Math 182 Lecture 10

§5.3 Priority Queues (Heaps cont.)

Example Have a set of jobs each has a priority value, need to keep track of which job has highest priority.

Two types of priority queues:
- max-priority queue ← will focus on this
- min-priority queue

Priority queue operations:

- INSERT(S, x), which inserts the element x into the set S, which is equivalent to the operations $S = S \cup \{x\}$.
- MAXIMUM(S), which returns the element of S with the largest key.
- EXTRACT-MAX(S), which removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k), which increases the value of x's key to the new value k, which is assumed to be at least as large as x's current key value.

Priority queues can be implemented as a max-heap

Assume A is a max-heap

```
HEAP-MAXIMUM(A)
1  return A[1]
```

$\Theta(1)$

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

$\Theta(1)$

line 6: $\Theta(\lg n)$ ← $n = A.heap\text{-}size$

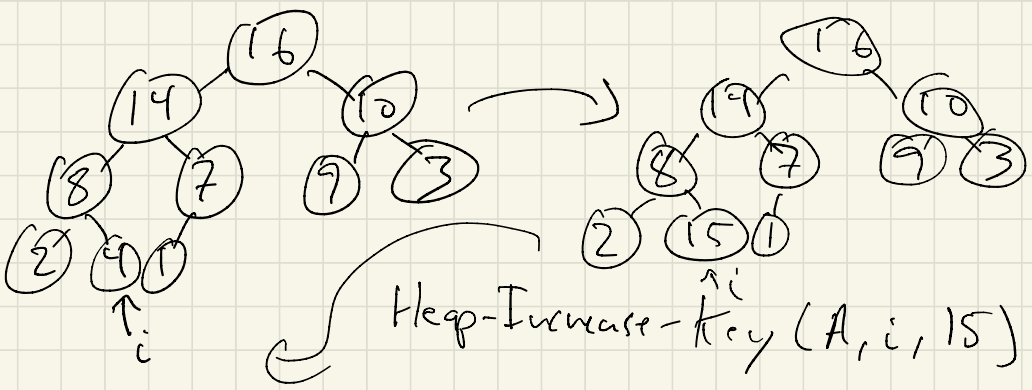
$O(\lg n)$

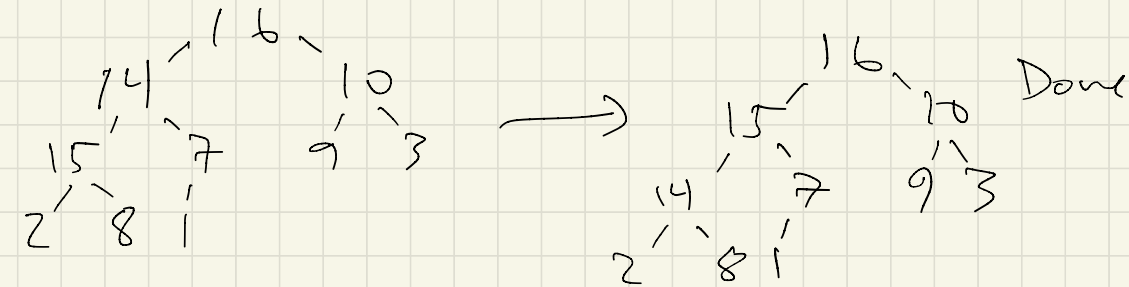
lines 1-2 check if heap is empty
o/w line 3 set max = A[1]
lines 4-5 set root to A[A.heap-size] and reduces heap-size by 1.
line 6 remake heap into max-heap again.
Then return x.

$O(\lg n)$

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2        error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        exchange A[i] with A[PARENT(i)]
6        i = PARENT(i)
```

bubble up key until it is in correct place.





Summary: changed 4 to a 15, then "bubbled up" the 15 through the heap until it gets to a place w/ max-heap property.

MAX-HEAP-INSERT(A, key) $O(\lg n)$.

1 $A.heap\text{-}size = A.heap\text{-}size + 1$ $\Big\}$ $\Theta(1)$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$ $O(\lg n)$

line 1: Increase heap size by 1
line 2: Assign $A[A.heap\text{-}size] = -\infty$
which guarantees max-heap property,
then increase to key, so
Heap-Increase-Key figures out where
it should go in the heap.

§5.4 Stacks and queues

Abstractly, stacks and queues are sets which support the following operations:

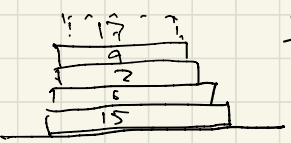
- Insert(S, x) inserts new key x into S
- Delete, which ~~deletes~~ removes and returns a prespecified element from S.

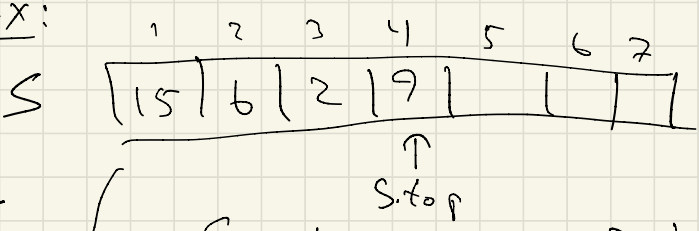
Stacks

Slogan: last-in, first-out (LIFO)

Stacks are implemented as an array S w/ attribute $S.top$ stack = $S[1..S.top]$

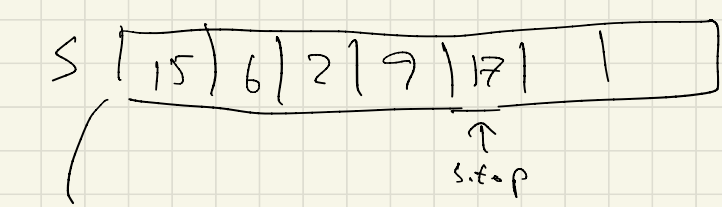
Ex:



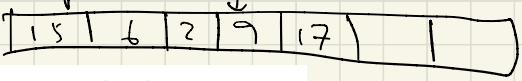


for stacks, say Push instead of Insert and Pop instead of Delete

Push(S, 17)



↳ Pop(S) stop return 17.




```
STACK-EMTPY(S)
1  if S.top == 0
2        return TRUE
3  else return FALSE
```

Checks if S is empty

O(1)

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

Return top of stack and delete it

O(1)

Lines 1-2, check if stack is empty.

Lines 3-4, reduce stack size by 1, return old stack top.

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

```
Push(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

$\Theta(1)$

If stack is full ($S.top = S.length$) and you try to Push(S, x) Then you get stack overflow error.

Queues Slogan:

first-in, first-out (FIFO)

A queue is like a stack except the oldest element is the one to be deleted.

(Insert(Q,x)) Enqueue(Q,x) insert x into back of the queue

(Delete(Q)) Dequeue(Q) deletes the element at front of queue and returns it to us.

ENQUEUE(Q, x)

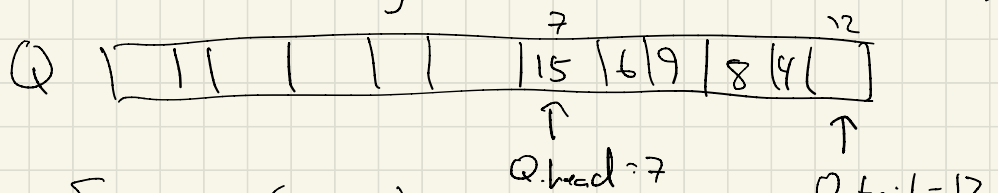
1 $Q[Q.tail] = x$
2 **if** $Q.tail == Q.length$
3 $\quad Q.tail = 1$
4 **else** $Q.tail = Q.tail + 1$

$\Theta(1)$

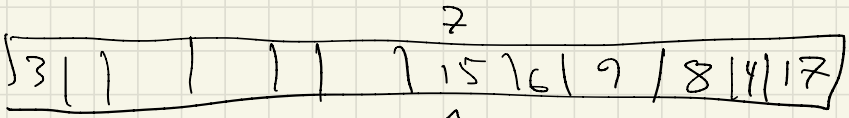
Queues have two attributes:

Q.head: first element in queue (next to be deleted)

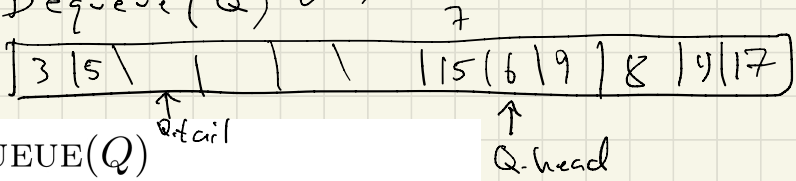
Q.tail: location where next insertion should go (after the newest element)



Enqueue(Q, 17)
Enqueue(Q, 3)



Enqueue(Q, 5)
Dequeue(Q) ⇝ returns 15



DEQUEUE(Q)

1 $x = Q[Q.head]$
2 **if** $Q.head == Q.length$
3 $\quad Q.head = 1$
4 **else** $Q.head = Q.head + 1$
5 **return** x

$\Theta(1)$

lines 1 & 5 return front of queue.
lines 2-4 assign new head of queue.

Remarks In general, need to check also if queue empty/full

(can't figure out what this means for your implementation) in order to avoid overflow/underflow errors.

Other data structures to check out:

(doubly) linked lists

various trees