CS3000

5/19 - Mon.

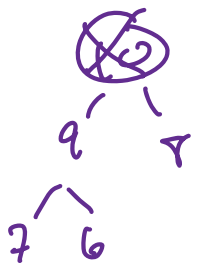
Admin

- Hw2 due tmrw 9pm
- Exam #1 5/22 during class
- APPY today, due 11:30 tmw

Agenda

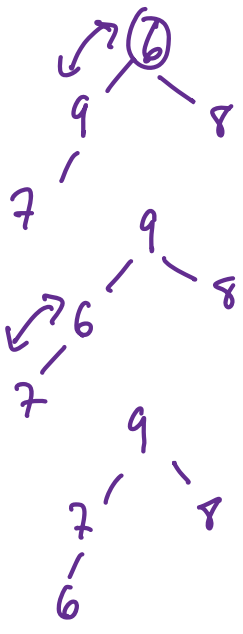
1. Data structures/arrays
2. Queues
3. Stacks
4. APPY

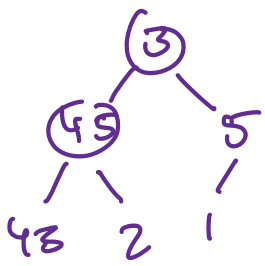
0. Heap Examples
   - remove root
   - heapify



Heapify:

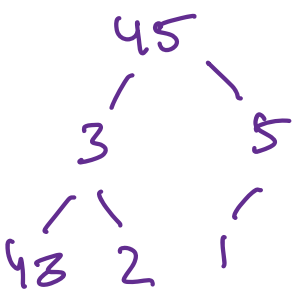
- linear in height of the tree
- for a complete binary tree $\Theta(\lg n)$

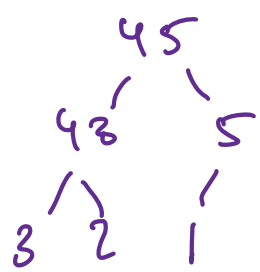




What does it mean to "remove"?

- during heapsort, value A[1] goes to the next sorted output





1. Data structures / Arrays

≠ can we do better? ≠

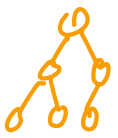
1. D+C
2. change data structure
3. ...

→ Heapsort!

Data Structure (ADT)

- array
- pointers
- other?

$A = \langle \_, \_, \_, \dots, \_ \rangle$



What other structures can have an underlying array?

- Heaps
- Stacks
- Queues

$A = \langle \_, \_, \_, \dots \rangle$

no goal algorithm!
instead, basic operations: insert, remove, find

array starting point

$\langle \_, \_, \_, \_ \rangle$

$\langle \_, \_, \_, \_ \rangle$

Find

- $\Theta(n)$ unsorted
- $\Theta(\lg n)$ sorted

Assumptions:
- fixed size
- track current "next" element (filled in so far)

let $A[1..n]$ be a new array

$A \langle \_, \_, \_, \dots, \_ \rangle$
$1\ 2\ 3 \quad k \quad n$

Insert

unsorted
- insert element in "next" position

$\langle \overline{1}, \overline{2}, \overline{3}, \dots, \overline{k}, \dots, \overline{n} \rangle$

- $A[k]$ = new element
- increment k

$\Theta(1)$

sorted
- maintain the sorted order
- find position where new element belongs

$A = 1, 4, 5, 10, 13 \mid \dots n$

- insert 3 at position 2

$A = 1, 3, 4, 5, 10, 13, \mid \dots n$

Run time: find position lgn
insert, shift n
$\Theta(n)$

Remove

↳ once found, (assumption)

unsorted

ex $\langle 7, 13, 5, 6, 2 \rangle$

- remove 5
  - shift left $\langle 7, 13, 6, 2, 2 \rangle$ $\Theta(n)$
  - update k

- what if we don't care about maintaining order?
  - swap 2 into 5's position
  - update k $\Theta(1)$

$\langle 7, 13, 2, 6, 2 \rangle$

sorted

ex $\langle 2, 5, 6, 7, 13 \rangle$

- remove 5
  - shift left $\langle 2, 6, 7, 13, 13 \rangle$
  - update k

$\Theta(n)$

Run-times (array)

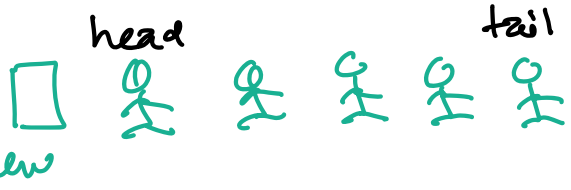
- find $\Theta(n)$ or $\Theta(\lg n)$
- insert $\Theta(1)$ or $\Theta(n)$
- remove $\Theta(n)$, $\Theta(1)$ or $\Theta(n)$

unsorted sorted

2. Queues

↳ underlying array  
waiting in line

 FIFO First In, First Out!

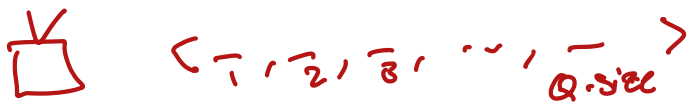


- head = front of queue
- tail = back of queue

Operations:
- insert: ENQUEUE (put new item at back of line)
- remove: DEQUEUE (remove first item from front of list)

Assumptions:
- never run out of space
- but, we can wrap!
- track head, tail (Q.head, Q.tail)
- don't shift!
- Q.size is max size

Questions:
- what changes in the queue?
- what happens to Q.head?
- what happens to Q.tail?



Q.size = 3

ENQUEUE(Q, x)
- increment Q.tail
- put x at position Q.tail
- wrap?

DEQUEUE(Q)
- return item at Q.head
- increment Q.head
- wrap?

ENQ(ACe)

ACe

ENQ(7C)

ACe, 7C

ENQ(7D)

ACe, 7C, 7D

DEQ()
- return ACe

7C, 7D

ENQ(KC)

KC, 7C, 7D

last next

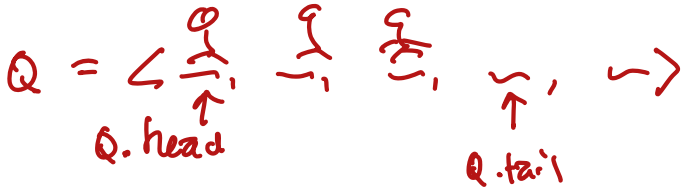
12:51

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.size
3        Q.tail = 1
4  else
5        Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.size
3        Q.head = 1
4  else
5        Q.head = Q.head + 1
6  return x
```

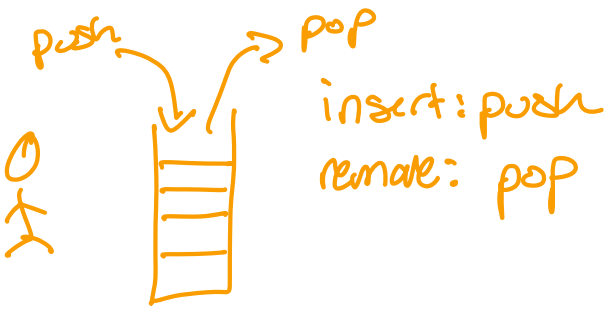
enqueue: $\Theta(1)$

dequeue: $\Theta(1)$



3. Stacks

↳ last in, first out (LIFO)



insert: push
remove: pop

- underlying array
- don't care about find

underlying array A = < 1, 2, 3, ... > S.size

- S.size = max size of stack
- S.top = current top

(no going over the size)

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

```
Pop(S)
    x = S[S.top]
    S.top = S.top - 1
    return x
```

(wide thing!) ↳

- error if overflow
- check for empty

```
PUSH(S, x)
1  if S.top == S.size
2        error "overflow"
3  else
4        S.top = S.top + 1
5        S[S.top] = x
```

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

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

APPLY!