Algorithm Analysis 3: Recurrences

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar	Kristin Li	Winston Jodjana
Maggie Jiang	Hans Zhang	Michael Duan
Jeffery Tian	Annie Mao	

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-06-29A

Announcements

- No quiz this week
  - Future quizzes will have a deadline of 3am on Saturday
  - Lecture Feedback: Pretty even split between Zoom, Google Docs, and PollEverywhere
- Ex 2,3 out today, due next Monday
  - Ex 3 is widely considered the hardest, so start early
- Project 1 Checkpoint 1 tomorrow!
- Guest lecturer on Wednesday

pollev.com/332summer :: tinyurl.com/332-06-29A

Lecture Outline

- Algorithm Analysis
  - **Review: Amortized bounds**
  - Where We've Come
  - Recurrences
- Priority Queue ADT

pollev.com/332summer :: tinyurl.com/332-06-29A

Complexity Cases

- We started with two cases:
  - **Worst-case complexity**: *maximum* number of steps algorithm takes on "most challenging" input of size N
  - **Best-case complexity**: *minimum* number of steps algorithm takes on "easiest" input of size N

- We are punting on one case: **Average-case complexity**
  - Sometimes: relies on distribution of inputs
    - Eg, binary heap's O(1) insert (we will get to this)
    - See CSE312 and STAT391
  - Sometimes: uses randomization in the algorithm
    - Will see an example with sorting; also see CSE312

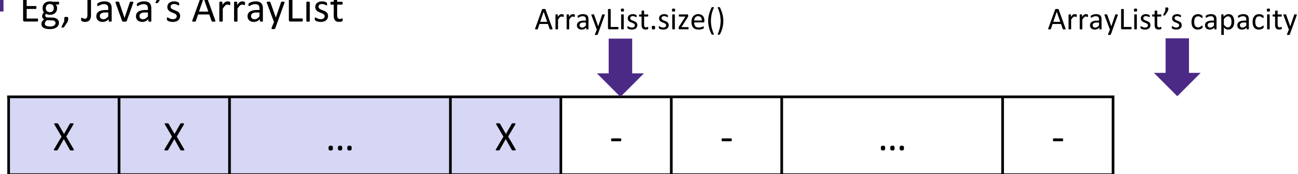
- We've mentioned, but not defined, one *category* of cases:
  - **Amortized-case complexity**

Amortized Analyses = Multiple Executions

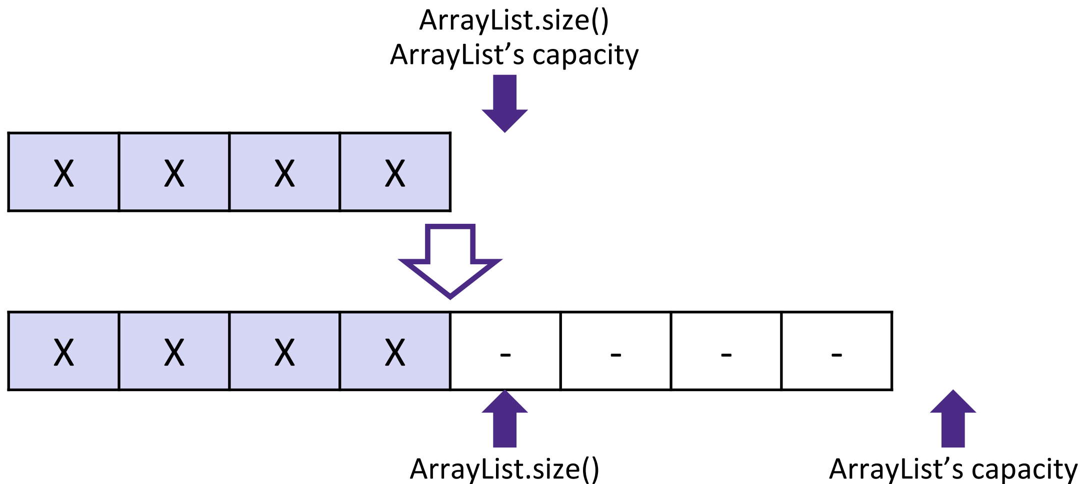
Single Execution	Multiple Executions
Worst Case	Amortized Worst Case
Best Case	Amortized Best Case
Average Case	*Amortized Average Case*

Amortized Analysis: `ArrayList.add()`

- Consider adding an element to an array-backed structure
  - Eg, Java's ArrayList



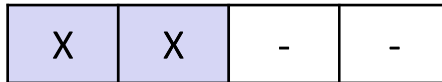
- When the underlying array fills, we allocate and copy contents

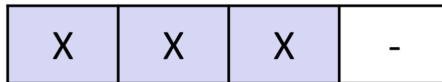


`ArrayList.add()`: Single Operation Runtime

- We know that copying a single element and allocating arrays are both constant-time operations
  - Let's call their runtimes 'c' and 'd', respectively

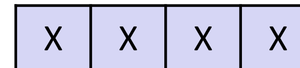
Most of the time

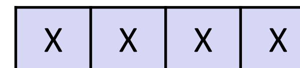
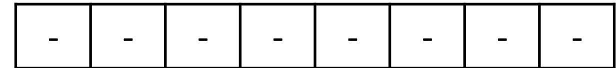
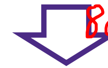
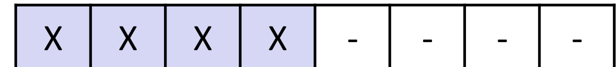
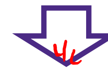
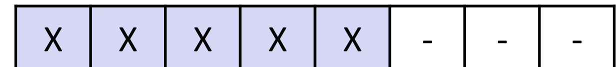




Runtime: $O(1)$

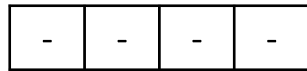
Worst case



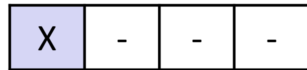







Runtime: $8d+5c = O(n)$

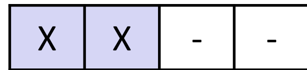
`ArrayList.add()`: Worst-Case Amortized Runtime



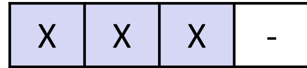
add(X)



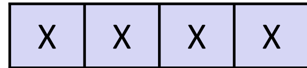
add(X)



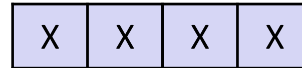
add(X)

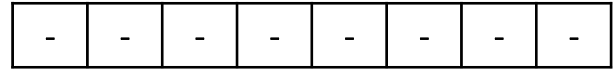


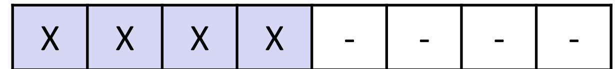
add(X)

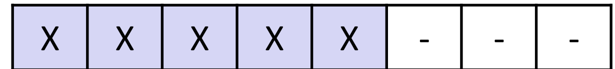


add(X)









Worst-case Aggregate Runtime:

$\frac{2nd + 2nc + c}{n+1} \in O(1)$

Amortized Analysis Intuition

- See Weiss, ch 11, for formal methods
- But the intuition is: if our client is willing to tolerate it, we will "smooth" the *aggregate cost of n operations* over n itself

Single Execution	Multiple Executions
Worst Case: O(n)	Amortized Worst Case: O(1)
Best Case:	Amortized Best Case:

- Note: we increased our array size by a factor of n (eg, 2n, 3n, etc). What if we increased it by a constant factor (eg, 1, 100, 1000) instead?

Lecture Outline

- Algorithm Analysis
  - Review: Amortized bounds
  - **Where We've Come**
  - Recurrences
- Priority Queue ADT

pollev.com/332summer :: tinyurl.com/332-06-29A

Closing Thoughts

- Asymptotic analysis gives us a common “frame of reference” with which to compare algorithms
  - Most common comparisons are Big-O, Big-Omega, and Big-Theta
  - But also little-o and little-omega

- Case Analysis != Asymptotic Analysis
  - We combine asymptotic analysis and case analysis to compare the behavior of data structures and algorithms

- When comparing two algorithms, you must pick all of these:
  - A case (eg, best, worst, amortized, etc)
  - A metric (eg, time, space)
  - A bound type (eg, big-O, big-Theta, little-omega, etc)

Closing Thoughts

- Big-Oh can also use more than one variable
  - Example: can sum all elements of an *n*-by-*m* matrix in $O(nm)$
  - We will use this when we get to graphs!

- Asymptotic complexity for ***small n*** can be misleading
  - Example: $n^{1/10}$ vs. `log` n
    - Asymptotically, $n^{1/10}$ grows more quickly
    - But the “cross-over” point (n_0) is around $5*10^{17} \approx 2^{58}$; you might prefer $n^{1/10}$
  - Example: QuickSort vs InsertionSort
    - *Expected runtimes:* Quicksort is O(n log n) vs InsertionSort $O(n^2)$
    - In reality, InsertionSort is faster for small n’s
    - (we’ll learn about these sorts later)

Closing Thoughts

- Asymptotic complexity for ***specific implementations*** can also be misleading …
  - *Evaluating an algorithm?* Use asymptotic analysis
  - *Evaluating an implementation?* Timing can be useful
    - Either a hardware or a software implementation

- At the core of CS is a backbone of theory & mathematics
  - We've spent 2 lectures on how to analyze an algorithm, mathematically, not the implementation
  - But timing has it's place in the real world
  - We do want to know whether implementation A runs faster than implementation B on data set C
  - Ex: Benchmarking graphics cards

Lecture Outline

- Algorithm Analysis
  - Review: Amortized bounds
  - Where We've Come
  - Recurrences
    - **Linear Search example**
    - Binary Search example
    - Binary & Linear Sum example
- Priority Queue ADT

Analyzing Code

- Basic ***operations*** take "some amount of" ***constant time***
  - Arithmetic
  - Assignment
  - Access one Java field **or array index**
  - Etc.
  - (Again, this is an *approximation of reality*)

Consecutive statements	*Sum of time of each statement*
Loops	*Num iterations * time for loop body*
Recurrence	Solve recurrence equation
Function Calls	*Time of function's body*
Conditionals	*Time of condition + time of {slower/faster} branch*

Analyzing Iterative Code: Linear Search

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
  for(int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
    return false;
}
```

Best case: 6 “ish” steps = $O(1)$

Worst case: 5 “ish” * (arr.length) + 1
= O(arr.length)

Runtime expression:
$5 * n + 1 = O(n)$

Analyzing Recursive Code

- Computing runtimes gets interesting with recursion

- *Example*: compute something recursively on a list of size n. Conceptually, in each recursive call we:
  - Perform some amount of work; call it **w(n)**
  - Call the function recursively with a smaller portion of the list

- If reduce the problem size by 1 during each recursive call, the runtime expression is:
  - *Recursive case*: T(n) = w(n) + T(n-1)
  - *Base case*: T(1) = 5 = O(1)

$$T(n) = \begin{cases} w(n) + T(n-1) \\ 5 \text{ if } n = 1 \end{cases}$$

- Recursive part of the expression is the “recurrence relation”

Example Recursive Code: Summing an Array

- We can ignore **sum**'s contribution to the runtime since it's called once and does a constant amount of work
- Each time **help** is called, it does that a constant amount of work, and then calls **help** again on a problem one less than previous problem size
- Runtime Relation:

  T(0) = σ(1) = c_1

  T(n) = w(n) + T(?)
  c_2 + T(n-1)

```
int sum(int[] arr) {
  return help(arr, 0);
}

int help(int[]arr,int i) {
  if(i == arr.length)
    return 0;
  return arr[i] + help(arr, i+1);
}
```

Solving Recurrence Relations: Expansion (1 of 2)

- Now we just need to solve our recurrence relation
  - ie, reduce it to a closed form
- Use Technique #1: Expansion
  - Also known as "unrolling"
- Basically, we write it out to find the general-form expansion

$$
\begin{aligned}
T(n) &= 5 + T(n-1) \\
&= 5 + 5 + T(n-2) \\
&= 5 + 5 + 5 + T(n-3) \\
&= \ldots \\
&= 5k + T(n-k)
\end{aligned}
$$

Solving Recurrence Relations: Expansion (2 of 2)

- We have a general-form expansion:

$$T(n) = 5k + T(n\text{-}k)$$

- And a base case:

$$T(0) = 3 \quad \leftarrow c_1$$

- When do we hit the base case?
  - When n-k = 0!

$n = k$

$$T(n) = 5n + T(n - n)$$
$$= 5n + 3$$
$$= c_2 n + c_1 \quad \in O(n)$$

Lecture Outline

- Algorithm Analysis
  - Review: Amortized bounds
  - Where We’ve Come
  - Recurrences
    - Linear Search example
    - **Binary Search example**
    - Binary & Linear Sum example

- Priority Queue ADT

Example Recursive Code: Binary Search

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k) {
  return help(arr,k,0,arr.length);
}
boolean help(int[] arr, int k, int lo, int hi) {
  int mid = (hi+lo)/2; // i.e., lo+(hi-lo)/2
  if(lo==hi)            return false;
  if(arr[mid] == k)     return true;
  if(arr[mid] < k)      return help(arr, k, mid+1, hi);
  else                  return help(arr, k, lo, mid);
}
```

Example Recursive Code: Binary Search

Base case:

$T(1) = c_1$

Recursive case:

$T(n) = T(\frac{n}{2}) + c_2$ (annotated: w(n))

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k) {
  return help(arr,k,0,arr.length);
}
boolean help(int[] arr, int k, int lo, int hi) {
  int mid = (hi+lo)/2; // i.e., lo+(hi-lo)/2
  if(lo==hi)          return false;
  if(arr[mid] == k)   return true;
  if(arr[mid] < k)    return help(arr, k, mid+1, hi);
  else                return help(arr, k, lo, mid);
}
```

Technique #1: Expansion

1. Determine the recurrence relation and base case

$T(n) = T(\frac{n}{2}) + c_2$ $\quad T(1) = c_1$

2. "Expand" the original relation to find the general-form expression *in terms of the number of expansions*

$$T(n) = T(\tfrac{n}{2}) + c_2 \quad k=1$$
$$= T(\tfrac{n}{4}) + c_2 + c_2 \quad 2$$
$$= T(\tfrac{n}{8}) + c_2 + c_2 + c_2 \quad 3$$
$$k \rightarrow T(\tfrac{n}{2^k}) + k \cdot c_2$$

3. Find the closed-form expression by setting *the number of expansions* to a value which reduces to a base case

$$1 = \frac{n}{2^k}, \quad 2^k = n, \quad \log_2 2^k = \log_2 n, \quad k = \log_2 n$$

$$T(\tfrac{n}{2^k}) + k c_2$$

$$T(\tfrac{n}{2^{\log_2 n}}) + \log_2 n \cdot c_2 = c_1 + \log_2 n \, c_2 \in O(\log n)$$

Lecture Outline

- Algorithm Analysis
  - Review: Amortized bounds
  - Where We've Come
  - Recurrences
    - Linear Search example
    - Binary Search example
    - **Binary & Linear Sum example**
- Priority Queue ADT

Summing an Array, Again (1 of 5)

Two "obviously" linear algorithms:

Iterative:

O(n)

```
int sum(int[] arr) {
  int ans = 0;
  for (int i=0; i < arr.length; ++i)
    ans += arr[i];
  return ans;
}
```

Recursive:

O(n)

```
int sum(int[] arr) {
  return help(arr,0);
}
int help(int[]arr,int i) {
  if (i == arr.length)
    return 0;
  return arr[i] + help(arr, i+1);
}
```

Summing an Array, Again (2 of 5)

- What about a binary version of **sum**?
  - Can we get a BinarySearch-like runtime?

$T(1) = C_1$

$T(n) = C_2 + 2T(\frac{n}{2})$

return 2· help(. ~~) → T(?)

```
int sum(int[] arr) {
    return help(arr, 0, arr.length);
}
int help(int[] arr, int lo, int hi) {
  0 if(lo == hi)    return 0;
  1 if(lo == hi-1) return arr[lo];
    int mid = (hi+lo)/2;
    return help(arr, lo, mid) + help(arr, mid, hi);
}
```

Summing an Array, Again (3 of 5)

$T(1) = c_2$ k

$T(n) = 2T(\frac{n}{2}) + c_1$ 1

$= 2\cdot(2T(\frac{n}{4}) + c_1) + c_1 = 4T(\frac{n}{4}) + 3c_1$ 2

$= 4\cdot(2T(\frac{n}{8}) + c_1) + 3c_1 = 8T(\frac{n}{8}) + 7c_1$ 3

. . . . .

$= 2^k T(\frac{n}{2^k}) + (2^k - 1)c_1$ k

$1 = \frac{n}{2^k}$

$k = \log_2 n$

$= 2^{\log_2 n} T(\frac{n}{2^{\log_2 n}}) + (2^{\log n} - 1)c_1$

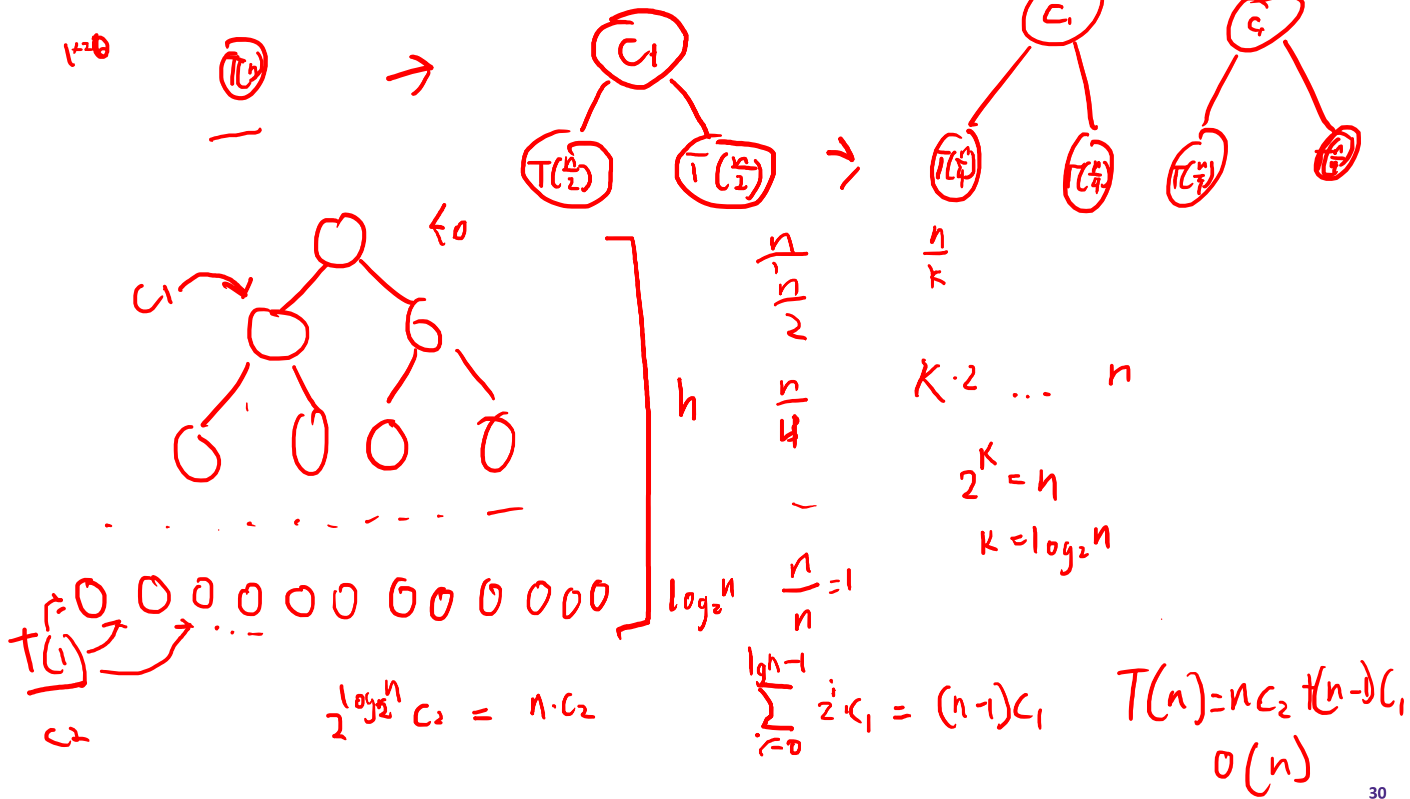
$= n \cdot c_2 + (n-1)c_1 = n \cdot c_2 + n \cdot c_1 - c_1 \in O(n)$

Technique #2: Tree Method

- Idea: We'll do the same reasoning, but give ourselves a visual to make the organization easier

- We'll make a **tree**
  - Each node of the tree represents one recursive call
  - The children of that node are the new recursive calls made

Summing an Array, Again (4 of 5)



Summing an Array, Again (5 of 5)

- Runtime is: O(n)

- Observation: it adds each number once while doing little else
  - Can't do better than O(n); have to read whole array!

```
int sum(int[] arr) {
   return help(arr, 0, arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo == hi)   return 0;
   if(lo == hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr, lo, mid) + help(arr, mid, hi);
}
```

Parallelism Teaser

- But suppose we could do two recursive calls *at the same time*

- If you have as much parallelism as needed, the recurrence becomes
  - T(n) = O(1) + **1** T(n/2) = O(log n)

```
int sum(int[] arr) {
    return help(arr, 0, arr.length);
}
int help(int[] arr, int lo, int hi) {
    if(lo == hi)   return 0;
    if(lo == hi-1) return arr[lo];
    int mid = (hi+lo)/2;
    return help(arr, lo, mid) + help(arr, mid, hi);
}
```

Really Common Recurrences

Recurrence Relation	*Big O Bounds*	*Name*	*Example*
T(n) = O(1) + T(n/2)	O(log n)	Logarithmic	Binary Search
T(n) = O(1) + T(n-1)	O(n)	Linear	Sum (v1: “Recursive Sum”)
T(n) = O(1) + 2T(n/2)	O(n)	Linear	Sum (v2: “Recursive Binary Sum”)
T(n) = O(n) + T(n/2)	O(n)	Linear	
T(n) = O(n) + 2T(n/2)	O(n log n)	Loglinear	MergeSort
T(n) = O(n) + T(n-1)	$O(n^2)$	Quadratic	
T(n) = O(1) + 2T(n-1)	$O(2^n)$	Exponential	Fibonacci

Lecture Outline

- Algorithm Analysis
  - Review: Amortized bounds
  - Where We've Come
  - Recurrences
- **Priority Queue ADT**

pollev.com/332summer :: tinyurl.com/332-06-29A

A Scenario

- What is the difference between waiting for service at a pharmacy versus an ER?
  - Pharmacies usually follow the rule “First Come, First Served”
  - Emergency Rooms assign priorities based on each individual's need

A New ADT: Priority Queue

- See Weiss Chapter 6

- A **priority queue** holds *compare-able data*
  - Unlike lists, stacks, and queues, we need to *compare items*
    - Given *x* and *y:* is *x* less than, equal to, or greater than *y?*
    - Much of this course will require comparable items: e.g. sorting
  - Typically two fields: the *priority* and the *data*

- Aside: we will use integers as priority and data
  - For simplicity in lecture, we'll suppose data are **`int`**s *and* that same **`int`** value is also the priority
    - **`int`** priorities are common, but really just need `Comparable`
  - Not having "other data" is very rare
    - Example: print job has a priority *and* the file to print

Priority Queue ADT

Priority Queue ADT. A collection storing a set of elements and their priority.

- A PQ has a size defined as the number of elements in the set
- You can add elements (and their priorities)
- You cannot access or remove arbitrary elements, only the element with the min priority

Primary Operations:

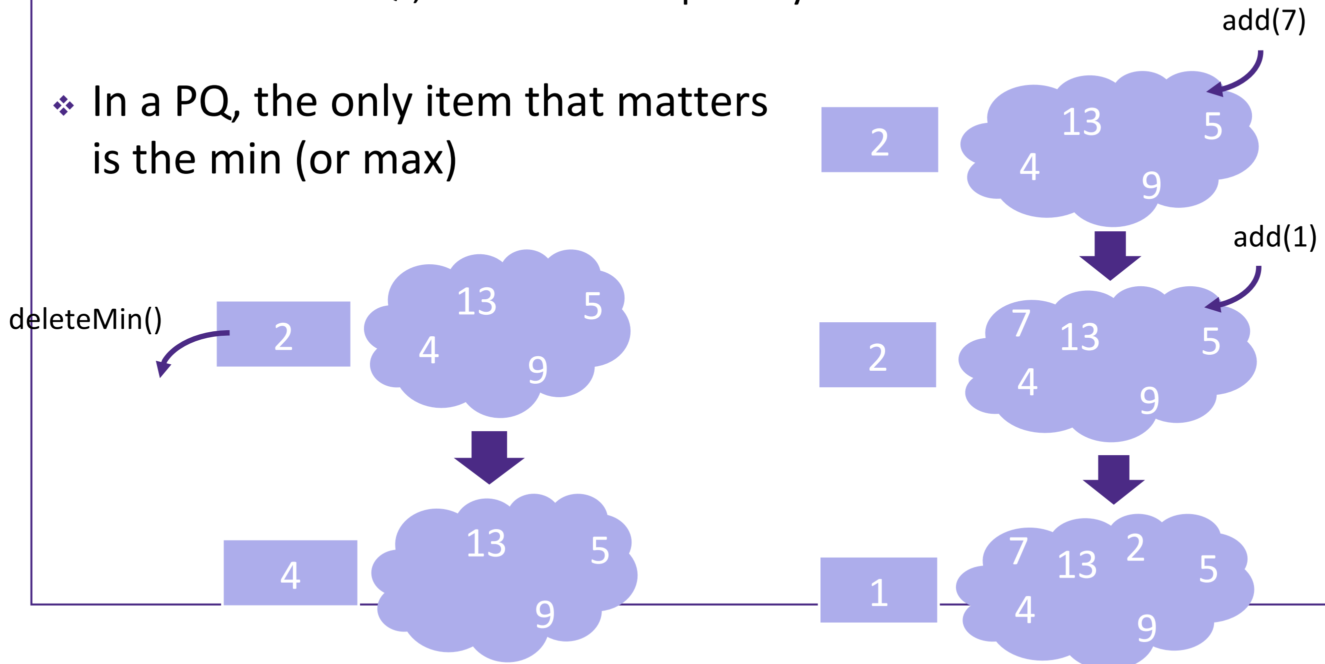
- `add`
- `deleteMin`

Key property:

- `deleteMin` removes and returns the "most important" item (lowest priority value)
- Can resolve ties arbitrarily

Priority Queues

- In lecture, we will study **min priority queues** but you may also see **max priority queues**
  - Same as minPQs, but invert the priority
- In a PQ, the only item that matters is the min (or max)



Priority Queue: Example

`add` *a* with priority *5*

`add` *b* with priority *3*

`add` *c* with priority *4*

w = **`deleteMin`**

x = **`deleteMin`**

`add` *d* with priority *2*

`add` *e* with priority *6*

y = **`deleteMin`**

z = **`deleteMin`**

```
after execution:
6->e

w = b
x = c
y = d
z = a
```

Analogy: `add` is like `enqueue`, and `deleteMin` is like `dequeue`

Unlike queues, priority queues use *priorities* instead of *time-of-insertion* to order its elements

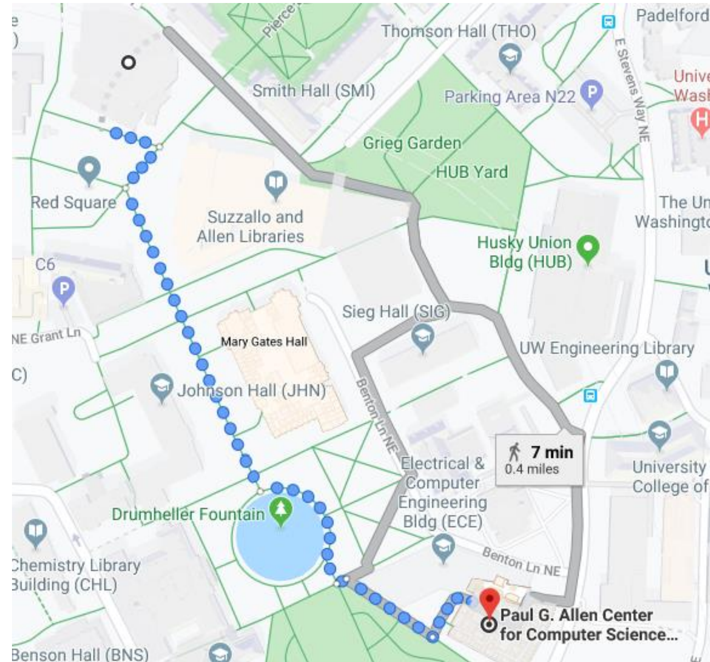
Priority Queue: Applications

- Run multiple programs in the operating system
  - “critical” before “interactive” before “compute-intensive”
- Triage (or treat) hospital patients in order of severity
- Order print jobs in order of decreasing length?
- Forward network packets by order of urgency
- Identify most frequently-used symbols for data compression
- Sorting!
  - **`add`** all elements, then repeatedly **`deleteMin`**

Priority Queue: More Applications

- Used heavily in **greedy algorithms**, where each phase of the algorithm picks the locally optimum solution

- Example: route finding
  - Represent a map as a series of *segments*
  - At each intersection, ask which segment gets you closest to the destination (ie, has max priority or min distance)



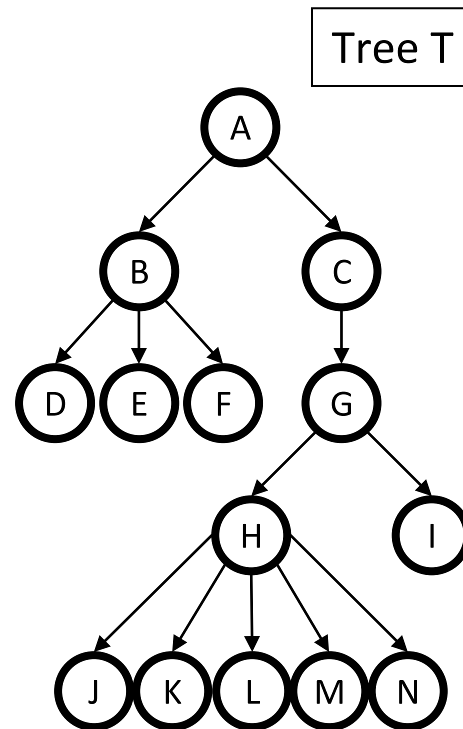
Lecture Outline

- Algorithm Analysis
  - Review: Amortized bounds
  - Where We've Come
  - Recurrences
- Priority Queue ADT
  - **Tree Review**

pollev.com/332summer :: tinyurl.com/332-06-29A

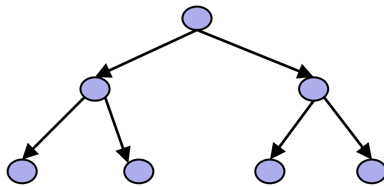
Review: Tree Terminology

- root(T):
- leaves(T):
- children(B):
- parent(H):
- siblings(E):
- ancestors(F):
- descendents(G):
- subtree(G):
- depth(B):
- height(G):
- height(T):
- degree(B):
- branching factor(T):

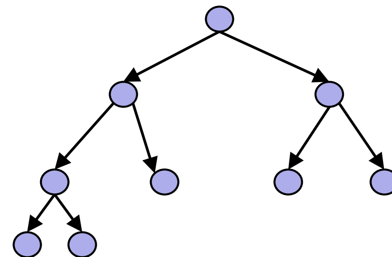


Types of Trees

Binary tree	Every node has ≤ 2 children
N-ary tree	Every node has ≤ n children
Perfect tree	Every row is completely full
Complete tree	All rows except possibly the bottom are completely full. The bottom row is filled from left to right



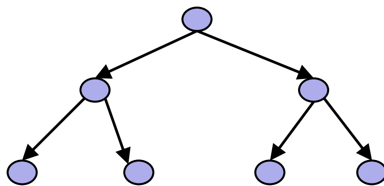
Perfect Tree



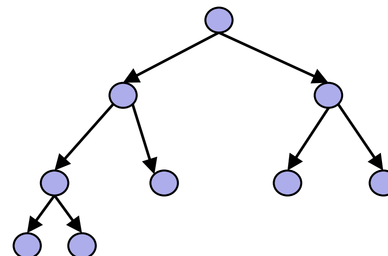
Complete Tree

Perfect Tree Properties

Height	Number of Nodes	Number of Leaves
1		
2		
3		
4		
h		



Perfect Tree



Complete Tree