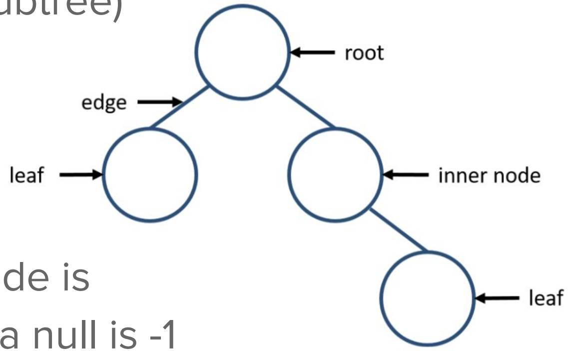
Binary Search Trees, Heaps, SkipLists

1332 Recitation: Week of May 25th

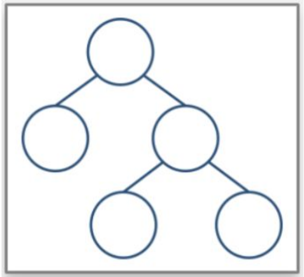
Trees Intro

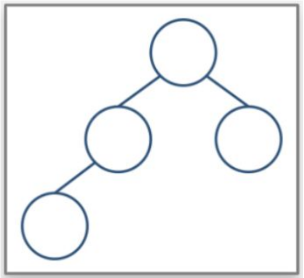
- ADT where nodes have references to child nodes
- Recursive in structure (each node is the “root” of a smaller subtree)
- An “internal” node is one that has children
- An “external” node or “leaf” does not have children
- *Binary Tree*
    - Each node has <= 2 children
- *Depth*: how many edges/references away from the root a node is
- *Height:* max(left child height, right child height) + 1, height of a null is -1
    - This means leaves have height 0

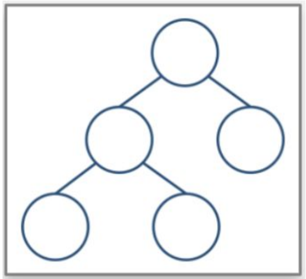


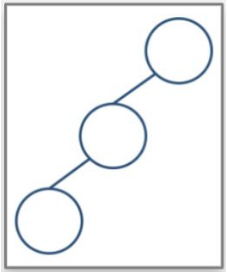
Tree Shape Properties

- *Full*: all internal nodes (nodes with children) have the *maximum* number of children
- *Balanced:* the height of the tree is O(log n)
  - Becomes a more important concept later, will be redefined later
- *Complete:* all levels are completely filled in, except the last level, which is filled from left to right
  - A complete tree is always balanced
- *Degenerate:* each node has appr. 1 child, height of tree is O(n)


Full tree


Complete tree


Full & complete tree


Degenerate tree

Binary Search Trees!

- A binary tree with an *order property*, which requires data to be Comparable
    - All data to the left of a node is < node.data
    - All data to the right of a node is > node.data
    - No duplicates in this class
- To search:
    - Start with current = root
    - If current == null, data not found
    - If data < current, recursive call on current.left
    - If data > current, recursive call on current.right
- If the BST is balanced, search is O(log n), since there are O(log n) levels
- If the BST is imbalanced/degenerate, search is O(n), since there are O(n) levels

Pointer Reinforcement!

- A way to recursively <u>modify</u> the tree without having to “look ahead”
  - Remember LL? We had to stop on the node *before* the index
  - Look-ahead is very ugly and unclean for BST operations (please don’t do it)
  - If you learn pointer reinforcement now, your HW 5 will be mostly copy-paste
- How does it work?
  - Method Signature: **BSTNode foo(BSTNode curr, ...);**
    - Takes in a node, *returns* a node
  - When making a recursive call to the left, set **curr.left = foo(curr.left, ...);**
  - The node you return at any given recursive call is the **new root** of its subtree
    - This means **return curr;** makes no change to the structure (it *reinforces* the pointer), and returning a different node changes the structure
  - In the public method, you have to reinforce the root: **root = foo(root, ...);**

BST Add

- We always add to the very bottom of the tree, creating a new leaf node
- Basically just a search, but with pointer reinforcement:
    - If data < curr, **curr.left = addR(curr.left)**, then return curr
    - If data > curr, **curr.right = addR(curr.right)**, then return curr
    - If data == curr, then we found a duplicate, so just return curr
    - When curr == null, we have found the spot where the new node should go
        - *Return a new node* with the data
            - Note that **return curr** would return null, which was already the parent's child
            - But **return newNode** changes the parent's reference from null to newNode
- Example!

BST Remove

- See “Snowcitation” video from Spring 2019 for implementation details: ***bit.ly/2m4iWSV***, also available on Piazza on Guide to Success & Resources in CS 1332
- Traverse to find data, similar to add. Once found, there are 3 cases
  - 0 children: just return null
  - 1 child: return the **non-null** child
  - 2 children: replace curr’s data with the predecessor/successor and remove the pred/succ
    - 5, 10, 25, 39, 50, 51, 99, 101; 25 is 39’s predecessor, 50 is 39’s successor
    - To find and remove the pred/succ, have another pointer-reinforced helper:
      - The pred. can be found by traversing left once, then right until curr.right is null
      - The succ. can be found by traversing right once, then left until curr.left is null
      - Once found, store the data in a dummy node, and return the pred/succ non-null child to remove the pred/succ

BST Traversals

- Two categories: breadth-first and depth-first
- *Breadth-first:* levelorder(), requires a queue and a while-loop
- *Depth-first:* preorder(), inorder(), and postorder(), requires recursion

```
levelorder():
    if root not null, put root in queue
    while (queue not empty):
        node <- queue.dequeue()
        visit node
        enqueue all node's non-null children
```

```
preorder(curr):
    if curr not null:
        visit curr
        preorder(curr.left)
        preorder(curr.right)
```

For inorder(), go left, then visit, then go right
For postorder(), go left, then go right, then visit

Heaps!

- A binary tree backed by an array
- Two types, both only care about one element at any given point:
    - MinHeap always keeps track of the smallest value
    - MaxHeap always keeps track of the greatest value
- Properties:
    - SHAPE: Always **complete** (all levels filled except last level, which is filled from left to right)
    - ORDER: Each element is less/greater than both its children
    - Note that it is easier to “fix” a broken order property than a broken shape property, so we will prioritize keeping completeness when doing operations

Array-backed tree? How?

- The completeness (lack of gaps) allows us to easily use an array to model their structure
- arr[0] is left empty for the sake of index arithmetic
- arr[1] is the root, fill indices in sequentially in level-order
- For any element at index i:
  - Its left child is at index (i * 2)
  - Its right child is at index (i * 2 + 1)
  - Its parent is at index (i / 2)

Heap Add

- Put the new element at the end of the array, then upheap()
- upheap()
  - Compare element, arr[i], to its parent -- if order property is not satisfied, then swap the element with its parent (swap arr[i] with arr[i/2])
  - If a swap occurred, then repeat on the parent index (set i = i/2 and repeat)
  - As soon as order property is satisfied, no more comparisons should occur.
- O(log n) -> max of 1 swap per level, and since the tree is complete, it is always balanced

Heap Remove

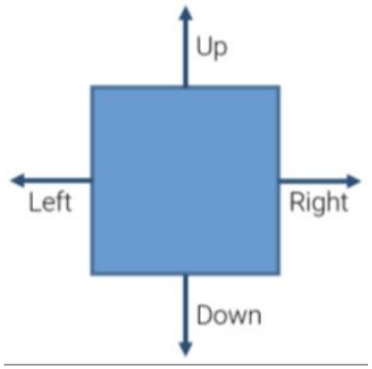
- Remember, we only care about the min/max at the root! This is the only element we will ever remove.
- Store the root (arr[1]) to return, and then replace that index with the very last element
- Now downheap():
  - Compare the element to its children -- if the order property is not satisfied, then a swap must occur. Swap with the smaller child in a MinHeap and the larger child in a MaxHeap
  - Repeat until the order property is satisfied.
- O(log n) -> max of 1 swap per level, and since the tree is complete, it is always balanced

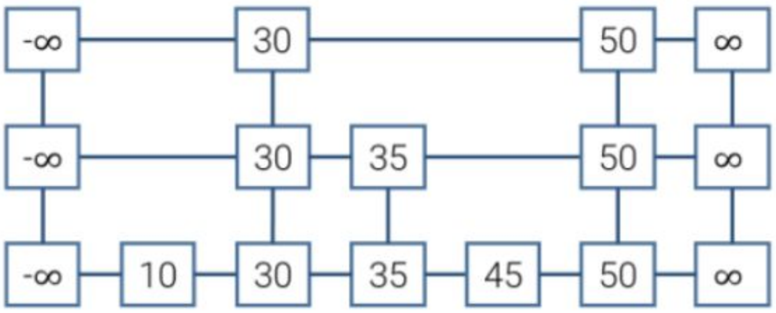
BuildHeap Algorithm

- When constructing a heap, you could add one-by-one
  - Add costs O(log n), repeated n times -> O(n log n)
- We can actually do better, using the BuildHeap algorithm
  - Put all the elements in the backing array in their current order
  - Start at the last internal node (i.e. the node closest to the back that still has children, located at index (size / 2))
  - for (i = size / 2; i > 0; i--) { downheap(i); }
- Due to some sequence/series math, this algorithm is actually O(n), much better than O(n log n)

SkipLists!

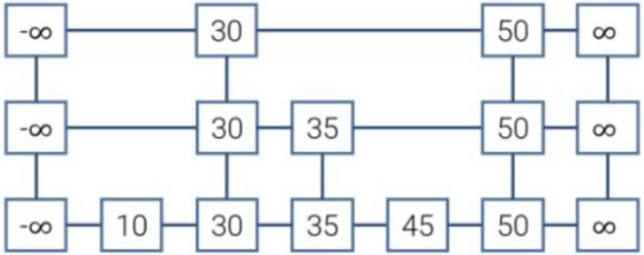
- A different data structure that uses probability to decide its structure
- Basically a bunch of DLLs stacked on top of each other
  - Nodes have left/right, like prev/next in a DLL
  - Also have up/down to connect up the DLLs
  - node.up.data == node.down.data == node.data
- Each level contains a subset of the data of the data below it
- We choose what data goes on what level using coin flips





SkipList Search

- Top-left infinity node is our entry point to the list
  - Check curr.right:
    - if less than data, move curr = curr.right
    - else, drop down a level and repeat
  - If we hit null, the data isn't found
- Ideally, we have n data on the bottom level, then n/2 on the next level, then n/4, etc., which makes us able to skip half the data at each level, resulting in O(log n) time



SkipList Add

- When adding, we search for the spot where the element *should* be
- All elements go on the bottom level
- The number of times we promote an element depends on the number of times we flip a coin before hitting a “tails”
    - Heads = promote
    - Tails = stop promoting and place the new nodes
- If there aren’t enough levels, then add a new one
    - We typically cap the number of levels at appr. log(n), so that the worst-case space complexity is O(n log n)

SkipList Remove

- One of the few structures where removing is easier than adding
- Simply traverse to the data, and remove all instances of the data by using the up/down pointers in the nodes