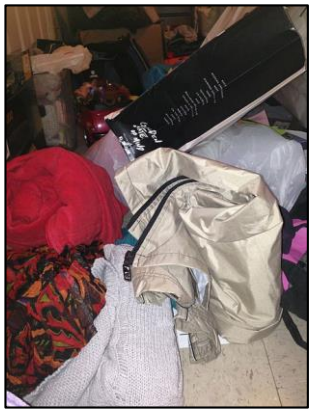
- Always insert in sorted order: O(N)
- Remove from front: O(1)

Priority queue implementations

We want the best of both

Fast add AND fast remove/peek

We will investigate trees as a way to get the best of both worlds



Fast add

+

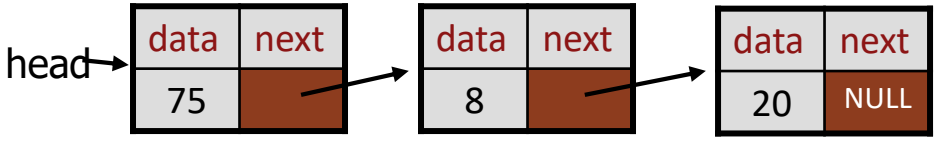
Fast remove/peek

=

Review: priority queue implementation options

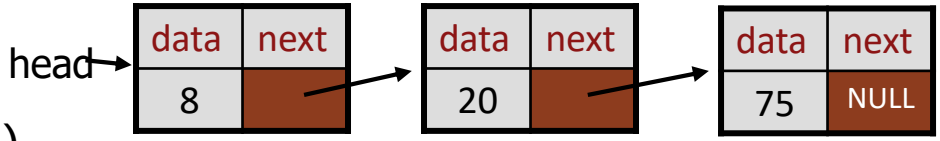
Unsorted linked list

- Insert new element in front: O(1)
- Remove by searching list: O(N)



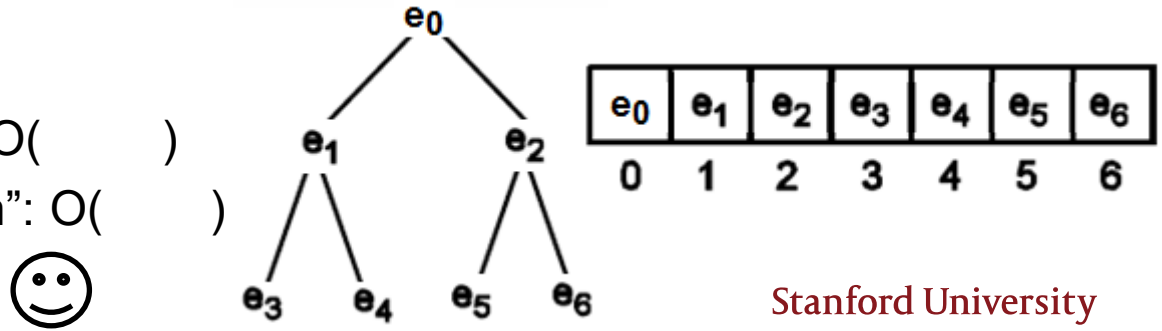
Sorted linked list

- Always insert in sorted order: O(N)
- Remove from front: O(1)



Binary heap

- Insert + "bubble up": O( )
- Delete + "trickle down": O( )



Binary Search Trees

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

Implementing Map interface with a Binary Search Tree (BST)

- Binary Search Trees 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

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
  - **Move 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 memory, so it doesn't have to move things over

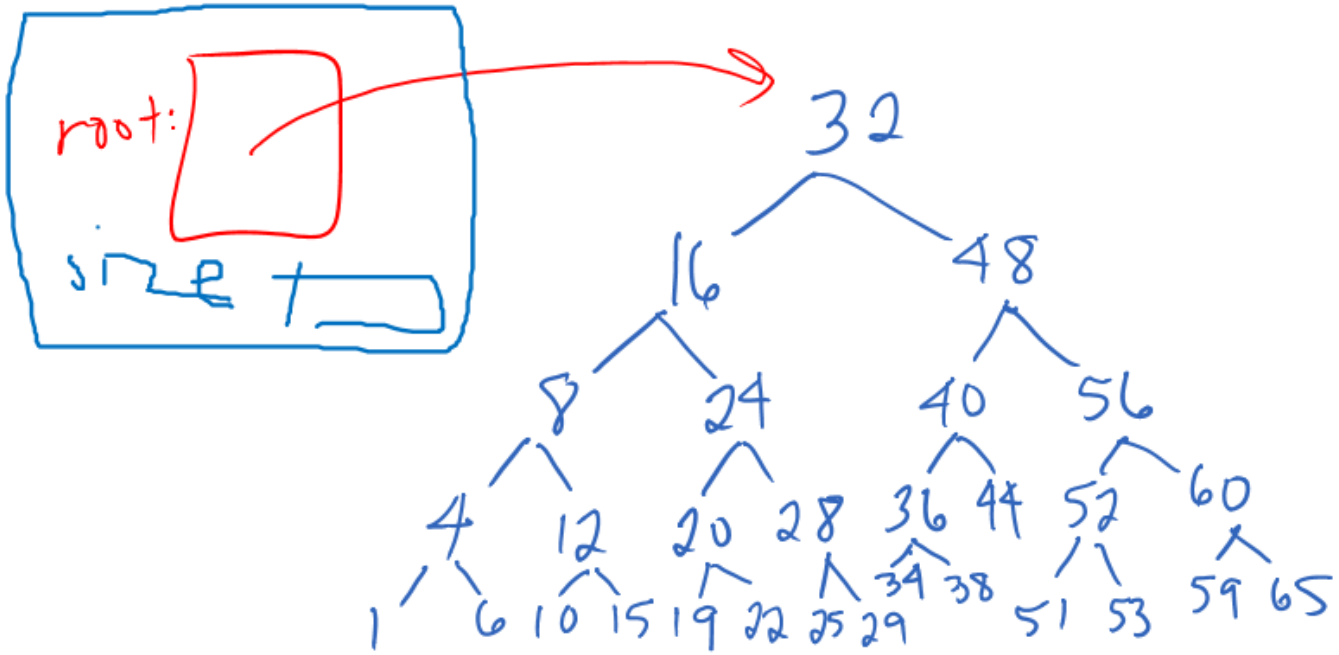
Q. Can we do binary search on a linked list to implement a quick insert?

A. No.

- The nodes are spread all over 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(logn) or O(1)

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

An Idealized Binary Search Tree



TreeMap

This is basically the same as Stanford Map

tree-map.h

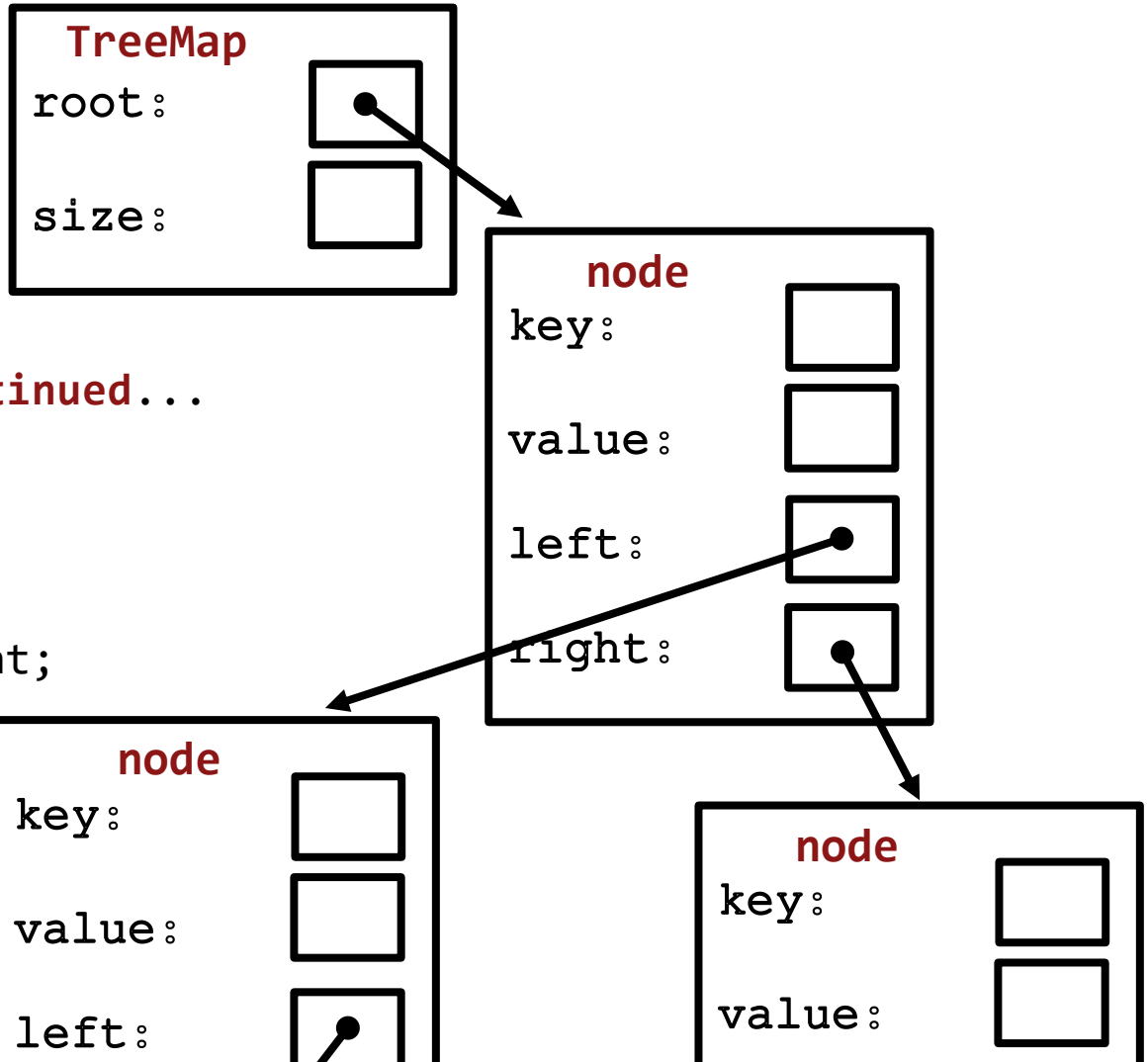
```
template <typename Key, typename Value>
class TreeMap {
public:
  TreeMap();
  ~TreeMap();

  bool isEmpty() const;
  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

```
// class TreeMap continued...
private:
  struct node {
    Key key;
    Value value;
    node *left, *right;
  };
  int size;
  node *root;

};
```

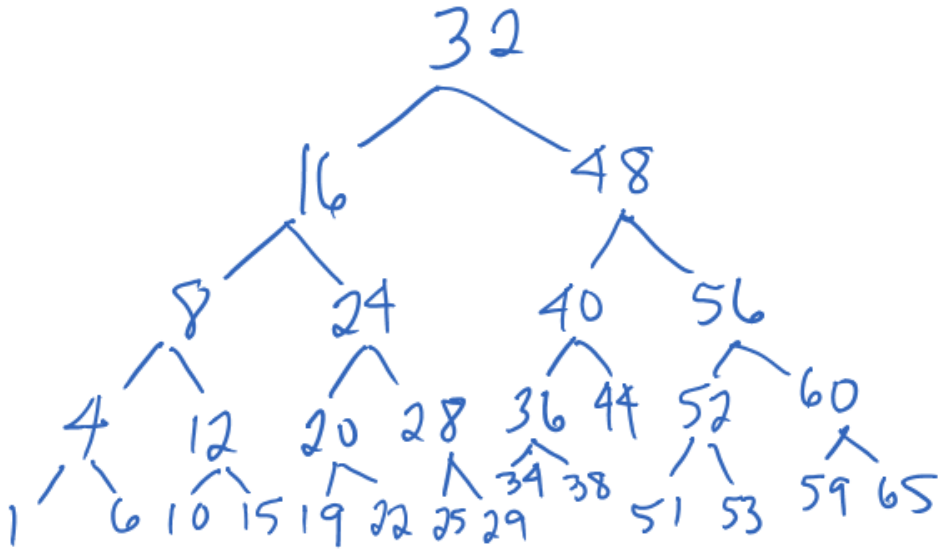


BST put()

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)

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

Question about our put() algorithm:

Pretty simple!

- If key > node's key
  - Go right!
- If key < node's key
  - Go left!

Q. What do we do if the key is equal to the node's key?

Stanford Map example:

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

Performance analysis:
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 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! ☹ ☹

Ok, so, long-chain BSTs are bad, should we worry about it? [math puzzle time]

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 (height equals number of nodes) for a tree with the 4 nodes listed above?

A. 2-3
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).

Midterm Questions?

Are you nervous? What can I do to help?