CSC263

Sanjana Girish

CSC263 Winter 2022

Data Structures and Analysis

Michelle Craig and Samar Sabie

You can download a copy of the unannoted slides from our 263 Quercus → Lectures page and annotate as we go.

Marking Scheme

Weekly Quercus Modules (11)	11 %	Each worth 1%.	Completed Individually
Problem Sets (3)	32 %	PS1 (10%) PS2 (10%) PS3 (12%)	Completed Individually or with one partner
Term Tests (2)	27 %	Test 1 (12%) Test 2 (15%)	Completed Individually
Final Exam	30 %	During the exam period	Completed Individually

must get ≥ 40% on final to pass the course

Weekly Quercus Modules

- 11 of these, each worth 1%
- Completed individually on Quercus
- Released Thursday by noon
- Due Tues the following week at noon
- No late submissions accepted
- Unlimited resubmissions; no feedback/grades until after the deadline

Problem Sets

- You may work **individually or in pairs**
- Submissions need to be typed into a PDF file and submitted to MarkUs
- Due dates in syllabus (all at 4pm)
- Late assignments: 0% penalty for the first hour, 5% for the next two hours, 10% up to 3 hours late, etc… (check table on Quercus)

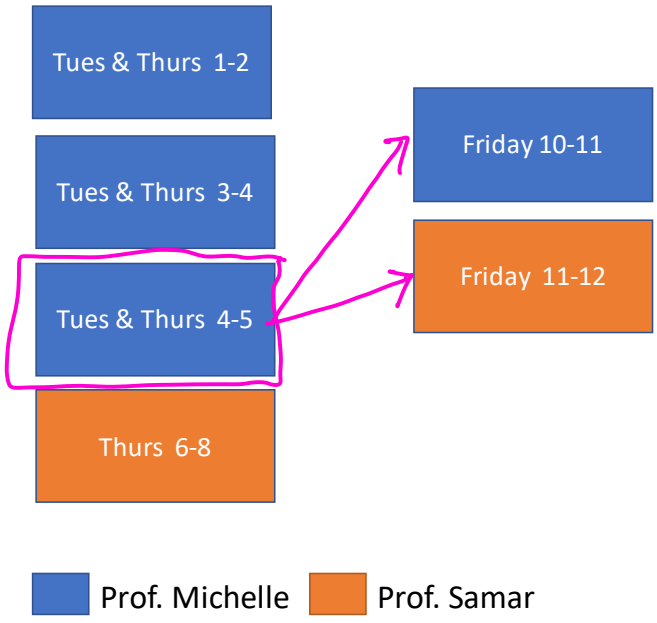
Typing up Problem Sets

- LaTeX is strongly encouraged (but not required)
- You can generate LaTeX files using a web-based tool such as
- https://www.overleaf.com/
- Many tutorials online, e.g.:
- http://www.maths.tcd.ie/~dwilkins/LaTeXPrimer/
- 50-minute workshop recording posted on Quercus > Problem Sets page

Term Tests

- Test 1: Friday Feb 18
- Test 2: Friday March 18
- Hopefully in person during tutorial times ?
- If either test has to be online, we will use MarkUs.

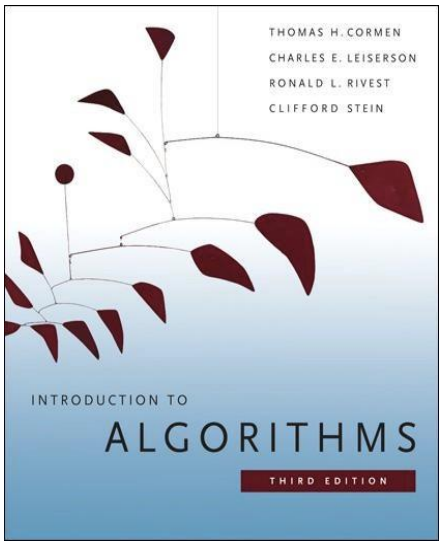
Lectures (3 hours per week)



- You may attend either Friday lecture
- Tues & Thurs lectures will move to in-person once allowed
- Friday lectures will stay on-line all term
- Friday tests will be held in-person if possible

Textbook

- “CLRS”
- Optional
- Available online at UofT library
- We will reference during lectures and Quercus modules and you can find appropriate chapters based on topics.



Online Learning (for a few? Weeks)

- Recordings — one per hour
- Chat — please have it open Yes!!(
- Cameras — YES YES YES !!!

Engagement is key!

From the Welcome Questionnaire

- 95% of you are in the Toronto time zone or +/- 3 hours
- Students from India, UAE, China, Sudan, Bahrain, Greece, Mauritius

- Lots of students excited about learning material for job interviews
- Some students excited to do more proofs – others worried about them
- Some of you are excited about meeting new people (in person)

- As instructors
  - We have coordinated due dates with CSC209
  - (At least while we are online) we will post unannotated slides before class and annotated versions after class
  - We plan to provide one recording per hour

Abstract data types are important for specification, they provide modularity & reuse since usage is independent of implementation

Chalk Talk Review of 148

What is a Data Structure Anyway?

↳ WHAT IS THE DATA AND WHAT CAN WE DO WITH IT? WHAT OPERATIONS?

Abstract Data Type (ADT) = Set of objects together with operations

Ex 1: Objects integers OPS: add(x,y), mult(x,y), x≥y?

Ex 2 Stack Objects pile of things

operations: push(x), pop(), isEmpty()

WHAT

Data Structure = Implementation of an ADT

A way to represent objects and an algorithm for every operation

Stack Implementations.

① linked list

② array

size==0?

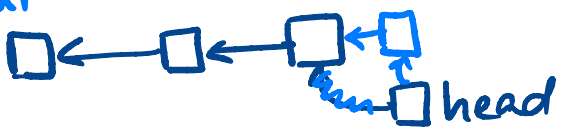
size

HOW

↳ HOW ARE WE STORING THIS? HOW DOES THAT DETERMINE HOW THE ALGORITMS HAVE TO WORK. ALL ABOUT IMPLEMENTATION

STACK ⇒ first in, last out

① linked list ⇒ push



pop

isempty head == null?

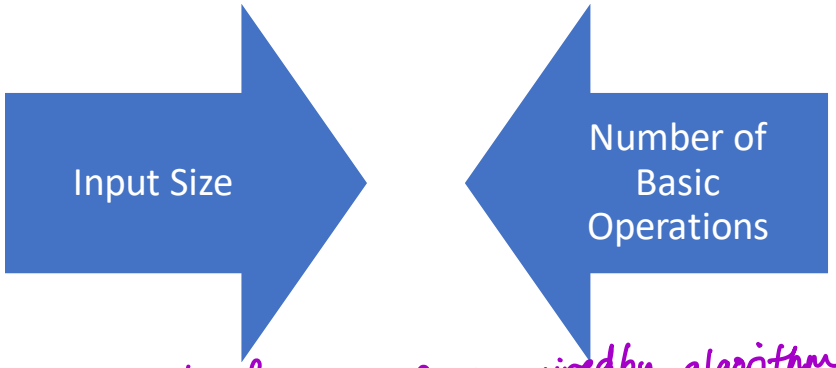
② array ⇒ push

size

isempty size == 0?

Chalk Talk Review of 165

Runtime Complexity Abstractions



Complexity = amount of resources required by algorithm as a function of input size

Resource = running time (memory/space)

Input Size is problem dependent

for list: length of list
for graph: # nodes and # edges
for number: # bits to represent

complexity = amount of resources required by algorithms as a function of input size

resource = running time OR (memory / space)

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

Analyzing Runtime Complexity

Asymptotic Notation >

- Big O
- Big Omega
- Big Theta

Runtime Cases >

- Best case ✓
- Average case → possibly new in 263
- Worst case ✓

Asymptotic Notation (Mathematically)

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

$f(n) = 2n^2 + n$ $O(n^2)$ $O(n!)$ $O(n^n)$ $O(n^{1000})$

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$

$f(n) = 2n^2 + n$ $\Omega(n^2)$ $\Omega(1)$ $\Omega(n)$

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

tight bound

$f(n) = 2n^2 + n$ $\Theta(n^2)$ but not $\Theta(n!)$ $\Theta(n^n)$ $\Theta(1)$ $\Theta(n)$

$\Theta(n^2)$ ↓ tight bounds

Chalk Talk Review of 165

Asymptotic Notation (Intuitively)

Big-Oh: $f(n) = O(g(n))$ means that the function $f(n)$ grows slower or at the same rate as $g(n)$

Big-Omega: $f(n) = \Omega(g(n))$ means that the function $f(n)$ grows faster or at the same rate as g(n)

Big-Theta: $f(n) = \Theta(g(n))$ means that the function $f(n)$ grows at the same rate as $g(n)$

Asymptotic Notation (Practically)

In practice for worst case:

Big-O (upper bound): argue that the algorithm executes no more than *cg(n)* steps on any input of size *n*.

Big-Omega (lower bound): exhibit one (one family of inputs (one for each n)) input on which algorithm executes at least *cg(n)* steps.

What about *big-Theta?*

worst case $\Theta(g(n))$

Worksheet Activity

- In Breakout Rooms
- Worksheets posted on Lectures page on Quercus
- Only do questions 1-3 now (fine to just get Q1 done if you get talking today!)
- Communicate via http://tinyurl.com/breakouts-263

Worst-case Analysis

Asymptotic Notation: What is the running time of an algorithm for an input size n?

Worst case analysis: What is the maximum possible running time of an algorithm for an input size n?

$$T(n): \max \{\mathrm{t(x)} : \text{x is an input of size n}\}$$

Best case can never be when n=0

Can count:
1> count the number of times line3 runs
2> Number of loop comparisons

Chalk Talk

Worst-case Analysis

comparisons
input size?
let n = size of list

```
   def hasTwentyOne(L):
1     j = L.head
2     while j != None and j.key != 21:
3        j = j.next
4     return
```

One family of bad input is lists of length n w/ no 21's.
2 comparisons for each element of the list → 2n
1 more that last time when j == None → 1
We need to show that any input of length n will take no more than 2n+1 comparisons. ttl = 2n+1 comparisons.
Each time we execute line 2, we either finish early or we execute line 3. Each time we execute line 3, we move to a new element of L

Chalk Talk

Worst-case Analysis

So we can execute line 3 at most n times.
after that j == None
∴ we can execute line 2 at most n times when we follow it with line 3 and then one more.
At most for any input of size n, we do 2n+1 comparisons.

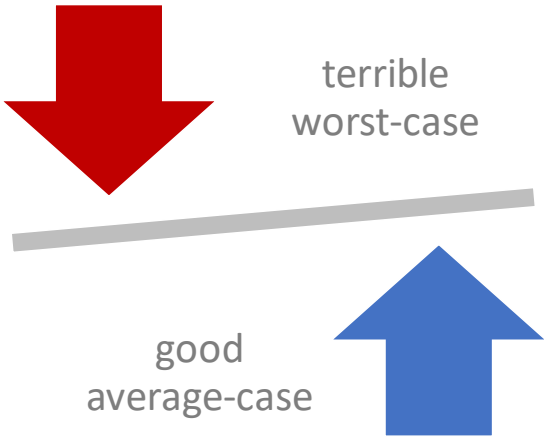
We conclude worst case 2n+1 comparisons
which is $\Theta(n)$

Best case → list of length n where first element is 21
2 comparisons. EMPTY LIST DOESN'T COUNT
(NEED FOR ARBITRARY N LIST)

Worst-case Analysis

- Give a pessimistic upper bound that could occur for any input of a fixed size n as $T(n) = O(f)$. Must be proved on all possible inputs
- Give a family of inputs (one for each input size) and a lower bound on the number of basic operations that occurs for this particular family of inputs as $T(n) = \Omega(f(n))$
- If the two expressions involve the same function, then $T(n) = \Theta(f(n))$

Average-case Analysis



Many Algorithms in Practice

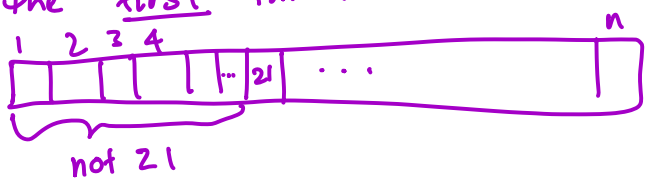
Chalk Talk

size : length of L = n
# of comparisons

```
  def hasTwentyOne(L):
1    j = L.head
2    while j != None and j.key != 21:
3       j = j.next
4    return
```

2K cuz 2 comparisons per iteration

The # of comparisons depends on when we see the 21 for the first time.



if first 21 is at K
2k comparisons
if no 21 at all
2n+1

time $t_n(x)$

Average-case Analysis

		time
X_0	all inputs where no 21 in the list	2n+1
X_1	all inputs where 21 is at the first element	2
X_2	all inputs where first 21 is at position 2	4
⋮	" " " " " " 3	6
	⋮	
X_n	all inputs where first 21 is at position n	2n

prob. distr. to approximate average

For algorithm A, define S_n = sample space of all inputs of size n

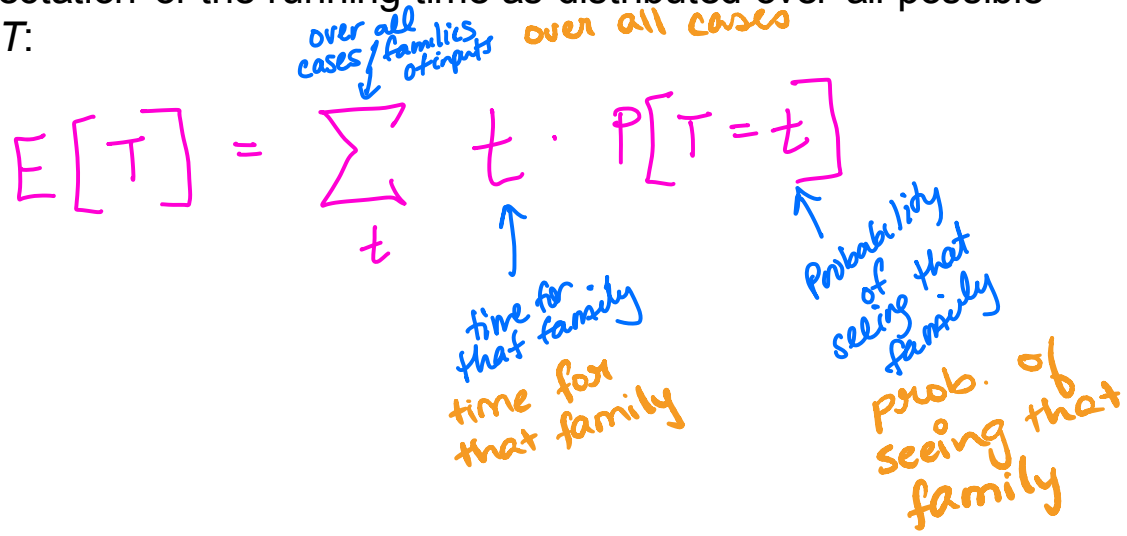
$S_n = \{X_0, X_1, X_2, \ldots, X_n\}$ ← we need a probability dist^n over this set

Let $t_n(x)$ = number of steps executed by A on input x (in S_n)

Let $T_{avg}(n)$: the (weighted) average of the algorithm's running time for all inputs of size n

Average-case Running Time

Is the expectation of the running time as distributed over all possible values of *T*:



Chalk Talk Continues

```
def hasTwentyOne(L):
1     j = L.head
2     while j != None and j.key != 21:
3         j = j.next
4     return
```

$$E[T] = \sum_{\text{all cases}} t \cdot Pr[T=t]$$

$$= \sum_{k=0}^{n} t_n(X_k) Pr(X_k)$$

$$= (2n+1)(Pr \text{ that no 21 in list}) + \sum_{k=1}^{n} 2k \, Pr(\text{first 21 occurs at pos } k)$$

Lets assume each position in the list can be 21 with probability p.

$$p(\text{not a 21 at position}) = 1-p$$

$$= (2n+1)(1-p)^n + \sum_{k=1}^{n} 2k(1-p)^{k-1} p$$

$$= (2n+1)(1-p)^n + \frac{2p}{1-p} \sum_{k=1}^{n} k(1-p)^k$$

$(1-p)^{k-1} p$

$(1-p)^{k-1}$: all the positions in front of k were not 21

p: position k is 21

Handy Summation Formulas

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$ arithmetic series

$$\sum_{k=0}^{n} a^k = \frac{a^{n+1}-1}{a-1}$$ geometric series

$$\sum_{k=1}^{n} ka^k = \frac{a^{n+1}(n(a-1)-1)+a}{(1-a)^2}$$ arithmetic-geometric series

$$E[T] = (2n+1)(1-p)^n + \frac{2p}{1-p}\sum_{k=1}^{n} k(1-p)^k$$

$$= (2n+1)(1-p)^n + \frac{2p}{1-p}\left[\frac{(1-p)^{n+1}(n(1-p)-1)+1-p}{(1-1-p)^2}\right]$$

$$= (2n+1)(1-p)^n + \frac{2p}{1-p}\left[(1-p)^{n+1}\frac{(n-np-1)+1-p}{p^2}\right]$$

$$= (1-p)^n\left[(2n+1) + 2\left[\frac{n-np-1}{p}\right]\right] + \frac{2}{p}$$

More space Chalk Talk Continues

```
def hasTwentyOne(L):
1    j = L.head
2    while j != None and j.key != 21:
3        j = j.next
4    return
```

$$E[T] = (2n+1)(1-p)^n + \frac{2p}{1-p}\sum_{k=1}^{n} k(1-p)^k$$

→ Substitute in from arithmetic-geometric series eq $\stackrel{*}{=}$ with $a = 1-p$

[carefully do a whole bunch of grade 9 algebraic manipulation]

$$E[T] = (1-p)^n\left[1 - \frac{2}{p}\right] + \frac{2}{p}$$

$0 < p \leq 1$

More space Chalk Talk Continues

```
  def hasTwentyOne(L):
1     j = L.head
2     while j != None and j.key != 21:
3         j = j.next
4     return
```

$$E[T] = (1-p)^n \left[1 - \frac{2}{p}\right] + \frac{2}{p}$$

as $n \to \infty$ $\quad (1-p)^n \to 0$ $\quad E[T] \to \frac{2}{p}$ $\quad \therefore \Theta(1)$

Average-case Analysis (Meta) Lesson

1. Define the possible set of inputs (e.g. numbers between 1 and 10) and a **probability distribution** over this set (e.g. uniform)
2. Define the "basic operations" being counted. This is often the operation that happens the most frequently (such as list access when searching one) or the most expensive operation
3. Define the **random variable** T over this probability space to represent the running time of the algorithm.
4. Compute the expected value of T, using the chosen probability distribution and formula for T.
5. Since the average number of steps is counted as an exact expression, it can be written as a Theta expression.

Worksheet

let length of lst ⇒ n

count array accesses

```
1  def evens_are_bad(lst):
2      if every number in lst is even:
3          repeat lst.length times:
4              calculate and print the sum of lst
5          return 1
6      else:
7          return 0
```

n accesses

n^2 (repeat lst.length times) × n (calculate and print the sum of lst)

total runtime : $n^2 + n$

Chalk Talk

```
1  def evens_are_bad(lst):
2      if every number in lst is even:
3          repeat lst.length times:
4              calculate and print the sum of lst
5          return 1
6      else:
7          return 0
```

Sample Space?

$S_n = \{$ all possible lists of length $n \}$

$S_n = \{\{$ all lists of length n that have all even #'s $\}, \{$ all lists of length n that have at least one odd $\}\}$

$T = n + n^2$

$T = n$

$E[T] = (n+n^2)P(\text{list is all even}) + nP(\text{list has at least one odd})$

See the posted worksheet solutions for a solution under the conditions where we assume the elements are independent and the probability of an element being even is $\frac{1}{2}$.

CSC263 Lecture Extras: Average-Case Runtime Analysis of `hasTwentyOne`

```
    def hasTwentyOne(L):
1       j=L.head
2         while j!= None and j.key !=21:
3              j=j.next
4       return
```

We will do this analysis under the assumption that each of the n items in the input is equally likely to have the value 21 with probability p. We will count comparisons (which both happen in line 2.) The algorithm ends either in line 2 when it encounters an element where the key is 21 or when it reaches the end of the list. The probability that it ends on the first occurrence of line 2 is the probability that the first item with key=21 is at position 1. This is p. In this case it has 2 comparisons.

The probability that the execution ends on the 2nd time we look at line 2 (the second item) is the probability that item 2 has key=21 times the probability that we made it to item 2. We only consider item 2, if item 1 did not have key=21. So, the probability that we exit after finding 21 on second item is $p(1-p)$. This takes 4 comparisons.

The probability that the execution ends on the third item is probability that we reached the third item $(1-p)^2$ times the probability that it has key=21 which is p. So, Pr (exit on item 3) $= p(1-p)^2$ and the number of comparisons is $2 * 3 = 6$.

This goes on for all n items.

$$Pr(\text{exit on item } k) = p(1-p)^{k-1} \text{and number of comparisons is } 2k. \tag{1}$$

If no item in the list has 21 as its key, we take $2n+1$ comparisons. That is 2 comparisons for each time j points to the n items and then 1 extra comparison when j is None. The probability of this happening is $(1-p)^n$. Now, let's calculate the expected runtime starting with the formula we saw earlier.

$$\begin{aligned}
E[t] &= \sum_{\text{all cases}} Pr(\text{each case}) * t(\text{each case}) && (2)\\
&= \sum_{k=1}^{n} Pr(\text{first 21 at position k}) * 2k + Pr(\text{no 21 in list}) * (2n+1) && (3)\\
&= \sum_{k=1}^{n} 2kp(1-p)^{k-1} + (1-p)^n(2n+1) && (4)\\
&= \frac{2p}{1-p} \sum_{k=1}^{n} k(1-p)^k + (2n+1)(1-p)^n && (5)
\end{aligned}$$

In the last step we pulled the $2p$ out in front of the sum and multiply by $1-p/1-p$. Notice that $\sum_{k=1}^{n} k(1-p)^k$ is of the form $\sum_{k=1}^{n} ka^k$ which has a closed form. You probably learned how to solve for this sum in your math class (but in this class it is fine to look up the result).

CSC263 Lecture Extras: Average-Case Runtime Analysis of `hasTwentyOne`

Aside You should keep handy the following sums:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \qquad \text{arithmetic series} \tag{6}$$

$$\sum_{k=0}^{n} a^k = \frac{a^{n+1}-1}{a-1} \qquad \text{geometric series} \tag{7}$$

$$\sum_{k=1}^{n} ka^k = \frac{a^{n+1}(n(a-1)-1)+a}{(1-a)^2} \qquad \text{arithmetico-geometric series} \tag{8}$$

No, you shouldn't be memorizing this last one - just know that it exists and that you can look it up Returning to our problem and substituting a for $1-p$ we get

$$E[t] = (\frac{2p}{1-p})\frac{(1-p)^{n+1}(n(1-p-1)-1)+1-p}{p^2} + (2n+1)(1-p)^n \tag{9}$$

$$= \frac{2((1-p)^{n+1}(-np-1)+(1-p))}{(1-p)p} + (2n+1)(1-p)^n \tag{10}$$

$$= \frac{2((1-p)^n(-np-1)+1)}{p} + (2n+1)(1-p)^n \tag{11}$$

$$= \frac{2(1-p)^n}{p}(-np-1) + \frac{2}{p} + (2n+1)(1-p)^n \tag{12}$$

$$= (1-p)^n[(\frac{2}{p}(-np-1)+(2n+1)] + \frac{2}{p} \tag{13}$$

$$= (1-p)^n[-2n - \frac{2}{p} + 2n + 1] + \frac{2}{p} \tag{14}$$

$$= (1-p)^n[1 - \frac{2}{p}] + \frac{2}{p} \tag{15}$$

Let's consider an actual value for p. Suppose that an item in the list has a probability of 0.25 of having the key = 21. $p = \frac{1}{4}$

$$E[t] = (\frac{3}{4})^n[1 - \frac{2}{\frac{1}{4}}] + \frac{2}{\frac{1}{4}} \tag{16}$$

$$= (\frac{3}{4})^n[1-8] + 8 \tag{17}$$

$$= 8 - 7(\frac{3}{4})^n \tag{18}$$

Looking at this value, we see that the larger n becomes, the smaller $\frac{3}{4}^n$ becomes. The average-case gets closer and closer to 8.

1. The first activity is to get to know a few of your classmates. Go around the group, taking turns introducing yourselves, saying what timezone you are in and explaining one type of food that you ate over the holiday break. Try to pick food that is different than what others in your group have said. If you ate something special with your family, your classmates might be interested to hear about this. As a group decide whose food was the most interesting and then have that person enter the food in the appropriate row for your group in http://tinyurl.com/breakouts-263.

2. Prove that $7n - 2$ is in $\mathcal{O}(n)$. When you are finished discussing this question, even if you aren't sure about your solution, enter that you are done in your row in http://tinyurl.com/breakouts-263

WTP: $\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow 7n-2 \leq c \cdot n$

Let $n_0 = 1$. Let $c = 7$

$7n - 2 \leq 7n$ (since $n \geq 1$)

3. Prove that $3\log n + \log(\log(n))$ is in $\Omega(\log(n))$.

WTP: $\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow 3\log n + \log(\log n) \geq c \cdot \log(n)$

Let $n_0 = 2$. Let $c = 3$.

$3\log n + \log(\log n) \geq 3\log(n)$

4. Let X_n be a random variable sampled uniformly from the set $\{2^k \mid 0 \leq k \leq n\}$, where $n \in N$.
Define the function: $f(n) := E[X_n]$ where $E[\cdot]$ denotes the expectation operator.
Answer the following questions about the function $f(n)$.

   (a) What does $f(0)$ evaluate to?
   (b) What does $f(3)$ evaluate to?
   (c) What does $f(9)$ evaluate to?
   (d) What is the asymptotic growth rate of f with respect to n?
      i. $\Theta(n)$
      ii. $\Theta(2^n)$
      iii. $\Theta\left(\frac{2^n}{n}\right)$
      iv. None of the above

$f(0) = E[X_0] = 1$

$f(3) = \frac{2^0 + 2^1 + 2^2 + 2^3}{4} = \frac{1+2+4+8}{4}$

$= \frac{15}{4} = 3.75$

$f(9) = \frac{2^0 + 2^1 + \ldots + 2^9}{10}$

$= \frac{1023}{10} = 102.3$

5. Consider the following algorithm. We want to do a runtime complexity analysis of it counting the number of times we access a list element as the operation of interest. Assume that the subroutine used to implement line 2 is not smart. Even if it encounters a non-even number early on, it examines all n numbers.

```
1. def evens_are_bad(lst):
2.     if every number in lst is even:
3.         repeat lst.length times:
4.             calculate and print the sum of lst
5.         return 1
6.     else:
7.         return 0
```

(a) Do a best case analysis. When you are finished and moving on to part b, have one group member indicate that you are finished part a by entering this in the breakout document at http://tinyurl.com/breakouts-263

Best case upper bound: the best case is when at least 1 elem. in lst is odd. After n iterations through the loop on line 2, the if condition fails and it runs 1 iteration on line 7 to return 0. Thus $O(n)$.

Best case lower bound: since every input would require n iterations on line 2, it is $\Omega(n)$.

Thus, $\Theta(n)$.

(b) Do a worst case analysis.

When every element in lst is even, after n iterations on line 2, we enter line 3 where we traverse the length of the lst again. Line 4 takes constant time. Thus it is $\Omega(n^2)$. For any input of size n, line 2 will be executed which is n traversals. Then either the else statement is executed which takes constant time, or if statement in line 3 is executed which takes n more traversals. Line 4 is constant time.

Thus $O(n^2)$. $\Rightarrow \Theta(n^2)$

(c) For the average case analysis we need to make some assumptions about the numbers that we expect to see in the input. As a group decide on a reasonable assumption in the absence of any other information. Be clear about your sample space of inputs, the probability distribution over the sample space and the time for each case. Once you have decided on your sample space and your probability distribution, put them in the Google doc linked from the zoom chat in the row for your group. Please do NOT edit the rows from the other breakout groups. Then continue on with your analysis until Zoom brings you back to the main room.

Assume even or odd chance of element in lst is equally likely. We will count loop comparisons.

Thus $P(\text{all even in lst}) = (0.5)^n$

$t(\text{all even in lst}) = n^2 + n$

$$E[T] = (0.5)^n(n^2+n) + (n)(1-(0.5)^n)$$
$$= (0.5)^n n^2 + (0.5)^n \cdot n - n(0.5)^n + n$$
$$= (0.5)^n(n^2) + n$$

As $n \to \infty$, $(0.5)^n \to 0$. $E[T]$ is $\Theta(n)$

1. The first activity is to get to know a few of your classmates. Go around the group, taking turns introducing yourselves, saying what timezone you are in and explaining one type of food that you ate over the holiday break. Try to pick food that is different than what others in your group have said. If you ate something special with your family, your classmates might be interested to hear about this. As a group decide whose food was the most interesting and then have that person enter the food in the appropriate row for your group in http://tinyurl.com/breakouts-263.

2. Prove that $7n - 2$ is in $\mathcal{O}(n)$. When you are finished discussing this question, even if you aren't sure about your solution, enter that you are done in your row in http://tinyurl.com/breakouts-263

   SOLUTION: In order to prove that a function is in $\mathcal{O}(n)$ we will use the formal definition of big Oh.

   $f(n)$ is in $\mathcal{O}(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n > n_0$.

   So we need to find one value for $c > 0$ and $n_0 > 0$ where we can argue that this statement is true for $g(n) = n$.

   Let $f(n) = 7n - 2$, $c = 7$ and $n_0 = 1$.

   For all values of n greater than 1, $0 < 7n - 2 < 7n$. That is, $0 < f(n) < 7g(n)$ for all $n \geq 1$. Thus, the statement holds for $c = 7$ and $n_0 = 1$.

3. Prove that $3 \log n + \log(\log(n))$ is in $\Omega(\log(n))$.

   SOLUTION: Let $f(n) = 3 \log n + \log(\log(n))$.

   To prove that $f(n)$ is in $\Omega(g(n))$ we need to find values for $c > 0$ and $n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$.

   So we need to find one value for $c > 0$ and $n_0 > 0$ where we can argue that this statement is true for $g(n) = \log n$.

   $f(n) = 3 \log n + \log(\log(n)) > 3 \log n$ for $n \geq 2$. Thus, the statement holds for $c = 3$ and $n_0 = 2$.

4. Let X_n be a random variable sampled uniformly from the set $\left\{2^k \,|\, 0 \leq k \leq n\right\}$, where $n \in N$.

   Define the function: $f(n) := E[X_n]$ where $E[\cdot]$ denotes the expectation operator.

   Answer the following questions about the function $f(n)$.

   (a) What does $f(0)$ evaluate to? SOLUTION: 1

   (b) What does $f(3)$ evaluate to? SOLUTION: 3.75

   (c) What does $f(9)$ evaluate to? SOLUTION: 102.3

   (d) What is the asymptotic growth rate of f with respect to n?

      i. $\Theta(n)$

      ii. $\Theta(2^n)$

      iii. $\boxed{\Theta\left(\frac{2^n}{n}\right)}$

      iv. None of the above

5. Consider the following algorithm. We want to do a runtime complexity analysis of it counting the number of times we access a list element as the operation of interest. Assume that the subroutine used to implement line 2 is not smart. Even if it encounters a non-even number early on, it examines all n numbers.

```
1. def evens_are_bad(lst):
2.     if every number in lst is even:
3.         repeat lst.length times:
4.                 calculate and print the sum of lst
5.         return 1
6.     else:
7.         return 0
```

   (a) Do a best case analysis. When you are finished and moving on to part b, have one group member indicate that you are finished part a by entering this in the breakout document at http://tinyurl.com/breakouts-263

   SOLUTIONS: The best case occurs whenever there is at least one odd value in the list. In this case the code takes n array accesses all in line 2 and then goes to line 7 and exits. Since every input would require executing line 2, we can't do better than this $\Theta(n)$ performance.

   (b) Do a worst case analysis.

   SOLUTIONS: The worst case occurs when we have a list of n elements that are all even. In this case we have n accesses in line 2 and then n in each execution of line 4 which is called n times. So the number of operations is $n^2 + n$ which is in $\Omega(n^2)$. There is no other path through the code other than this case and the else case (the best one), so there is no way to require more array accesses than this. This means that the worst case is also in $\mathcal{O}(n^2)$. Therefore we have a tight bound of $\Theta(n^2)$

   (c) For the average case analysis we need to make some assumptions about the numbers that we expect to see in the input. As a group decide on a reasonable assumption in the absence of any other information. Be clear about your sample space of inputs, the probability distribution over the sample space and the time for each case. Once you have decided on your sample space and your probability distribution, put them in the Google doc linked from the zoom chat in the row for your group. Please do NOT edit the rows from the other breakout groups. Then continue on with your analysis until Zoom brings you back to the main room.

   SOLUTIONS: There are a number of sample spaces one could pick but in the absence of other information a reasonable choice might be lists of length n of all possible integers. There are an infinite number of elements and so we need to break this into a finite set of input classes for which inputs in each class have a common running time. We will use inputs that have at least one odd number and inputs that have all even numbers. Now we can define random variable T over the probability space to represent the running time of the algorithm.

$$T = \begin{cases} n^2 + n & \text{input contains only even numbers} \\ n & \text{otherwise} \end{cases}$$

   Now we use our formula to calculate the expected value of T as

$$\begin{aligned} E[T] &= \sum_t tPr[T = t] \\ &= (n) \times Pr[\text{list contains at least one odd}] + (n^2 + n) \times Pr[\text{list elements all even}] \end{aligned}$$

   Now we have to decide on the probability distribution over our inputs. That depends on what lists we expect our algorithm to encounter. In the absence of other information it might be a reasonable assumption to say that any particular list element has a probability $\frac{1}{2}$ of being even. Let's use that assumption. So the probability that a list of n elements are all even is $(\frac{1}{2})^n$. This means that the probability of the list having at least one odd element is $1 - (\frac{1}{2})^n$. Let's substitute that into our equation.

$$\begin{aligned} E[T] &= n(1 - (\tfrac{1}{2})^n) + (n^2 + n)((\tfrac{1}{2})^n) \\ &= n - n(\tfrac{1}{2})^n + n^2(\tfrac{1}{2})^n + n(\tfrac{1}{2})^n \\ &= n + n^2(\tfrac{1}{2})^n \end{aligned}$$

Notice that as n grows the n in the exponent causes $(\frac{1}{2})^n$ to shrink faster than n^2 grows and so the second term goes to 0 as n goes to infinity. So $E[T]$ is $\Theta(n)$

KEY POINTS FROM WEEK 1 PREP

1. WorstCase on $O(g(n))$: argue that algorithm runs max. $c \cdot g(n)$ steps on any input of size n

   Worst Case on $\Omega(g(n))$: find 1 input on which algorithm executes at least $c \cdot g(n)$ steps

   WC on $\Theta(g(n))$: argue both

2. $E[T] = \sum_t t\, P(T=t)$

3. $$P(X_n = i) = \begin{cases} 1 - \sum_{j=1}^{n} P(X_n = j) & \text{if } i = 0 \\ 2^{-i} & \text{if } 0 < i \le n \end{cases}$$

   $f(0) = 0 \cdot P(X_0 = i) = 0$

   $f(4) = \sum_{i=1}^{4} i \cdot P(X_4 = i) = 1(2^{-1}) + 2(2^{-2}) + 3(2^{-3}) + 4(2^{-4})$

   $= \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} = 1.625$

   $\lim_{n\to\infty} P(X_n = 0) = 1 - \sum_{j=1}^{n} P(X_n = j) = 1 - \sum_{j=1}^{\infty} 2^{-j} = 1 - 1 = 0$

   $\lim_{n\to\infty} n \cdot P(X_n = n) = \lim_{n\to\infty} n \cdot 2^{-n}$

4. $\sum_{i=0}^{127} \left(\frac{1}{128}\right) i = \frac{1}{128}\left[\frac{127(128)}{2}\right] = \frac{127}{2} = 63.5$

5. $\sum_{i=0}^{99} \left(\frac{1}{100}\right) i = \frac{1}{100}\left(\frac{99(100)}{2}\right) = \frac{99}{2} = 49.5$

   No. of times AnalyseWord called : $E[\text{len}(T)] * E[\text{words}(R)]$

   $= 49.5 \times 63.5 = 3143.25$

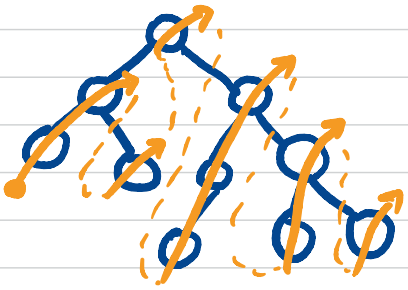
6. Priority Queue ADT

- Data: collection of items where each item has a priority.
- Operation:
  - Insert(PQ, x, priority): Add x, with the given priority, to PQ
  - FindMax(PQ): return an item in PQ with highest priority
  - ExtractMax(PQ): Remove and return an item from PQ with highest priority

7. predecessor of node = if BST, max value of its left subtree

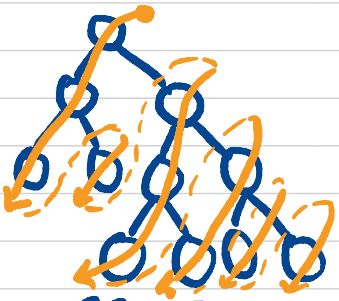
successor of node = if BST, min value of its right subtree

8 Tree traversal



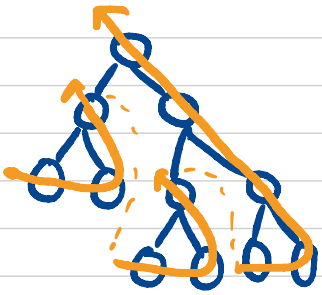
INORDER

left → root → right



PREORDER

root → left → right



POSTORDER

left → righ → root

WEEK 2 : Heaps

Announcements

- Recognized Study Groups (RSG) → pinned post on Quercus
- Piazza signup
- Academic Integrity Reminder
- New to course this week? Check out Quercus
  - Syllabus
  - Lecture notes and worksheets
  - Videos from last week

cs.utoronto.ca

csc263-2022-01@cs.toronto.edu.

Designing a Data Structure for Priority Queue ADT

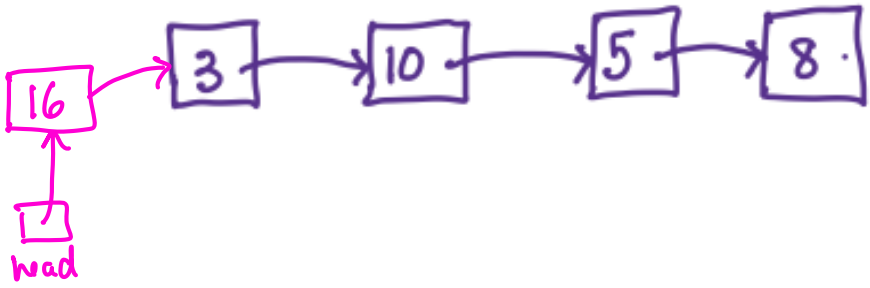
Data	**Operations**
A collection of items which each have a priority	Insert(PQ, x, priority) FindMax(PQ) ExtractMax(PQ)

Example sequence used across various implementations:
IN(8), IN(5), IN(10), IN(3), FM(), IN(16), EM(), EM(), IN(7)

Approach 0: An unsorted linked list

Discussed in DISCOVER module

Insert in O(1)
FindMax in O(n)
ExtractMax in O(n)



Important Side Note

- Both Python and Java often hide the complexity of operations
- Appending an element into a Python list

  L.append(x) ← much more than an array access.

In this course we want to write and analyze algorithms from the simple operations that don't depend on hidden complexity.

Chalk Talk

Approach 1: An unsorted Array

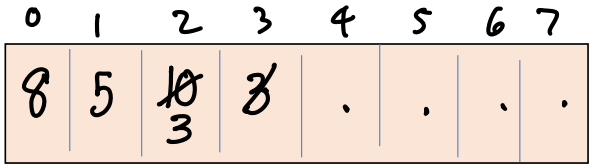
Insert $\Theta(1)$

FM $\Theta(n)$

EM $\Theta(n)$ to find

$\Theta(1)$ to remove

Size



IN(8), IN(5), IN(10), IN (3), FM(), IN(16), EM(), EM(), IN(7)

Chak Talk

Approach 2: A Sorted Array

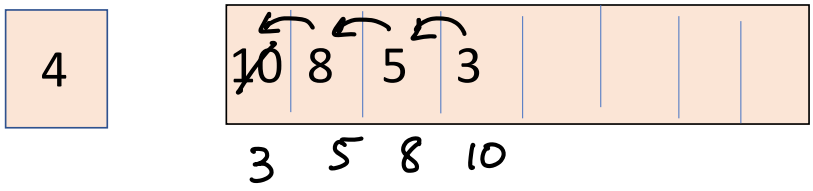
What if we kept the items in the array in sorted order?

FM $\Theta(1)$

EM ~~$\Theta(n)$~~ $\Theta(1)$

IN $\Theta(n)$

Size



IN(8), IN(5), IN(10), IN(3), || FM(), IN(16), EM(), EM(), IN(7)

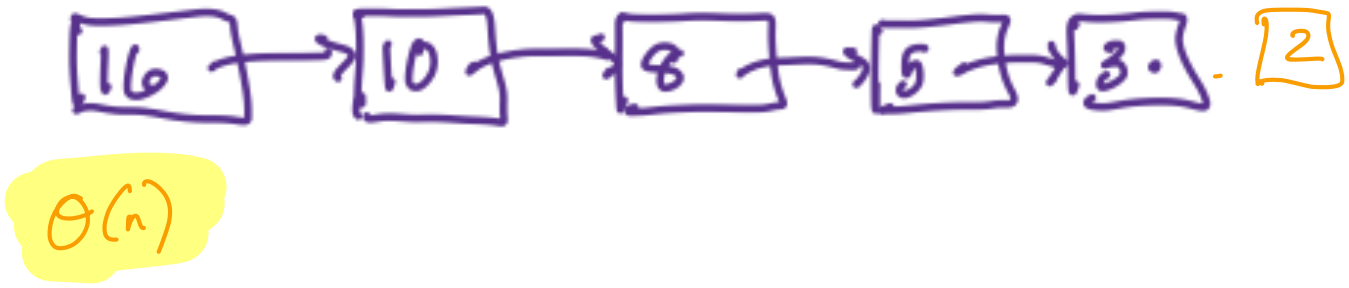
when array is sorted from highest to lowest : FM = $\Theta(1)$, EM = $\Theta(n)$, IN = $\Theta(n)$

when array is sorted from lowest to highest : FM = $\Theta(1)$, EM = $\Theta(1)$, IN = $\Theta(n)$

Worksheet Q1-4

Approach 3: An Ordered Linked List

What if we kept the items in an ordered linked list?

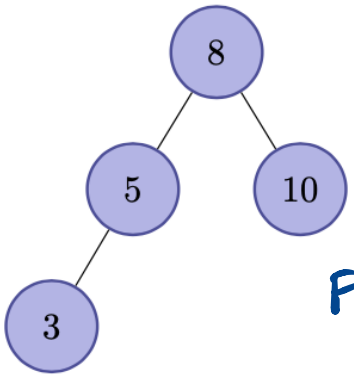


IN(8), IN(5), IN(10), IN(3), FM(), IN(16), EM(), EM(), IN(7)

Worksheet Q1-4

Approach 4: A Binary Search Tree

What if we kept the items in a binary search tree?



Θ(height of the tree)

worst case → height of the tree is n

FM: so Θ(n)

IN(8), IN(5), IN(10), IN(3), FM(), IN(16), EM(), EM(), IN(7)

Heaps

- Based on a nearly complete binary tree
- Heap Property determines relationship between values of parents and children
- Kind of sorted: enough to make query operations fast while not requiring full sorting after each update.
- Stored in an array

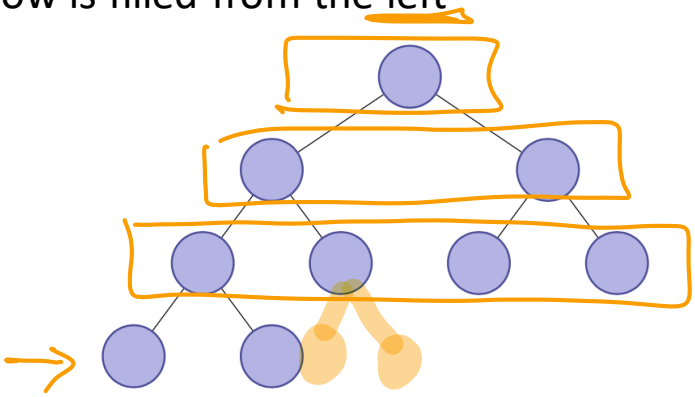
complete binary tree → every row of BST is filled completely

Nearly Complete Binary Tree

CLRS

↳ every row is filled completely except last row (filled from the left)

- Binary tree
- Every row is completely filled except possibly the lowest row
- The lowest row is filled from the left



Heap Property

- Max Heap

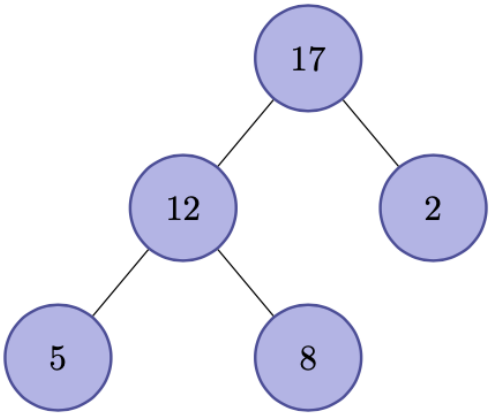
  The value at every node is equal to or greater than the value of its immediate children → the maximum element is at the root

- Min Heap (you fill this one in)

  the value at every node is equal to or less than the value of its immediate children

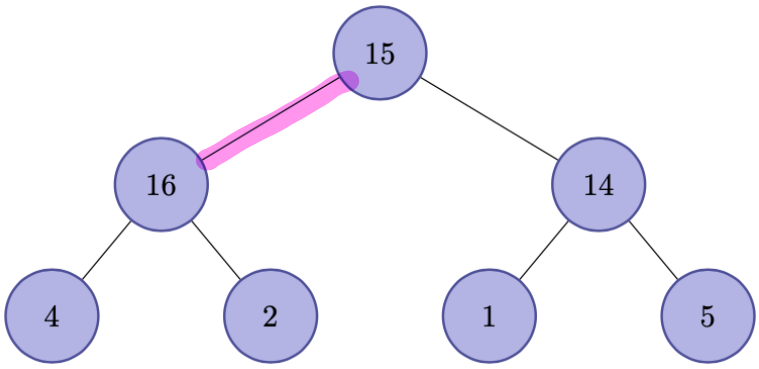
Zoom reaction activity

Is this a valid heap?



Zoom reaction activity

Is this a valid heap?



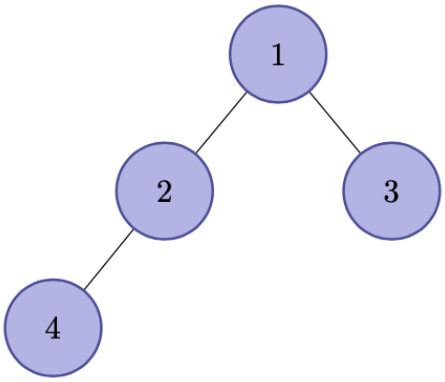
Not maxheap or minheap [BST]

Zoom reaction activity

Is this a valid heap?

Zoom reaction activity

Is this a valid heap?

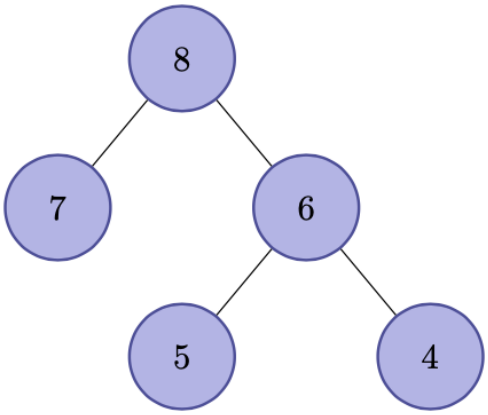


✔

a min heap

Zoom reaction activity

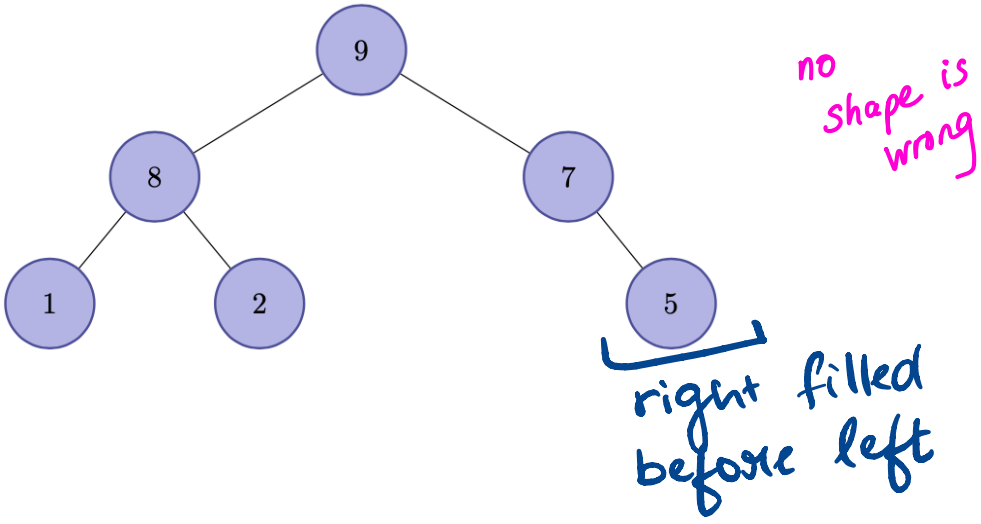
Is this a valid heap?



no
shape wrong
(not filled
from left)

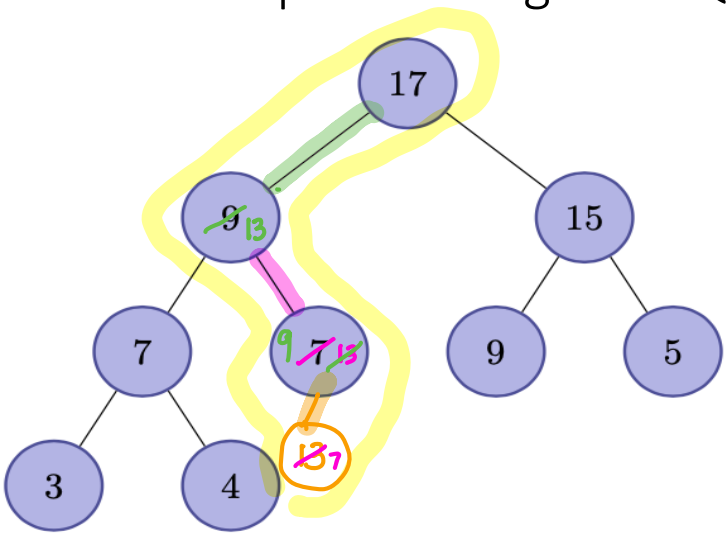
Zoom reaction activity

Is this a valid heap?



Chalk Talk

Implementing the PQ Operations: Insert

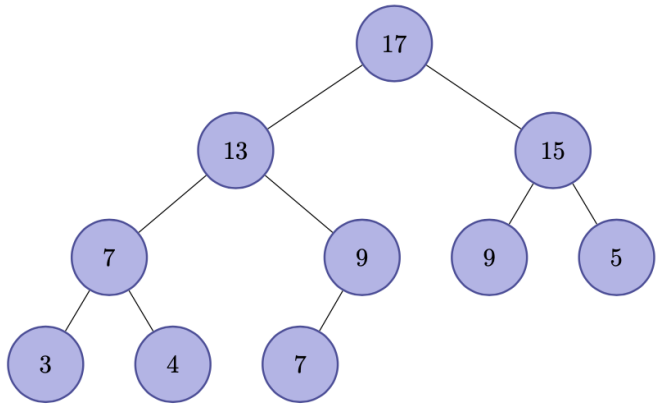


Insert(13)

- Increment heapsize and add element at next position
- Result might violate heap property so bubble up
- Running time? Θ(height of tree) = $\Theta(\log n)$

Chalk Talk

Implementing the PQ Operations: FindMax

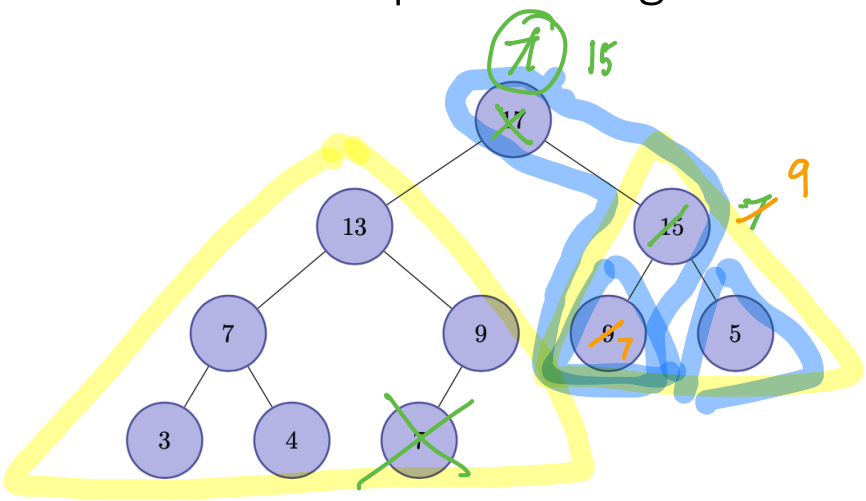


Findmax()

- Doesn't change heap
- Running time? $\Theta(1)$

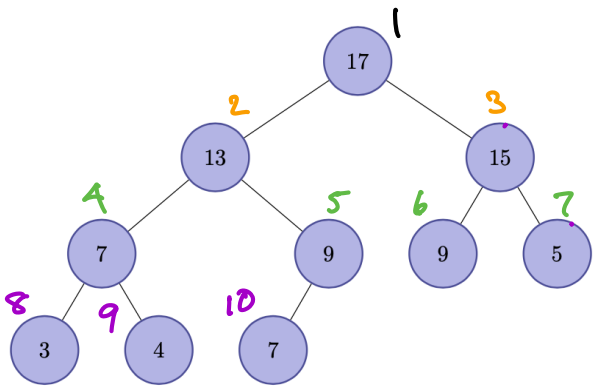
Chalk Talk

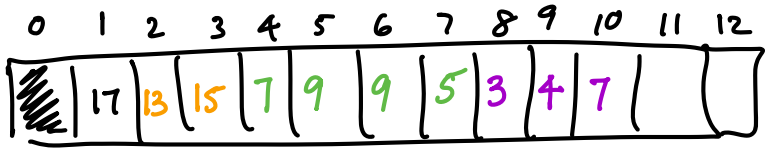
Implementing the PQ Operations: ExtractMax



- Remove and return root element
- Strategy: restore shape first then fix heap property
- Bubble down also called max-heapify
- Running time? Θ(height of tree) $\Theta(\log(n))$

Now the really cool bit!!!



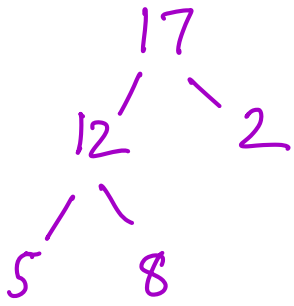


- Use an array to store the heap (not linked nodes and references)
- Convention: use 1-based indexing so root is at element 1
- For node at position i, left child is at 2i, right child is at 2i+1, and parent is at $\lfloor i/2 \rfloor$. i//2

Zoom reaction activity

Is this a valid heap?

A = [17, 12, 2, 5, 8]



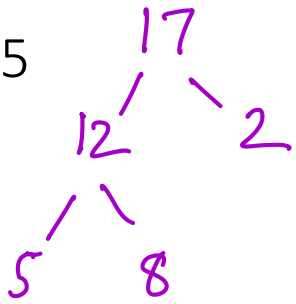
Yes

Zoom reaction activity

Is this a valid heap?

A = [17, 12, 2, 5, 8, 0, 0, 0] heapsize=5

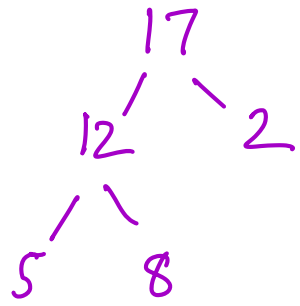
Same heap as before

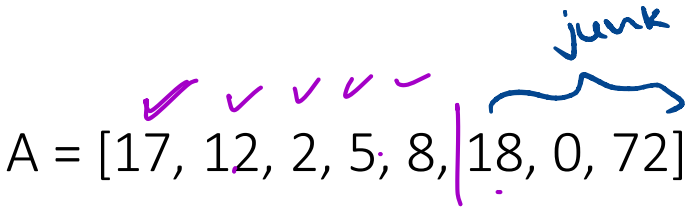


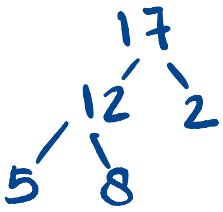
YES

Zoom reaction activity

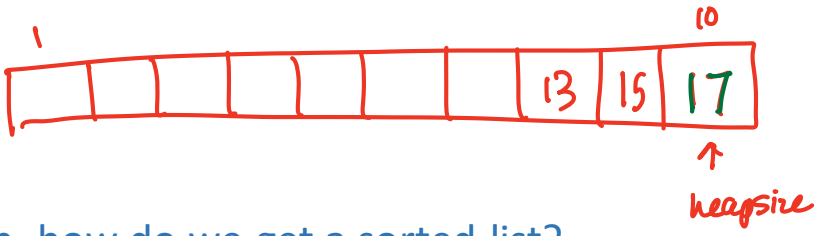
Is this a valid heap?





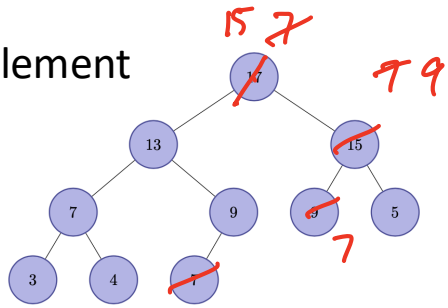


Heap Sort



Assuming we start with a valid heap, how do we get a sorted list?

Notice that the root of the heap stores the maximum element

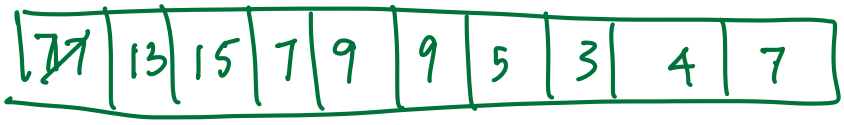


KEY IDEA

- Remove the root and put it in an array at the end
- Decrement heapsize
- Restore the heap property

2nd KEY IDEA

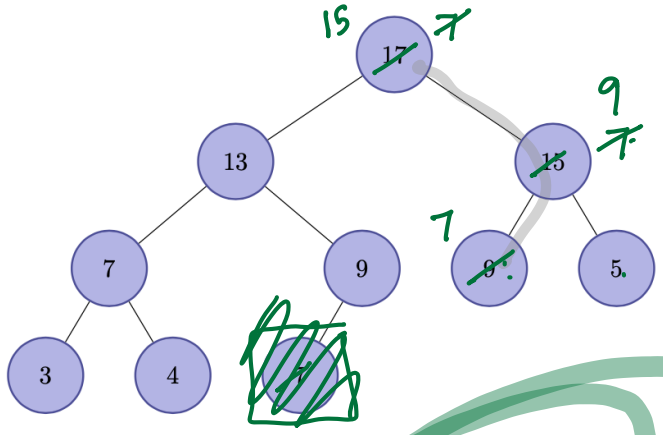
- Do this in place since replacement item for root was in the position where we want to put the root anyway.

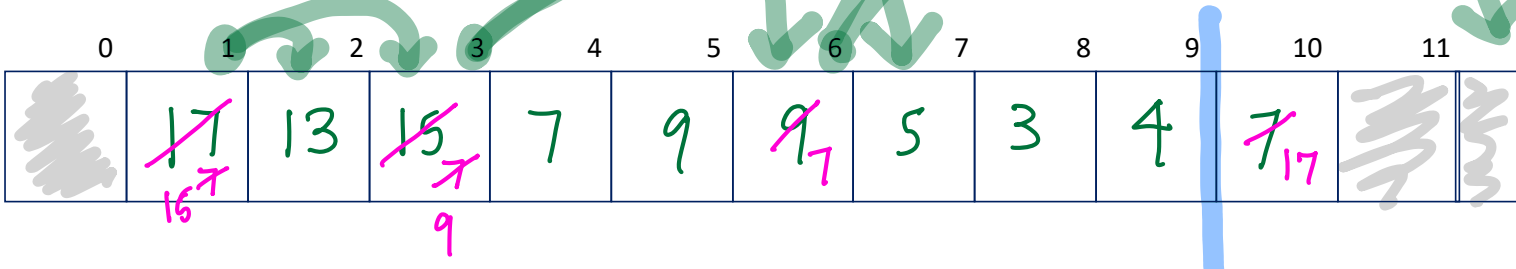


Chalk Talk

Heap Sort

Going from a Max Heap to a listed sorted in non-decreasing order:





children are at indices !! larger than heapsize..

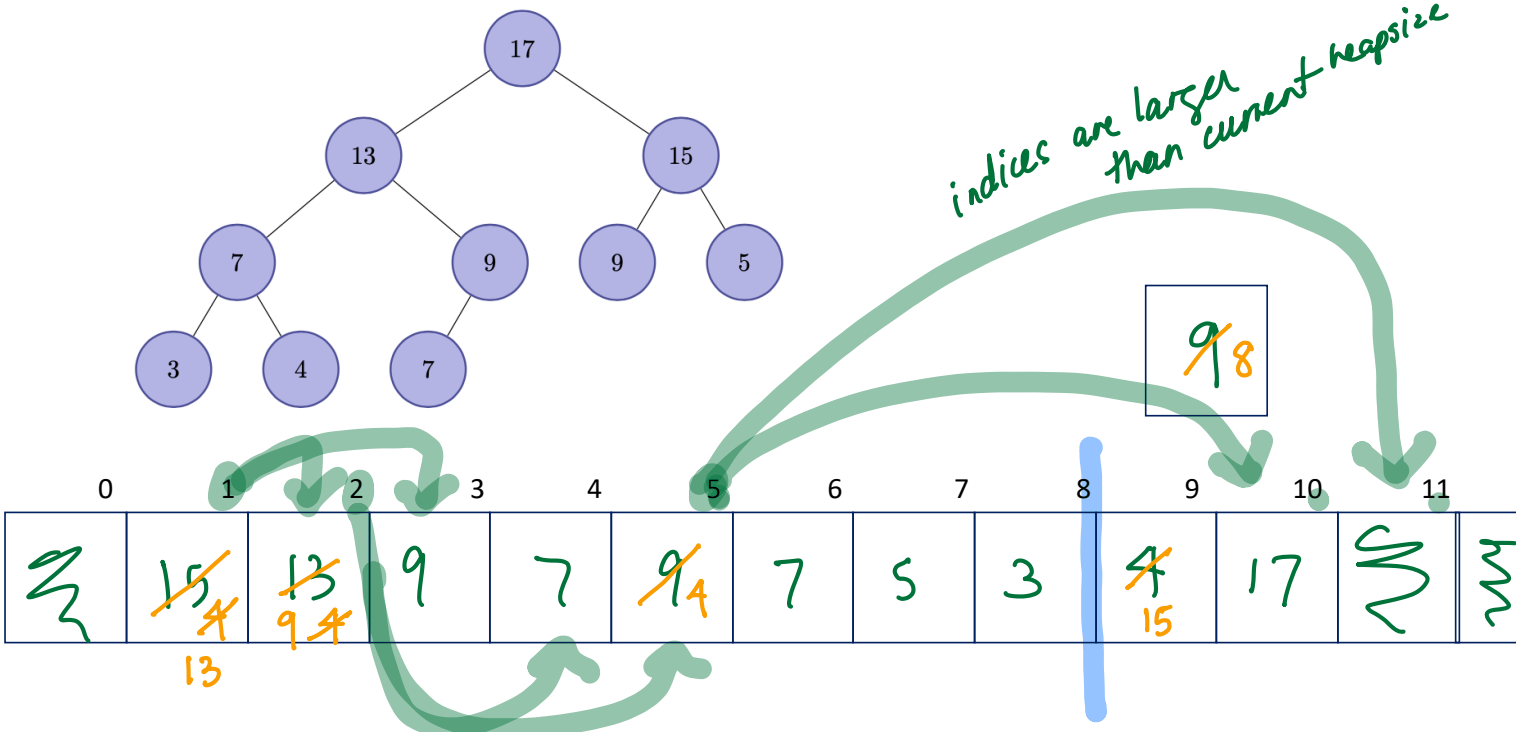
heapsize

when we switch 1 and 17, 17 becomes part of the garbage. But it is part of the sorted heap

Chalk Talk

Heap Sort

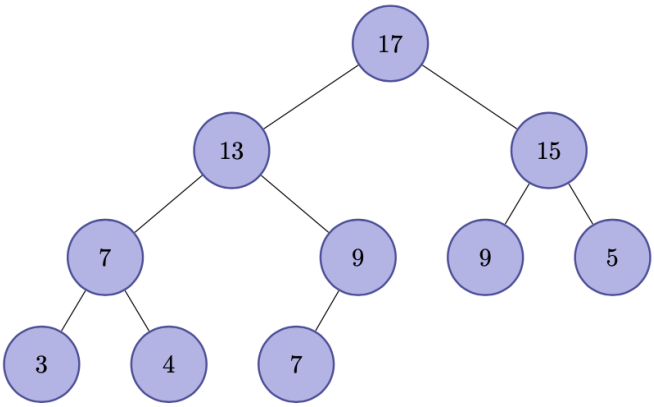
Going from a Max Heap to a listed sorted in non-decreasing order:

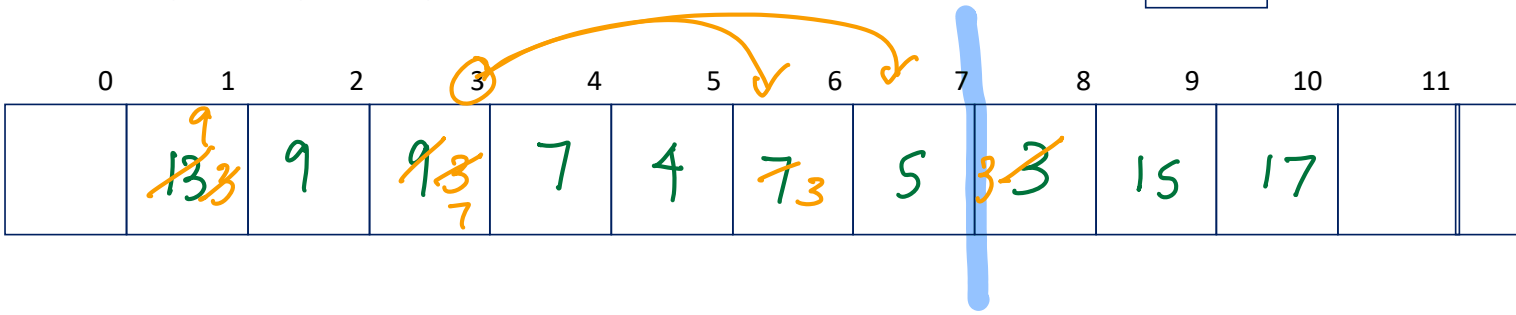


Chalk Talk

Heap Sort

Going from a Max Heap to a listed sorted in non-decreasing order:

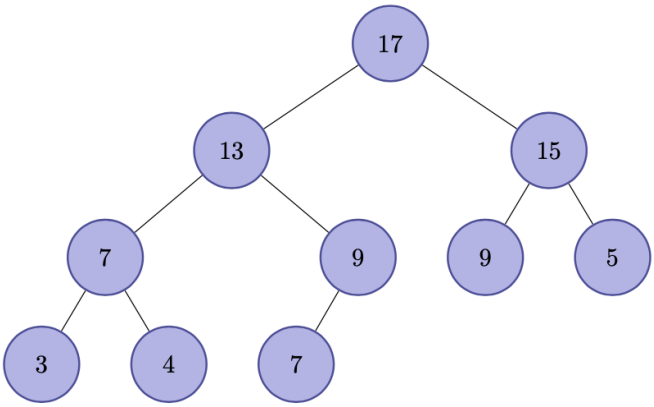




Chalk Talk

Heap Sort

Going from a Max Heap to a listed sorted in non-decreasing order:



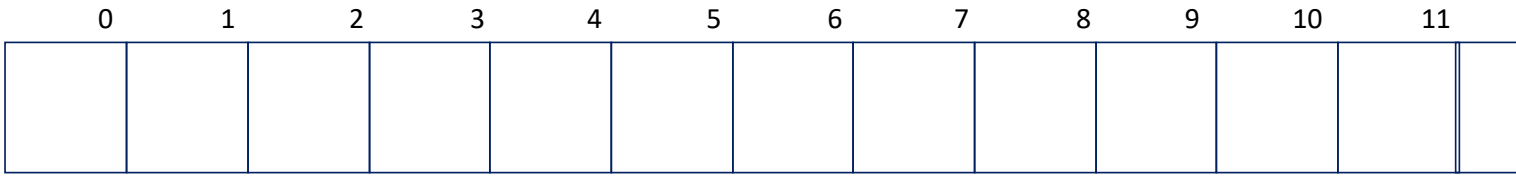
Complexity of going from heap to sorted list

Extract Max → log(n)

where n heapsize

how many times? n

$O(n \log(n))$



left ⇒ return 2i
right ⇒ return 2i+1

Bubble-down or Max-Heapify

List or Array

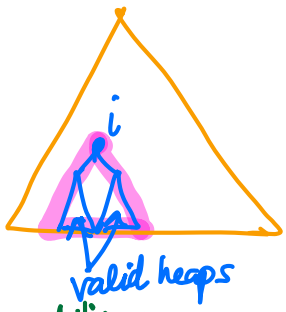
index into array

indices of subtrees of i

sets largest to index

```
def maxHeapify(L, i):
1       l = left(i)
2       r = right(i)
3       if l <= L.heapsize and L[l] > L[i]:
4               largest = l
5       else:
6               largest = i
7       if r <= L.heapsize and L[r] > L[largest]:
8               largest = r
9       if largest != i:
10              exchange L[i] with L[largest]
11              maxHeapify(L, largest)
```

precondition: L is an array representing a nearly binary tree from 1 to heapsize
subtrees @ left(i) and right(i) are valid heaps
but subtree at i isn't necessarily a valid heap.

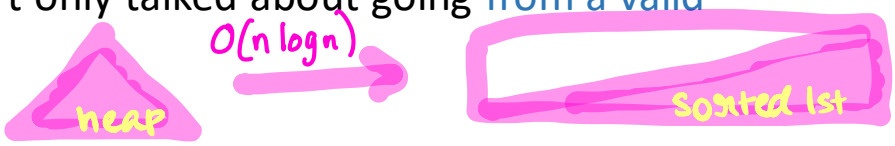


post condition:
subtree at i is a valid heap.

if left and right subtree were the same, this function picks left.

Building a Heap in the First Place

- Our discussion of heap sort only talked about going from a valid heap to a sorted list.



- But typically, we want sorting algorithms to go from an unsorted list to a sorted list.

- So, how do we efficiently go from an unsorted list to a valid heap?

Worksheet

Building a Heap in the First Place

- Complete Q5 and Q6 in this week's worksheet
- Put your results into the Google Doc for your breakout group
  https://tinyurl.com/breakouts-263
- If you are working alone (watching the videos later), stop the video and give yourself 10 minutes to do the exercise and write down your answers.
- If and only if you have finished Q5 and Q6, go on to Q7

More Space for Chalk Talk

Runtime of Building a Heap

- First Approach - start with empty heap, repeatedly call insert

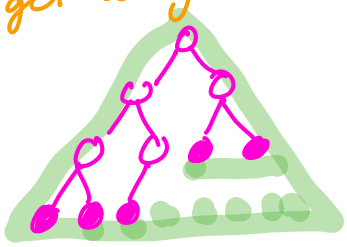
n total number of items

last insert takes $\log(n)$ time

all n inserts $\leq \log(n)$ time

build time $O(n \log n)$

can we get a tight bound?



family → add elements in increasing order.

at least half of the elements are leaves $\lceil \frac{n}{2} \rceil$

a leaf takes at least $\log(n) - 1$ time

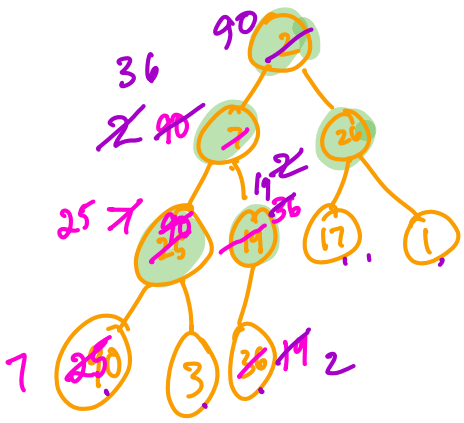
$\Omega \quad \frac{n}{2}(\log n - 1) \rightarrow \Omega(n \log n) \quad \Rightarrow \Theta(n \log n)$

Chalk Talk

Runtime of Building a Heap

start w/ unsorted array

- Using the second approach we make O(n) calls to Max-Heapify and each one takes O(log n), so we immediately get a bound of O(n log n) but we can do better!



90 36 26 25 19 17 1

7 3 2

start i at $\lfloor \frac{n}{2} \rfloor$ and work back to i=1

call Max Heapify (L, i)

$O(n \log n)$

More Space for Chalk Talk

Runtime of Building a Heap

Running time of MaxHeapify(A,i) is proportional to height of tree rooted at i

- So how many subtrees of each height do we have?

$\frac{n}{2}$ leaves at height 0 → require 0 swaps

$n/4$ nodes at height 1 → require 1 swap

$n/8$ nodes at height 2 → " 2 swaps

⋮

2 nodes at height $\log n - 1$ → " $\log n - 1$ swaps

1 node at height $\log n$ → " $\log n$ swaps

A heap contains at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height h

More Space for Chalk Talk

Runtime of Building a Heap

total time $= \sum_{h=1}^{\lfloor \log n \rfloor} h * \#\text{nodes at height } h$

$= \sum_{h=1}^{\lfloor \log n \rfloor} h \left\lceil \frac{n}{2^{h+1}} \right\rceil$

$= O\left(\frac{n}{2}\right) \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}$

$< \sum_{h=1}^{\infty} \frac{h}{2^h}$

$= \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}$

$= 2$

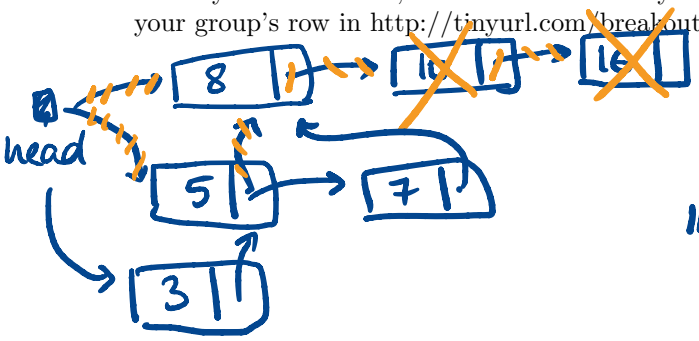
$= O(2n) = O(n)$

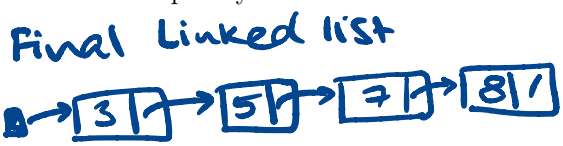
Gabriel's Staircase Series:

use $r = 1/2$

$$\sum_{k=1}^{\infty} kr^k = \frac{r}{(1-r)^2}, \quad \text{for } 0 < r < 1$$

1. As a group, discuss how we would use an ordered linked list to implement a priority queue. Determine the worst-case complexity of each operation. Here is the list of operations that we have been using so far to demonstrate the different implementations: IN(8), IN(5), IN(10), IN(3), FM(), IN(16), EM(), EM(), IN(7). When you are finished, have one member of your group put the worst-case time complexity for insert into your group's row in http://tinyurl.com/breakouts-263





INSERT → $\Theta(n)$
FM → $\Theta(1)$
EM → $\Theta(1)$

2. Now consider how we would use a binary search tree to implement a priority queue. Determine the worst-case complexity of each operation. Again, put the worst-case complexity for insert into the breakouts Google doc.

Use max heap
Insert → $\Theta(\log n)$
FM → $\Theta(1)$
EM → $\Theta(\log n)$

3. What makes the sorted linked list implementation appealing over the BST implementation? (Which operations are faster and why?)

ExtractMax faster in sorted linked list

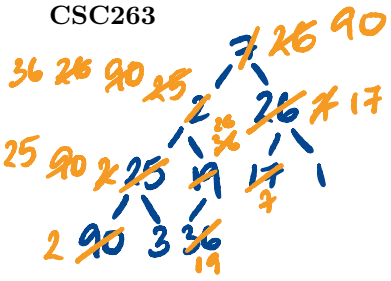
4. What makes the BST implementation seem possibly appealing over the sorted linked-list implementation? (Which operation might be faster?)

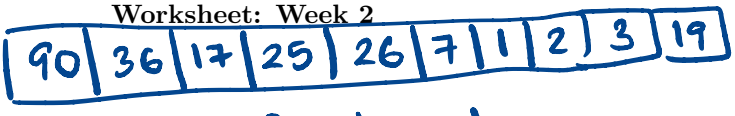
Insert & FindMax is faster

5. Supposed that we have an array A of n unsorted values. One approach to building a heap from these values is to start with an empty (i.e. garbage-filled) array B of length n and for each value in A, call Insert to insert it into the heap.

   (a) Using the values for A [2,7,26,25,19,17,1,90,3,36], trace this algorithm's execution and determine the heap B that would result. Put the array representing the resulting heap in the Google Doc (http://tinyurl.com/breakouts-263) for this breakout session.

   (b) What is the total worst-case running time for this approach?





O(log n) for insert

worst case → elements sorted in increasing order. O(n log n)

6. Suppose that we have that same array A of n unsorted values. A different approach to building a heap from these values is to work directly within array A starting from back to front calling bubble-down (or MaxHeapify(i) on each node.

   (a) Using the values for A [2,7,26,25,19,17,1,90,3,36], trace this algorithm's execution and determine the array that would result. **Strong Suggestion:** Do not try to do this inside the array. Instead draw the tree representation and execute the bubble-down on the appropriate subtrees. Then, when you are finished, copy the values back into an array. Copy the array representing the resulting heap in the breakouts Google Doc.

   (b) Discuss these questions as a group:

   Not same, both valid

   - Are the heaps created by these two approaches the same? If not, is this ok?
   - Do we really need to start at n and run MaxHeapify n times or can we do better?
   - What is the total worst-case running time for this approach expressed in Big O?

7. Once you have entered your group's results into the Google doc, to go https://visualgo.net/en/heap. When the site first loads, you will be in e-Lecture mode. You need to press ESC to leave this mode and be able to work with the visualizations. On the bottom left is an expanding menu where the first option is **Create(A) O(N log N)**. Select this option and then pick **Go** to see a visualization of something that should look very familiar. You can confirm that you traced correctly earlier in this worksheet.

   Caution: CS education research suggests that students don't actually learn much at all from *merely watching* visualizations of data structures. The theory claims that you will learn more if you carefully make predictions about what will happen, write down your prediction, and then run the visualization to check (and possible correct) your understanding.

   After you have confirmed that your tracing of the first approach was correct, look at the second one by running the **Create(A) O(N)** option.

1. As a group, discuss how we would use an ordered linked list to implement a priority queue. Determine the worst-case complexity of each operation. Here is the list of operations that we have been using so far to demonstrate the different implementations: IN(8), IN(5), IN(10), IN(3), FM(), IN(16), EM(), EM(), IN(7). When you are finished, have one member of your group put the worst-case time complexity for insert into your group's row in http://tinyurl.com/breakouts-263

   SOLUTION: Keep the items in order with the highest priority at the head of the list. FindMax() is then $\Theta(1)$ since we just look at the end, and ExtractMax() is also constant time, since we only have to reset the head pointer. Insert requires traversing the list and in the worst case the inserted item may have the lowest priority and go at the tail of the list. So it is $\Theta(n)$.

2. Now consider how we would use a binary search tree to implement a priority queue. Determine the worst-case complexity of each operation. Again, put the worst-case complexity for insert into the breakouts Google doc.

   SOLUTION: The items could be ordered by the priority queue but that doesn't actually help us much. Both FindMax() and ExtractMax() will require finding the largest element in the tree which means starting at the root and repeatedly looking for the right child until we find a node that doesn't have a right child. In the worst case, the tree is actually a linked list in sorted order with increasing priorities, and this is worse than the sorted linked list in question 1.

   Insert() finds the appropriate place in the tree to add the new leaf. In the worst case it is Θ(height).

3. What makes the sorted linked list implementation appealing over the BST implementation? (Which operations are faster and why?)

   SOLUTION: FindMax and ExtractMax are both constant time operations if we keep the list sorted so the highest priority item is first. In the BST, we would have to start at the root and find the right-most child.

4. What makes the BST implementation seem possibly appealing over the sorted linked-list implementation? (Which operation might be faster?)

   SOLUTION: If we could keep the tree compact, then the insertion would be faster because it is $\mathcal{O}$(height).

5. Supposed that we have an array A of n unsorted values. One approach to building a heap from these values is to start with an empty (i.e. garbage-filled) array B of length n and for each value in A, call Insert to insert it into the heap.

   (a) Using the values for A [2,7,26,25,19,17,1,90,3,36], trace this algorithm's execution and determine the heap B that would result. Put the array representing the resulting heap in the Google Doc (http://tinyurl.com/breakouts-263) for this breakout session.

   (b) What is the total worst-case running time for this approach?

   SOLUTION: Resulting heap is [90, 36, 17, 25, 26, 7, 1, 2, 3, 19]. Running time is $\Theta(n \log n)$. $\mathcal{O}(\log n)$ for each of the n insertions, and in the worst-case, $\Omega(\log(n) - 1)$ for at least $n/2$ elements. The worst-case behaviour of the algorithm is exhibited when the input array A is sorted in strictly increasing order, for example. In this scenario, each the last $n/2$ elements of the array (i.e. the right half of A) must bubble-up to the root of the heap (whose height is at least $\log(n/2)$) when they are inserted.

6. Suppose that we have that same array A of n unsorted values. A different approach to building a heap from these values is to work directly within array A starting from back to front calling bubble-down (or MaxHeapify(i) on each node.

   (a) Using the values for A [2,7,26,25,19,17,1,90,3,36], trace this algorithm's execution and determine the array that would result. **Strong Suggestion:** Do not try to do this inside the array. Instead draw the tree representation and execute the bubble-down on the appropriate subtrees. Then, when you are finished, copy the values back into an array. Copy the array representing the resulting heap in the breakouts Google Doc.

   (b) Discuss these questions as a group:

   - Are the heaps created by these two approaches the same? If not, is this ok?
   - Do we really need to start at n and run MaxHeapify n times or can we do better?
   - What is the total worst-case running time for this approach expressed in Big O?

SOLUTION: Resulting heap is [90, 36, 26, 25, 19, 17, 1, 7, 3,2]. The heaps are different but both are valid max heaps with these values. We did not need to run MaxHeapify on the leaves since they are already heaps. Since at least half of the nodes in a nearly complete binary tree must be leaves, we can start at the node that is at $\lfloor \frac{n}{2} \rfloor$. Running time is $\mathcal{O}(n \log n)$. $\mathcal{O}(\log n)$ for each of the $\lfloor \frac{n}{2} \rfloor$ calls to MaxHeapify. But this is not a tight bound. In lecture we discussed how we can prove that the algorithm runs in $\mathcal{O}(n)$. See the posted lecture notes or the video for this solution.

7. Once you have entered your group's results into the Google doc, to go https://visualgo.net/en/heap. When the site first loads, you will be in e-Lecture mode. You need to press ESC to leave this mode and be able to work with the visualizations. On the bottom left is an expanding menu where the first option is **Create(A) O(N log N)**. Select this option and then pick **Go** to see a visualization of something that should look very familiar. You can confirm that you traced correctly earlier in this worksheet.

   Caution: CS education research suggests that students don't actually learn much at all from *merely watching* visualizations of data structures. The theory claims that you will learn more if you carefully make predictions about what will happen, write down your prediction, and then run the visualization to check (and possible correct) your understanding.

   After you have confirmed that your tracing of the first approach was correct, look at the second one by running the **Create(A) O(N)** option.

WEEK 2 PREP

1.

ADT	Insert	FindMax	ExtractMax
unsorted linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sorted array (ascending)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
BST	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
max-heap	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$

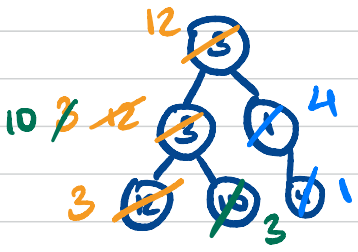
↳ the height of tree could be n worst case

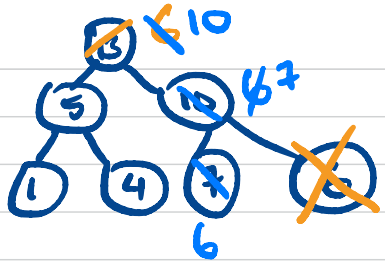
2. Heap ⇒ Nearly complete BST
   ⇒ max or min
   ⇒ stored in array & kinda sorted

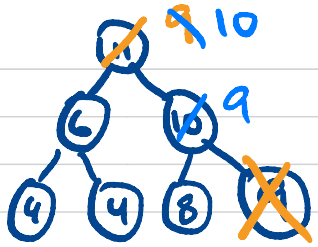
3. Heap → for node at position i
   - left child is $2i$
   - right child is $2i+1$
   - parent at $\lfloor \frac{i}{2} \rfloor$

4. Max-heap to sorted non-decreasing array
   - switch last element and root (first element in array)
   - bubble down
   - repeat

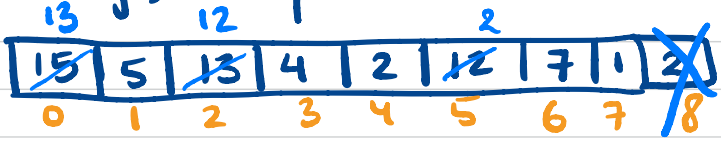
5. IN(3), IN(3), IN(1), IN(12), IN(10), IN(4)

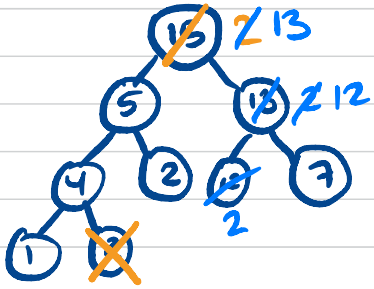


6. 



7. Array, Heapsize = 9





8. Dictionary ADT

Data: set of items where each item has a key which is UNIQUE

Operations:

- INSERT(S, x): Insert x in S. If some element y in S has y.key = x.key, replace y with x.
- Search(S, k): Return x in S with x.key = k or null if no x in S
- Delete(S, x): Delete x from S. (x is element not key)

9. Valid heap $\xrightarrow[O(n \log n)]{}$ sorted list

Unsorted list $\xrightarrow[O(n \log n)]{}$ valid heap

10. A heap contains at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height h

Runtime for building a heap = $O(n)$