9.1 priority queues

Tuesday, November 1, 2022

TODAY:

- review AVL trees
- review amortized analysis
- priority queue ADT
- heaps

REVIEW

Q: What's the difference between an AVL tree and a BST?
An AVL tree is a type of BST with the additional invariant that every node has children whose heights differ by at most 1.

Q: How many rotations are needed in an AVL tree to rebalance after an insert or remove operation?
At most 2 at any particular node.
We must check every node along the path to the root, $O(h)$.

Q: How tall is a BST with n nodes? max n
min $\log_2(n)$

How tall is an AVL tree with n nodes? height $O(\log_2(n))$

AMORTIZED ANALYSIS

- worst case might be slow but rare,
  so giving worst case analysis runtime can be overly pessimistic
- amortized analysis considers the running time over a SEQUENCE of operations

ex: inserting into ArrayList

We do n inserts each taking $O(1)$ work,
then 1 insert taking $O(n)$ work.

total work: $n \cdot O(1) + 1 \cdot O(n) = O(n)$ ← recall there can be a hidden constant here e.g. $2n$ is $O(n)$

number of insert operations: $n+1$

amortized cost of each insert $= \frac{O(n)}{n+1}$ is $O(1)$.

This is why we said ArrayList insert is amortized $O(1)$.

Note: "average" is a special technical word, and is different from "amortized".

PRIORITY QUEUES

- type of queue where every item has an associated priority
- queue is ordered from highest to lowest priority (max priority queue)

priorities do not need to be unique

examples:
- homework to-do list (assignments due sooner have higher priority than assignments due later)
- airplane boarding
- elevator call buttons?
- shipping packages
- emergency room triage
- CS course lotteries

priority queue ADT

templated on priority type P and value V

(P: usually numeric; V: can be any type)

```
void insert(P priority, V value)
V remove()  // removes highest-priority item and returns value
V peek()  // returns highest-priority value
P peekPriority()  // returns highest priority
bool isEmpty()
int getSize()
```

example:

```
pq -> insert(3, "BST lab");
pq -> insert(18, "seminar paper");
pq -> insert(11, "Spanish worksheet");
pq -> insert(11, "math hw");
cout << pq -> peek() << endl; // prints "seminar paper"
cout << pq -> peekPriority() << endl; // prints "18"
result = pq -> remove();
```

```
cout << pq -> peekPriority() << endl; // prints "18"
result = pq -> remove();
cout << pq -> getSize() << endl; // prints "3"
cout << pq -> peek() << endl; // prints either "Spanish worksheet" or "math hw"
```

How to implement the priority queue ADT to make it efficient?

operation	sorted array	
void insert(P, V)	O(n)	find the place to add this item, shift everything later down by 1
V remove()	O(1)	
V peek()	O(1)	
P peekPriority()	O(1)	

We can go directly to the last item in the array.

IDEA: Keep the highest-priority item as the last value in the array. Keep the entire array in sorted order, lowest to highest priority.

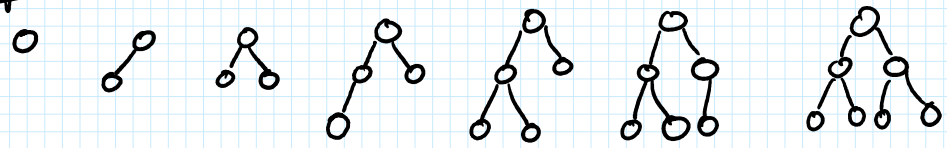
↳ sorted LL has the same runtimes

How can we be more efficient?

COMPLETE TREE

- a binary tree
- every level of the tree is full, except the final level which is filled from left to right
- n nodes ⇒ height $O(\log_2(n))$

examples:

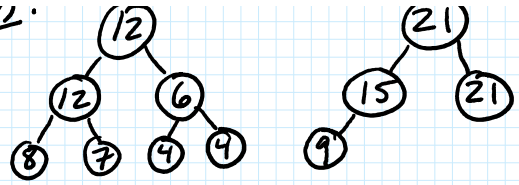


HEAP

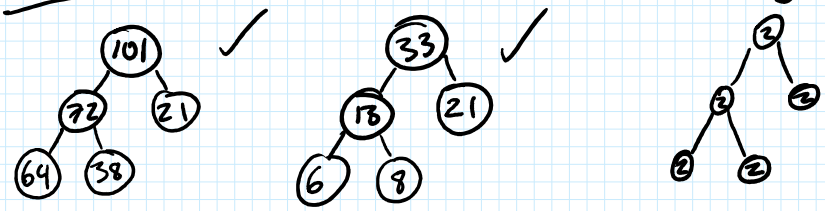
- a complete binary tree
- invariant: for every node in the heap, that node's priority ≥ priorities of all its descendents

examples:

example:



practice: Create a heap of size 5, using any priorities.



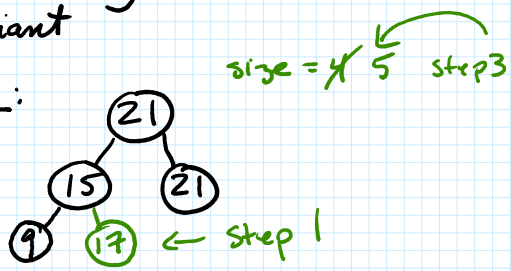
heap insert: must make sure it's a complete binary tree and satisfies the heap invariant

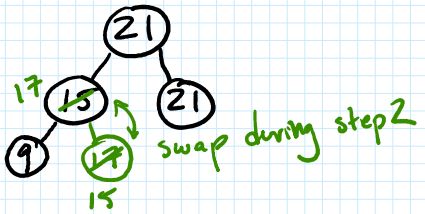
algorithm

1. Add the new element in the next open spot on the complete tree.
2. Fix the heap by bubbling up.
3. Increment size.

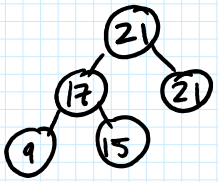
example:

insert(17)





final heap:




```
bubble up(node):
    if current node is root
        return
    if parent's priority < node's priority
        swap node and parent contents
        bubble up (parent)
```

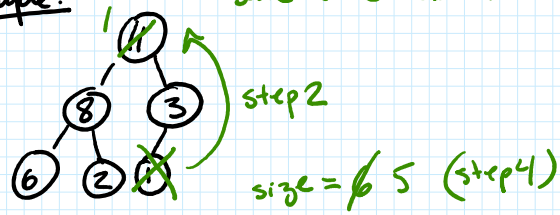
heap remove: must make sure it's a complete binary tree and satisfies the heap invariant

algorithm

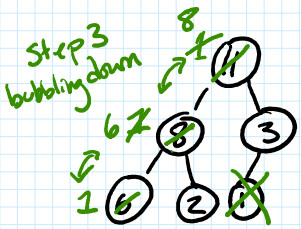
1. Save the value at the root.
2. Replace root data with data from last (rightmost leaf) node. (Also remove that node.)
3. Fix the heap by bubbling down.
4. Decrement size.

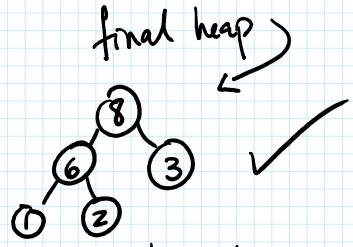
example:

step1: save value associated with 11



3. Fix the heap by bubbling down.
4. Decrement size.
5. Return saved value.



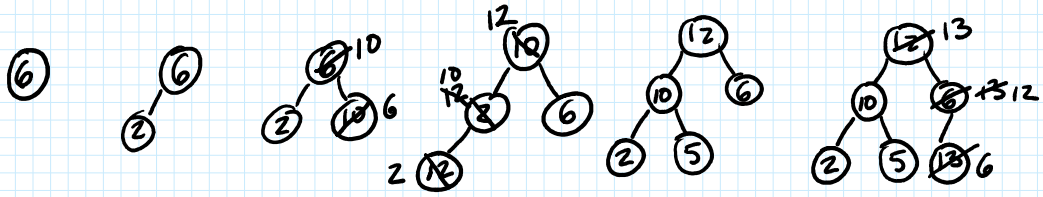


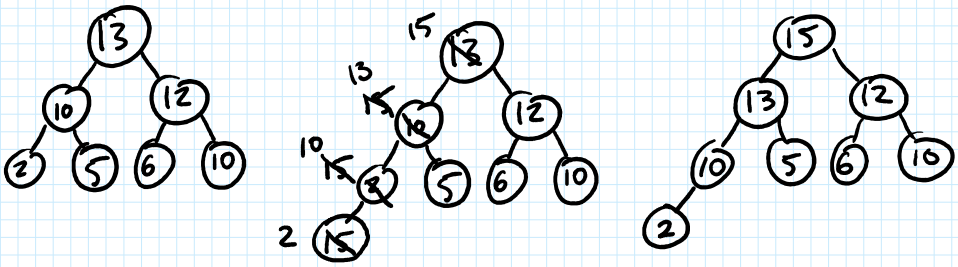
returns the value saved which was associated with 11

```
bubbledown(node):
  if current node is leaf:
     return
  if current's priority < max(child priorities)
     swap node and its max priority child contents
     bubbledown( that child)
```

Practice: Insert the following priorities into a heap:

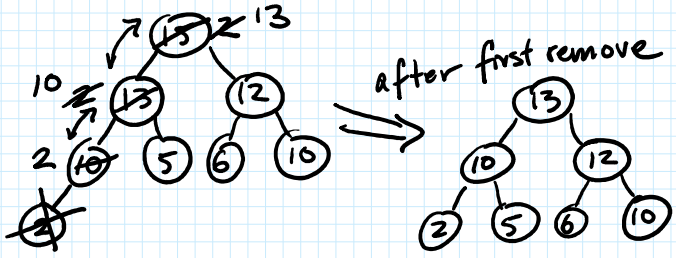
6, 2, 10, 12, 5, 13, 10, 15 (in that order)



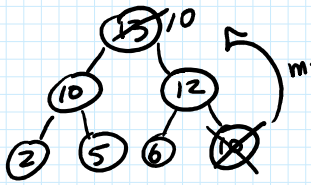


Next let's do 3 removes:

first remove: should return value associated with 15



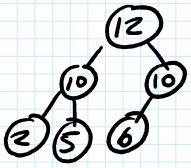
second remove: should return the value associated with 13



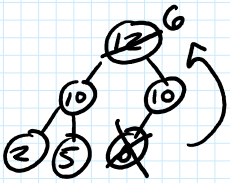
moves 10 to root

bubble down will swap 10 & 12
then 6 ≤ 10 so it stops

after second remove:



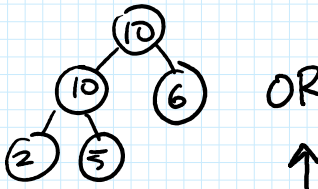
third remove: should return value associated with 12



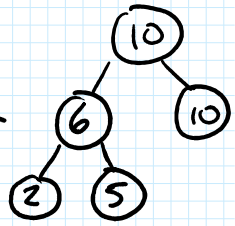
moves 6 to root

bubble down will swap 6 & 10
(either 10 is fine since they're tied)

after third remove:



OR



so there are 2 possible heaps after the bubble down... both are fine!

Sneak peek of next time:

How to represent heaps so that these operations are fast?

- bubble up
- bubble down
- insert at bottom right
- lookup parent/children

We'll use ... an array!