Lecture 15: Midterm Review

CSE 373 Data Structures and Algorithms

Announcements

1 fill out the poll ☺

Midterm

1. NO LATE ASSIGNMENTS – DUE May 7th at 11:59pm

2. Closed course staff

- can ask clarifying questions

P2 succcckcked

extend late turn in for P2 until Monday night at 11:59pm

max usage of 3 late days on the assignment

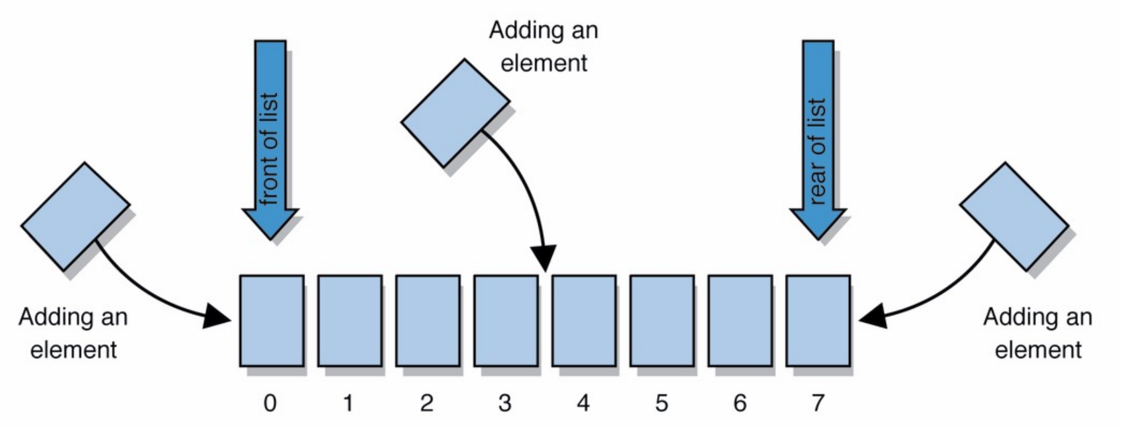
Abstract Data Types (ADT)

Abstract Data Types

- An abstract definition for expected operations and behavior
- Defines the input and outputs, not the implementations

Review: List - a collection storing an ordered sequence of elements

- each element is accessible by a 0-based index
- a list has a size (number of elements that have been added)
- elements can be added to the front, back, or elsewhere
- in Java, a list can be represented as an ArrayList object

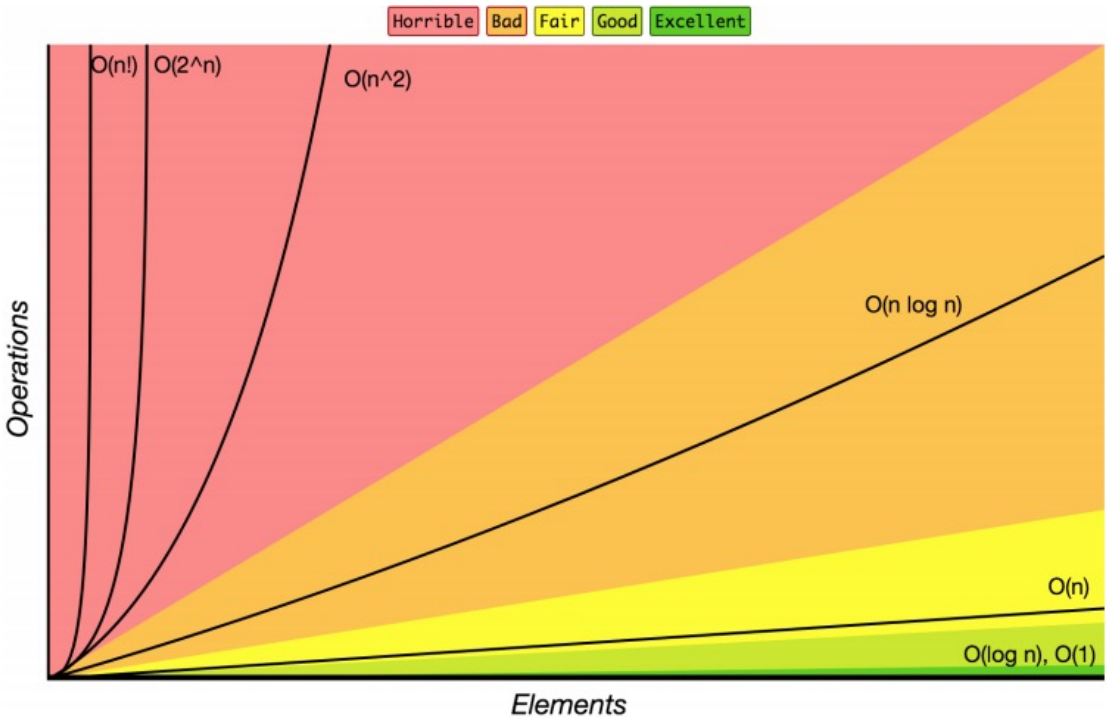


Review: Complexity Class

Note: You don't have to understand all of this right now – we'll dive into it soon.

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion



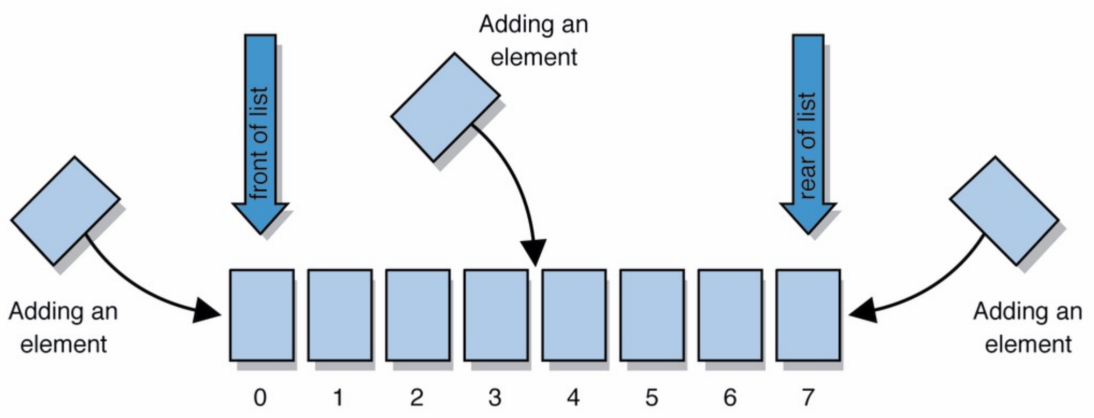
Case Study: The List ADT

list: a collection storing an ordered sequence of elements.

- Each item is accessible by an index.
- A list has a size defined as the number of elements in the list

Expected Behavior:

- **get(index):** returns the item at the given index
- **set(value, index):** sets the item at the given index to the given value
- **append(value):** adds the given item to the end of the list
- **insert(value, index):** insert the given item at the given index maintaining order
- **delete(index):** removes the item at the given index maintaining order
- **size():** returns the number of elements in the list



Case Study: List Implementations

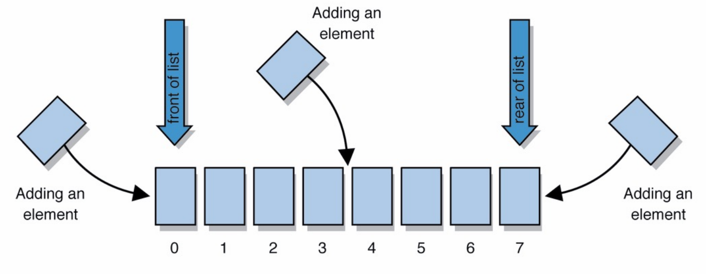
List ADT

state

Set of ordered items
Count of items

behavior

<u>get(index)</u> return item at index
<u>set(item, index)</u> replace item at index
<u>append(item)</u> add item to end of list
<u>insert(item, index)</u> add item at index
<u>delete(index)</u> delete item at index
<u>size()</u> count of items



ArrayList

uses an Array as underlying storage

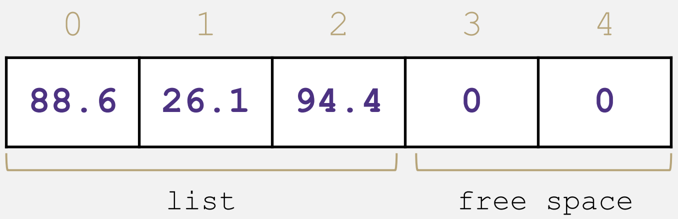
ArrayList<E>

state

data[]
size

behavior

<u>get</u> return data[index]
<u>set</u> data[index] = value
<u>append</u> data[size] = value, if out of space grow data
<u>insert</u> shift values to make hole at index, data[index] = value, if out of space grow data
<u>delete</u> shift following values forward
<u>size</u> return size



LinkedList

uses nodes as underlying storage

LinkedList<E>

state

Node front
size

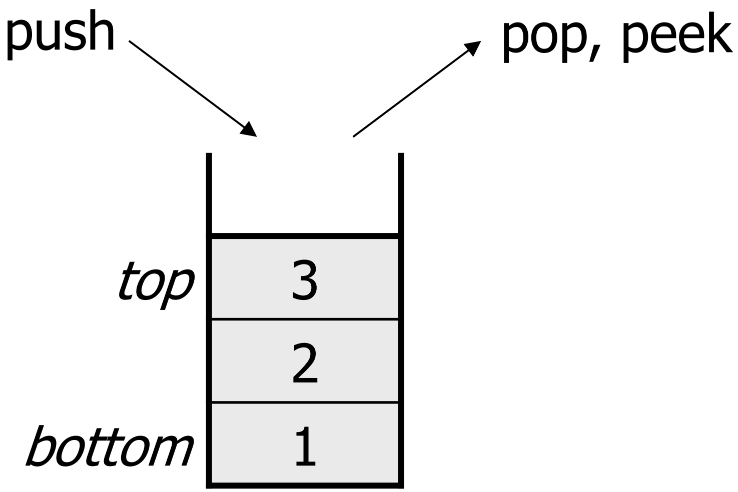
behavior

<u>get</u> loop until index, return node's value
<u>set</u> loop until index, update node's value
<u>append</u> create new node, update next of last node
<u>insert</u> create new node, loop until index, update next fields
<u>delete</u> loop until index, skip node
<u>size</u> return size

Review: What is a Stack?

stack: A collection based on the principle of adding elements and retrieving them in the opposite order.

- Last-In, First-Out ("LIFO")
- Elements are stored in order of insertion.
  - We do not think of them as having indexes.
- Client can only add/remove/examine the last element added (the "top").



Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

supported operations:

- **push(item)**: Add an element to the top of stack
- **pop()**: Remove the top element and returns it
- **peek()**: Examine the top element without removing it
- **size()**: how many items are in the stack?
- **isEmpty()**: true if there are 1 or more items in stack, false otherwise

Implementing a Stack with an Array

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

ArrayStack<E>

state

data[]
size

behavior

push data[size] = value, if out of room grow data
pop return data[size - 1], size-1
peek return data[size - 1]
size return size
isEmpty return size == 0

Big O Analysis

pop()	O(1) Constant
peek()	O(1) Constant
size()	O(1) Constant
isEmpty()	O(1) Constant
push()	O(N) linear if you have to resize O(1) otherwise

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

numberOfItems = 2

Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst possible runtime of the “push()” operation will be?

Implementing a Stack with Nodes

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

LinkedStack<E>

state

```
Node top
size
```

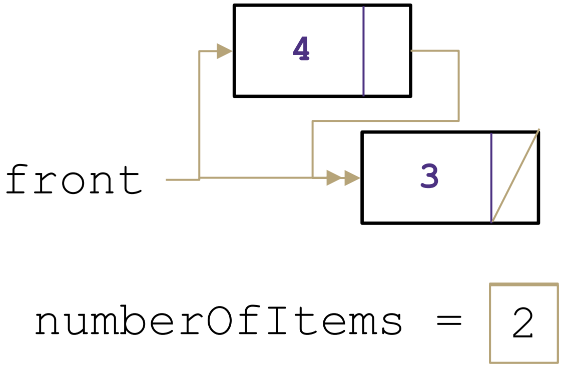
behavior

```
push add new node at top
pop return and remove node at top
peek return node at top
size return size
isEmpty return size == 0
```

Big O Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	O(1) Constant

```
push(3)
push(4)
pop()
```



Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst possible runtime of the "push()" operation will be?

Review: What is a Queue?

queue: Retrieves elements in the order they were added.

- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

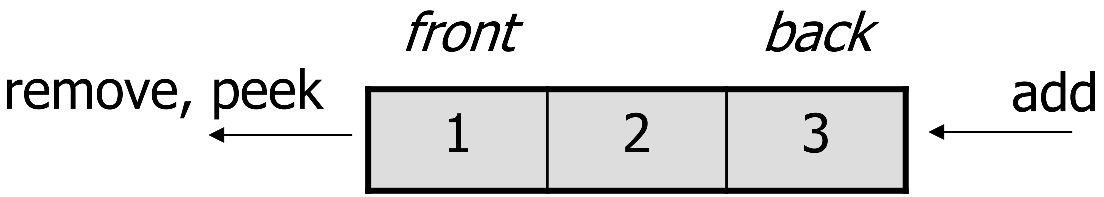
Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?



supported operations:

- **add(item):** aka "enqueue" add an element to the back.
- **remove():** aka "dequeue" Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise

Implementing a Queue with an Array

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

ArrayQueue<E>

```
state
  data[]
  Size
  front index
  back index
behavior
 add - data[size] = value, if
 out of room grow data
 remove - return data[size -
 1], size-1
 peek - return data[size - 1]
 size - return size
 isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(N) linear if you have to resize O(1) otherwise

```
add(5)
add(8)
add(9)
remove()
```

0	1	2	3	4
5	8	9		

```
numberOfItems = 3
front = 1
back = 2
```

Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst possible runtime of the "add()" operation will be?

Implementing a Queue with Nodes

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

LinkedQueue<E>

state

```
Node front
Node back
size
```

behavior

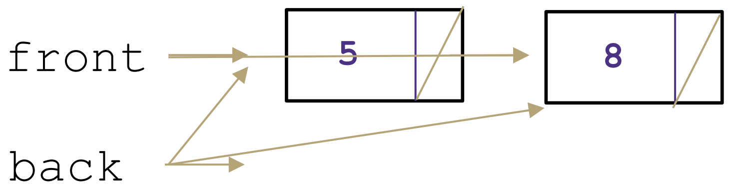
```
add - add node to back
remove - return and remove
node at front
peek - return node at front
size - return size
isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(1) Constant

```
numberOfItems = 2
```

```
add(5)
add(8)
remove()
```



Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst case runtime of the "add()" operation will be?

Review: Dictionaries

Dictionary ADT

state
Set of items & keys
Count of items

behavior
put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

Why are we so obsessed with Dictionaries?

When dealing with data:

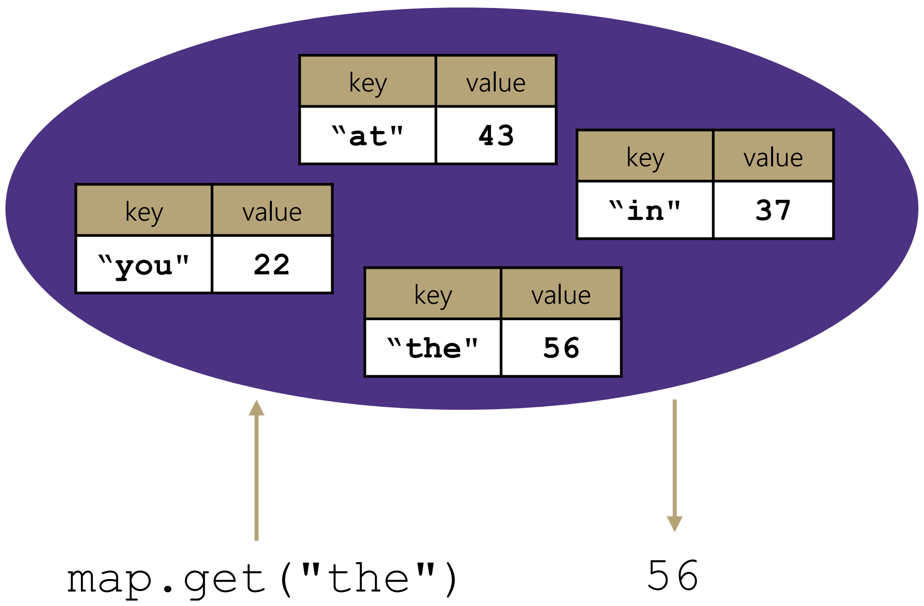
- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

Operation		ArrayList	LinkedList	HashTable	BST	AVLTree
put(key,value)	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
get(key)	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
remove(key)	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$

Review: Maps

map: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.

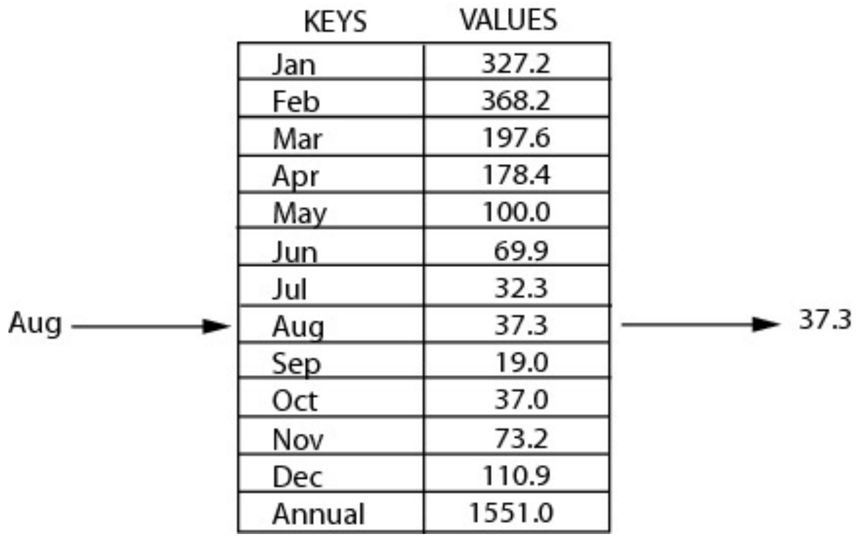
- a.k.a. "dictionary"



Dictionary ADT
state Set of items & keys Count of items
behavior put(key, item) add item to collection indexed with key get(key) return item associated with key containsKey(key) return if key already in use remove(key) remove item and associated key size() return count of items

supported operations:

- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - **if the map previously had a mapping for the given key, old value is replaced.**
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(key): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value



Implementing a Map with an Array

Map ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

ArrayMap<K, V>

state

`Pair<K, V>[] data`

behavior

`put` find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
`get` scan all pairs looking for given key, return associated item if found
`containsKey` scan all pairs, return if key is found
`remove` scan all pairs, replace pair to be removed with last pair in collection
`size` return count of items in dictionary

```
containsKey('c')
get('d')
put('b', 97)
put('e', 20)
```

0	1	2	3	4
('a', 1)	('b', 97)	('c', 3)	('d', 4)	('e', 20)

Big O Analysis – (if key is the last one looked at / not in the dictionary)

`put()`	O(N) linear
`get()`	O(N) linear
`containsKey()`	O(N) linear
`remove()`	O(N) linear
`size()`	O(1) constant

Big O Analysis – (if the key is the first one looked at)

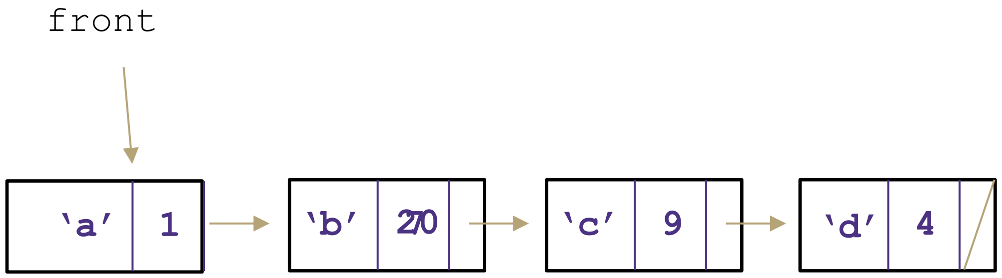
`put()`	O(1) constant
`get()`	O(1) constant
`containsKey()`	O(1) constant
`remove()`	O(1) constant
`size()`	O(1) constant

Implementing a Map with Nodes

Map ADT
state Set of items & keys Count of items **behavior** put(key, item) add item to collection indexed with key get(key) return item associated with key containsKey(key) return if key already in use remove(key) remove item and associated key size() return count of items

LinkedMap<K, V>
state front size **behavior** put if key is unused, create new with pair, add to front of list, else replace with new value get scan all pairs looking for given key, return associated item if found containsKey scan all pairs, return if key is found remove scan all pairs, skip pair to be removed size return count of items in dictionary

```
containsKey('c')
get('d')
put('b', 20)
```



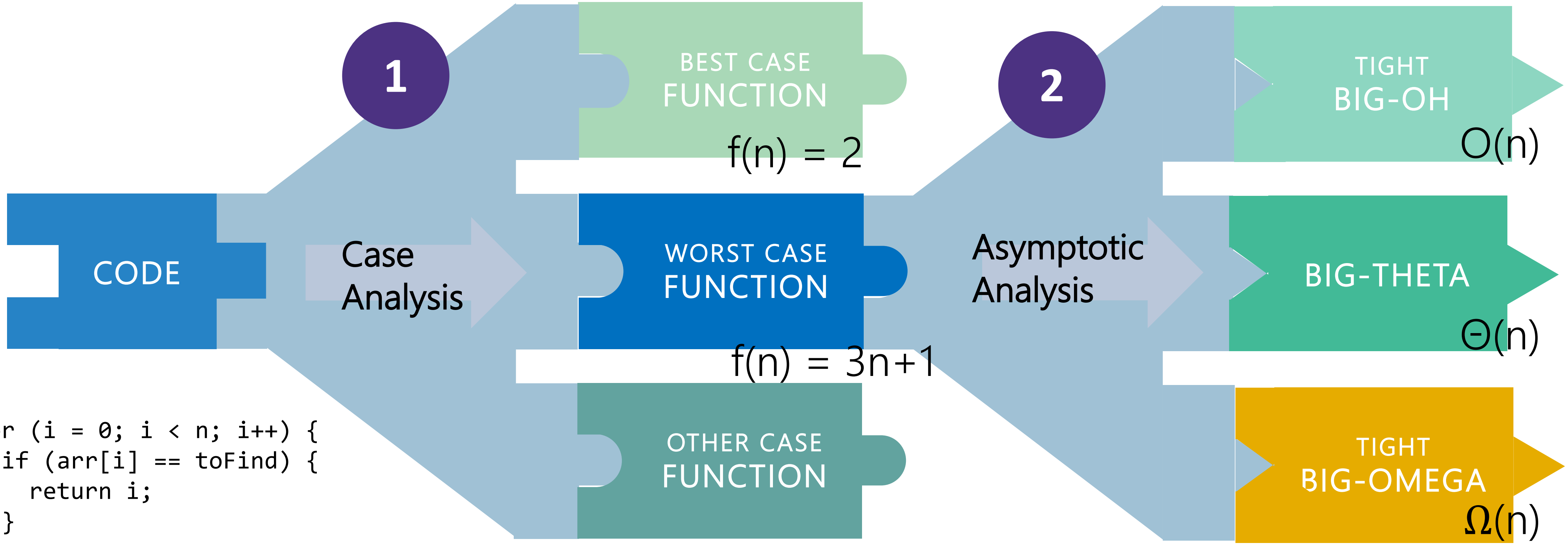
Big O Analysis – (if key is the last one looked at / not in the dictionary)

put()	O(N) linear
get()	O(N) linear
containsKey()	O(N) linear
remove()	O(N) linear
size()	O(1) constant

Big O Analysis – (if the key is the first one looked at)

put()	O(1) constant
get()	O(1) constant
containsKey()	O(1) constant
remove()	O(1) constant
size()	O(1) constant

Algorithmic Analysis Roadmap




```
for (i = 0; i < n; i++) {
  if (arr[i] == toFind) {
    return i;
  }
}
return -1;
```

Code Modeling Example 2

```
public void method2(int n) {
    int sum = 0;                    +1
    int i = 0;                      +1
    while (i < n) {                 +1
        int j = 0;                  +1
        while (j < n) {             +1
            if (j % 2 == 0) {       +2
                // do nothing
            }
            sum = sum + (i * 3) + j;   +4
            j = j + 1;              +2
        }
        i = i + 1;                  +2
    } return sum;                   +1
}
```

This inner loop runs n times

+9 *n

This outer loop runs n times

9n + 4 *n

$f(n) = (9n+4)n + 3$

Review Oh, and Omega, and Theta, oh my

Big-Oh is an **upper bound**

- My code takes at most this long to run

Big-Omega is a **lower bound**

- My code takes at least this long to run

Big Theta is “**equal to**”

- My code takes “exactly”* this long to run
- *Except for constant factors and lower order terms
- Only exists when Big-Oh == Big-Omega!

Big-Oh

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Big-Omega

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \geq c \cdot g(n)$$

Big-Theta

$f(n)$ is $\Theta(g(n))$ if

$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

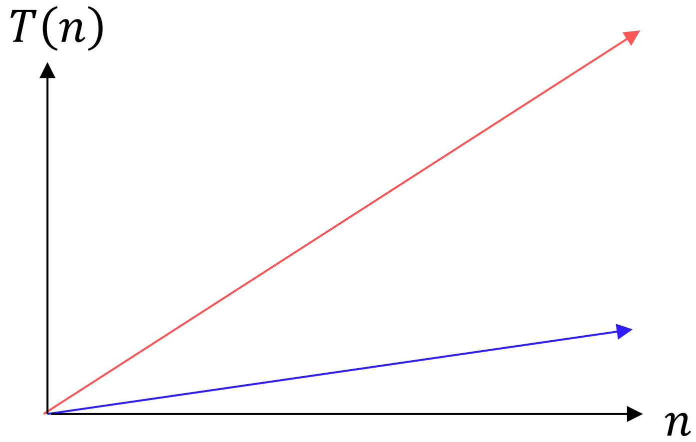
(in other words: there exist positive constants $c1, c2, n_0$ such that for all $n \geq n_0$)

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

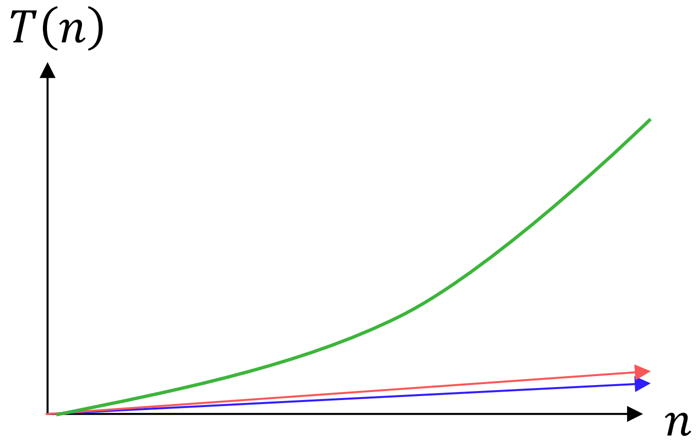
Function growth

Imagine you have three possible algorithms to choose between.
Each has already been reduced to its mathematical model

$f(n) = n$ $g(n) = 4n$ $h(n) = n^2$

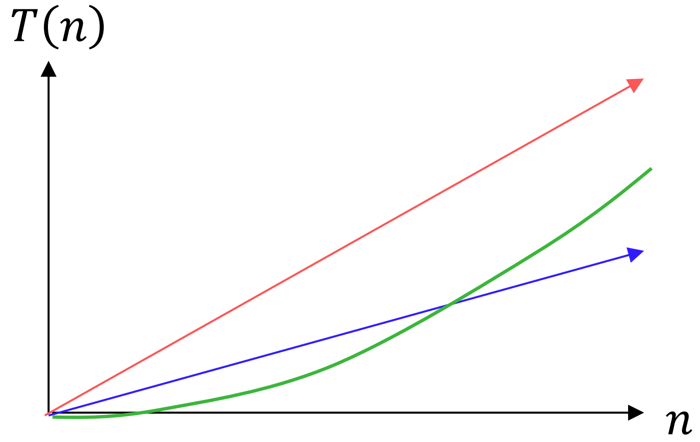


The growth rate for f(n) and g(n) looks very different for small numbers of input



...but since both are linear eventually look similar at large input sizes

whereas h(n) has a distinctly different growth rate



But for very small input values h(n) actually has a slower growth rate than either f(n) or g(n)

Examples

$4n^2 \in \Omega(1)$

true

$4n^2 \in \Omega(n)$

true

$4n^2 \in \Omega(n^2)$

true

$4n^2 \in \Omega(n^3)$

false

$4n^2 \in \Omega(n^4)$

false

$4n^2 \in O(1)$

false

$4n^2 \in O(n)$

false

$4n^2 \in O(n^2)$

true

$4n^2 \in O(n^3)$

true

$4n^2 \in O(n^4)$

true

Big-O

$f(n) \in O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,
$f(n) \leq c \cdot g(n)$

Big-Omega

$f(n) \in \Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,
$f(n) \geq c \cdot g(n)$

Big-Theta

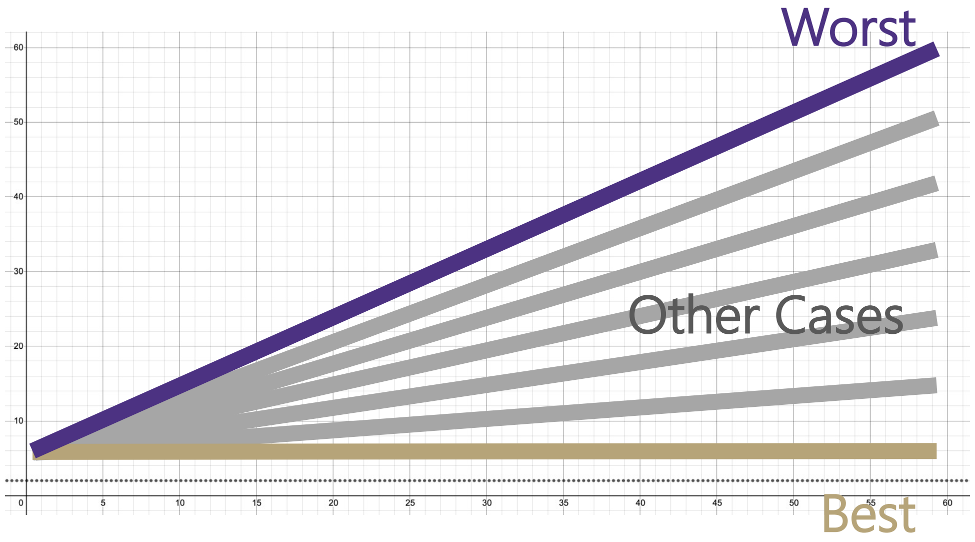
$f(n) \in \Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Case Analysis

Case: a description of inputs/state for an algorithm that is specific enough to build a code model (runtime function) whose only parameter is the input size

- Case Analysis is our tool for reasoning about **all variation other than n!**
- Occurs during the code → function step instead of function → O/Ω/Θ step!

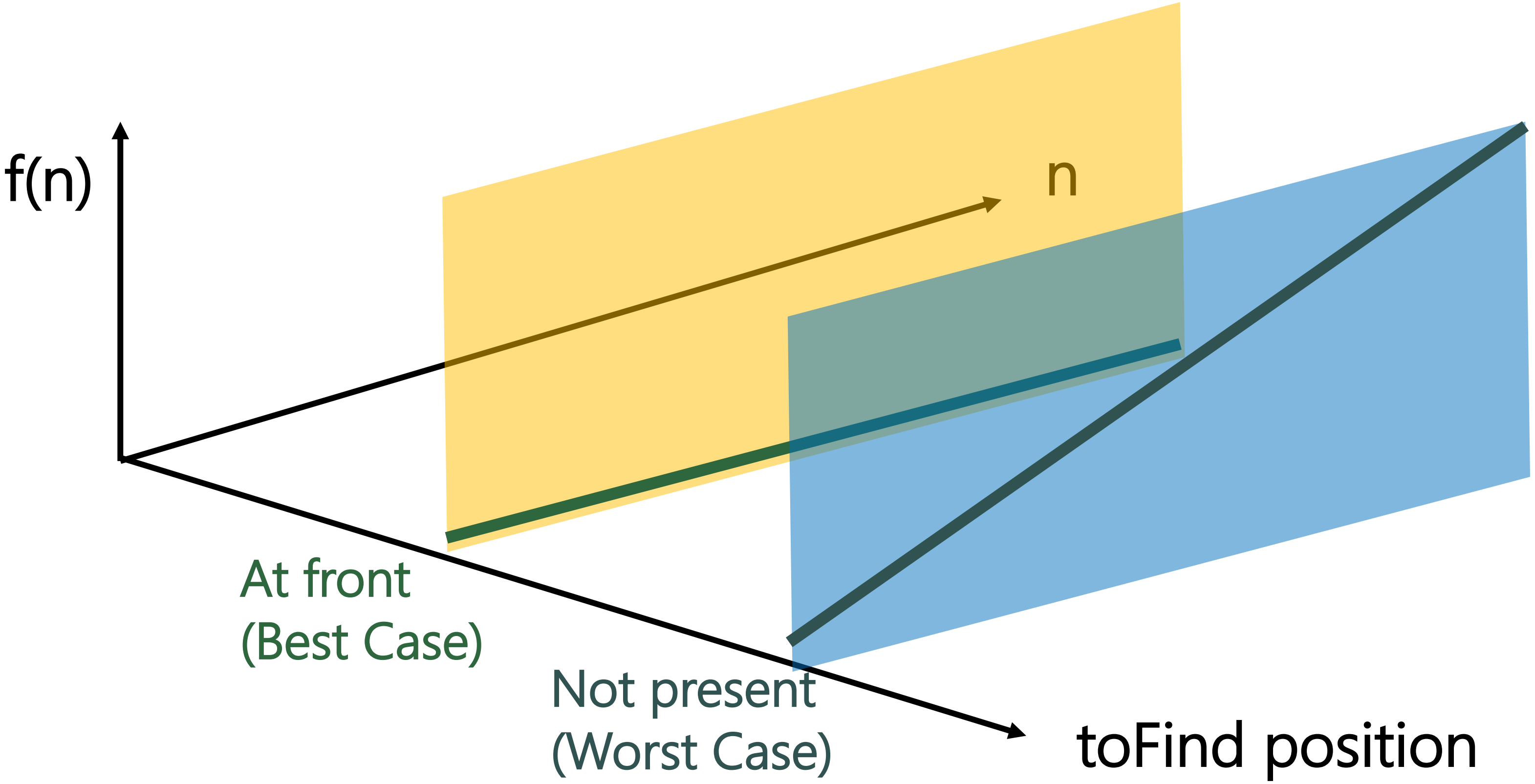
- (Best Case: fastest/Worst Case: slowest) that our code could finish on input of size n.
- Importantly, *any* position of toFind in arr could be its own case!
  - For this simple example, probably don't care (they all still have bound O(n))
  - But intermediate cases will be important later



Review When to do Case Analysis?

Imagine a 3-dimensional plot

- Which case we're considering is one dimension
- Choosing a case lets us take a "slice" of the other dimensions: n and f(n)
- We do asymptotic analysis on each slice in step 2



How to do case analysis

1. Look at the code, understand how thing could change depending on the input.
   - How can you exit loops early?
   - Can you return (exit the method) early?
   - Are some if/else branches much slower than others?
2. Figure out what inputs can cause you to hit the (best/worst) parts of the code.
3. Now do the analysis like normal!

Warm Up!

Please fill out the Poll at- pollev.com/21sp373

What's the theta bound for the runtime function for this piece of code?

```
public void method1(int n) {
    if (n <= 100) {
        System.out.println(":3");
    } else {
        System.out.println(":D");
        for (int i = 0; i<16; i++) {
            method1(n / 4);
        }
    }
}
```

$$T(n) = \begin{cases} \textit{constant work} & \text{if } n \leq 100 \\ 16T\left(\frac{n}{4}\right) + \textit{constant work} & \text{otherwise} \end{cases}$$

a = 16, b = 4, c = 0

$\log_4 16 = 2$

$\log_4 16 > 0$

$T(n) \in \Theta\left(n^{\log_b a}\right)$

$\Theta\left(n^{\log_4 16}\right) = \boldsymbol{\Theta\left(n^2\right)}$

Master Theorem

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Meet the Recurrence

A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s)

It's a lot like recursive code:

- At least one base case and at least one recursive case
- Each case should include the values for n to which it corresponds
- The recursive case should reduce the input size in a way that eventually triggers the base case
- The cases of your recurrence usually correspond exactly to the cases of the code

$$T(n) = \begin{cases} 5 & \text{if } n < 3 \\ 2T\left(\frac{n}{2}\right) + 10 & \text{otherwise} \end{cases}$$

Tree Method

Draw out call stack, what is the input to each call? How much work is done by each call?

How much work is done at each layer?

64 for this example -> n work at each layer

Work is variable per layer, but across the entire layer work is constant - always n

How many layers are in our function call tree?

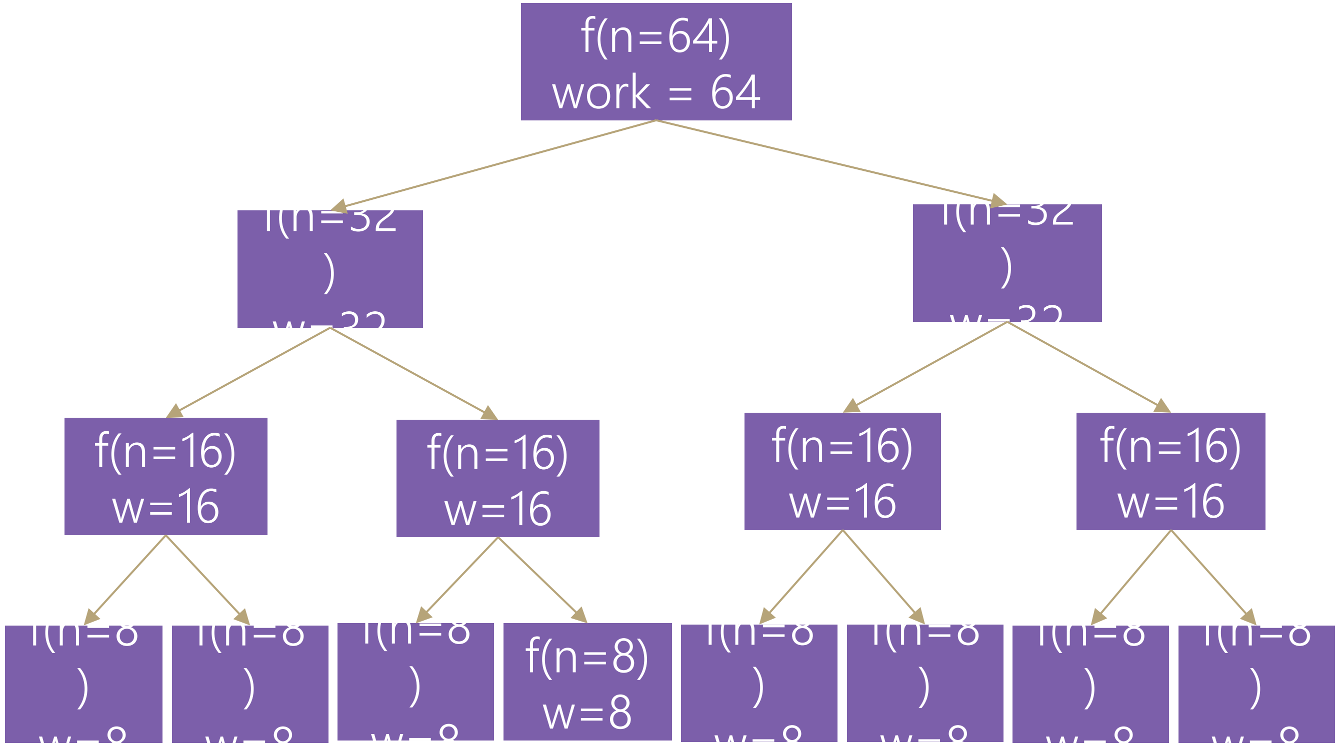
Hint: how many levels of recursive calls does it take *binary search* to get to the base case?

Height = $\log_2 n$

It takes $\log_2 n$ divisions by 2 for n to be reduced to the base case 1

$\log_2 64 = 6$ -> 6 levels of this tree

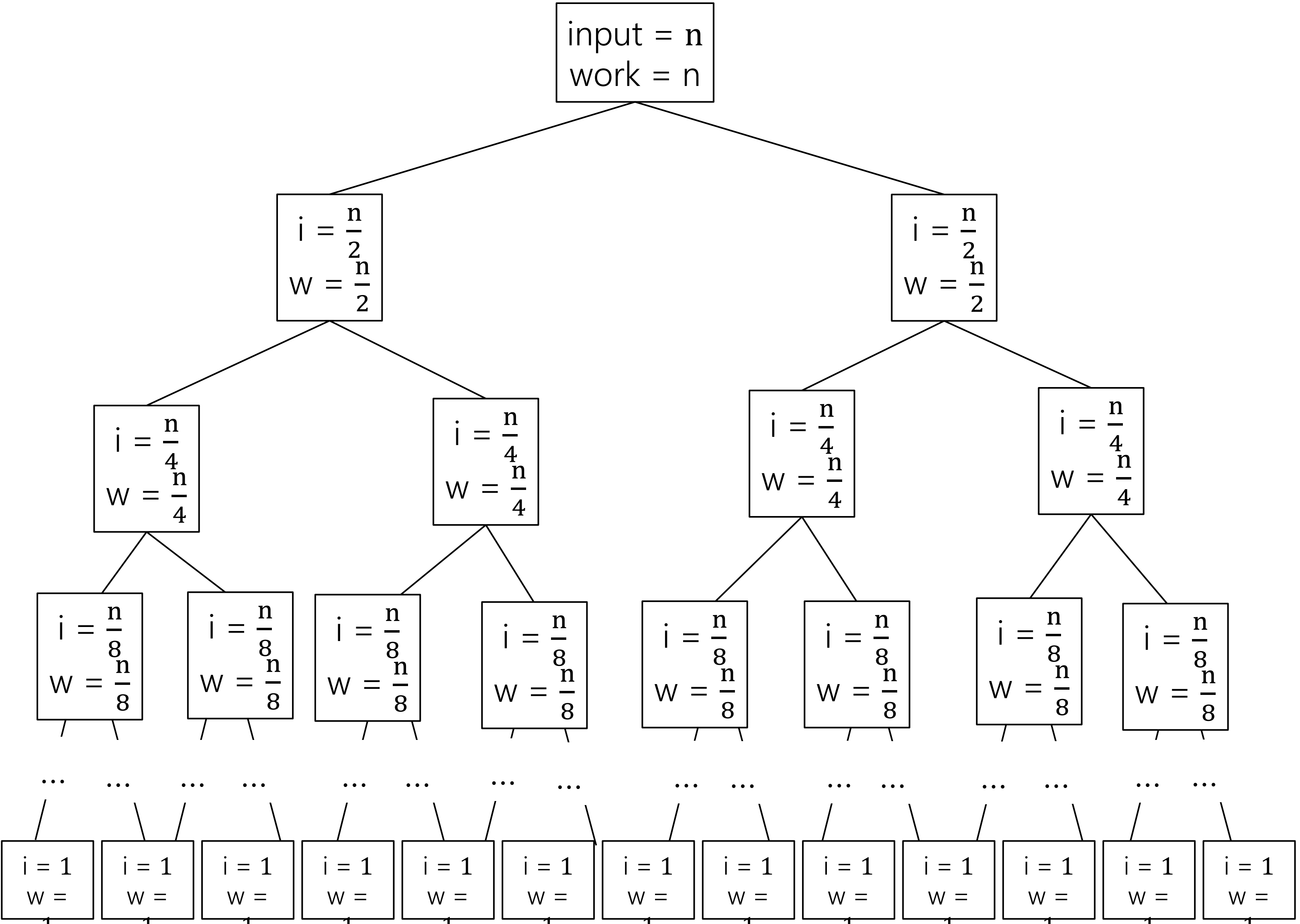
Merge Sort $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases}$



... and so on...

Tree Method

$$T(n) = \begin{cases} 1 \text{ when } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n \text{ otherwise} \end{cases}$$



Recursive level	How many nodes at each level?	How much work done by each node?	How much work across each level?
0	1	n	n
1	2	$\frac{n}{2}$	n
2	4	$\frac{n}{4}$	n
3	8	$\frac{n}{8}$	n
logn	n	1	n

Tree Method Practice

$$T(n) = \begin{cases} 1 \text{ when } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n \text{ otherwise} \end{cases}$$

1. What is the size of the input on level i? $\frac{n}{2^i}$

2. What is the work done by each node on the i^{th} recursive level? $\left(\frac{n}{2^i}\right)$

3. What is the number of nodes at level i? 2^i

4. What is the total work done at the i^{th} recursive level?

   $numNodes * workPerNode = 2^i\left(\frac{n}{2^i}\right) = n$

5. What value of i does the last level occur?

   $\frac{n}{2^i} = 1 \rightarrow n = 2^i \rightarrow i = \log_2 n$

6. What is the total work across the base case level?

   $numNodes * workPerNode = 2^{\log_2 n}(1) = n$

Level (i)	Number of Nodes	Work per Node	Work per Level
0	1	n	n
1	2	$\frac{n}{2}$	n
2	4	$\frac{n}{4}$	n
3	8	$\frac{n}{8}$	n
$\log_2 n$	n	1	

Combining it all together…

$$T(n) = \sum_{i=0}^{\log_2 n - 1} n + n = nlog_2 n + n = \Theta(\text{nlogn})$$

power of a log

$x^{\log_b y} = y^{\log_b x}$

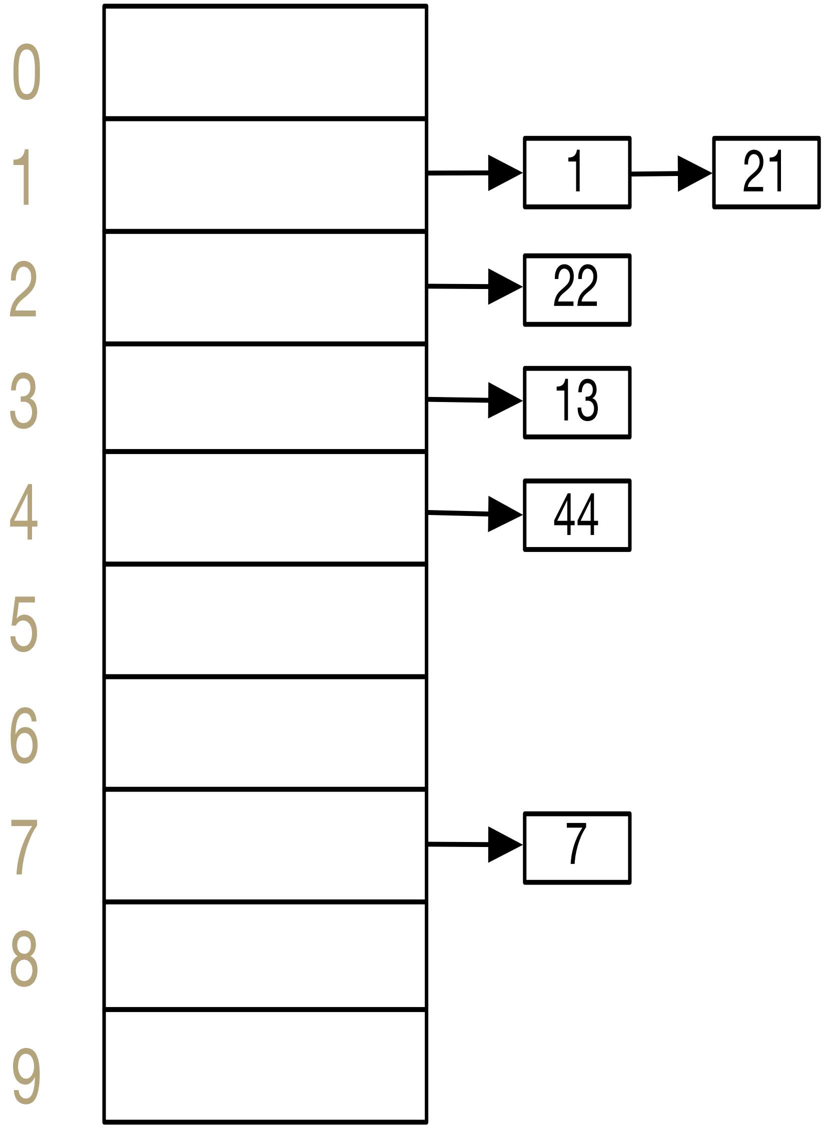
Summation of a constant

$\sum_{i=0}^{n-1} c = cn$

Separate chaining

Reminder: the implementations of put/get/containsKey are all very similar, and almost always will have the same complexity class runtime

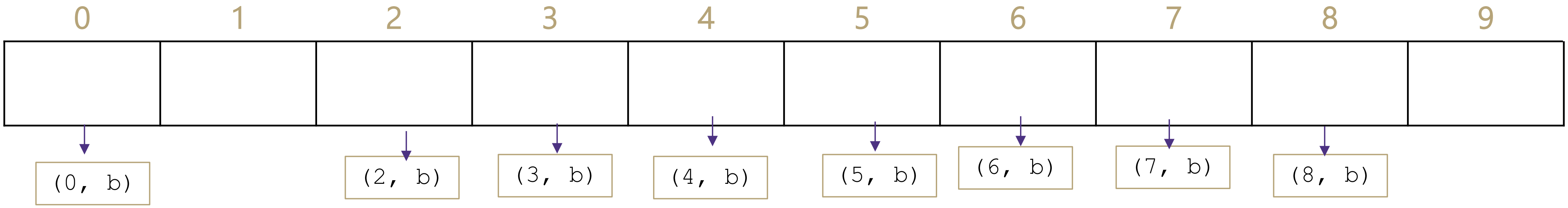
```
// some pseudocode
public boolean containsKey(int key) {
    int bucketIndex = key % data.length;
    loop through data[bucketIndex]
        return true if we find the key in
        data[bucketIndex]
    return false if we get to here (didn't
     find it)
}
```



<u>runtime analysis</u>
Are there different possible states for our Hash Map that make this code run slower/faster, assuming there are already n key-value pairs being stored?

Yes! If we had to do a lot of loop iterations to find the key in the bucket, our code will run slower.

A best case situation for separate chaining



It’s possible (and likely if you follow some best-practices) that everything is spread out across the buckets pretty evenly. This is the opposite of the last slide: when we have minimal collisions, our runtime should be less. For example, if we have a bucket with only 0 or 1 element in it, checking containsKey for something in that bucket will only take a constant amount of time.

We’re going to try a lot of stuff we can to make it more likely we achieve this beautiful state ☺.

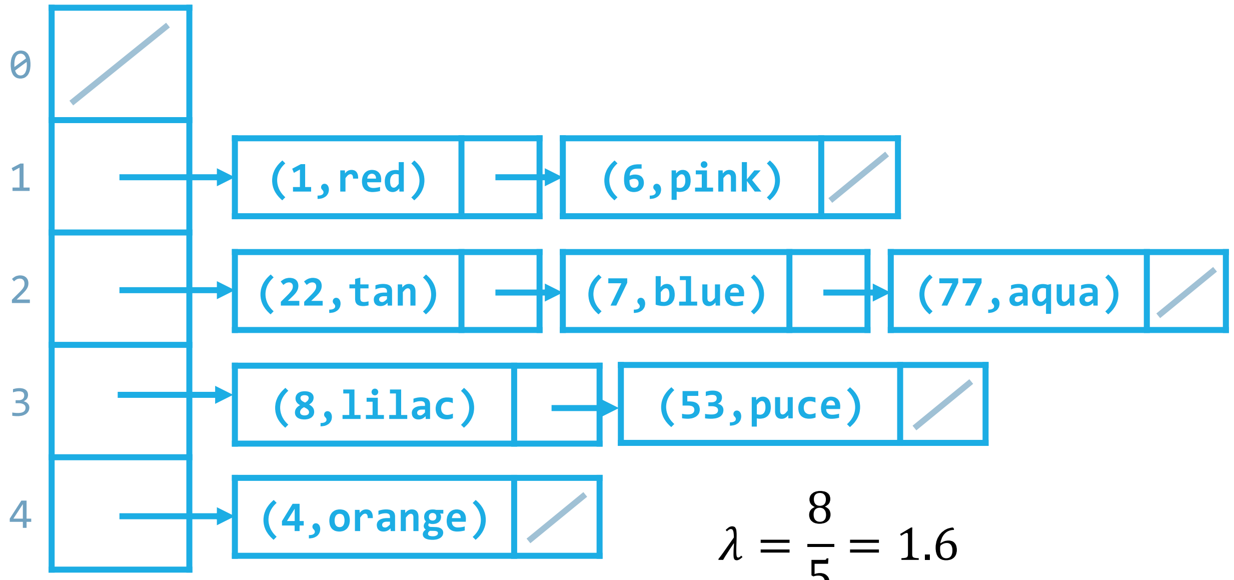
When to Resize?

In ArrayList, we were forced to resize when we ran out of room

- In SeparateChainingHashMap, never *forced* to resize, but we want to make sure the buckets don't get too long for good runtime

How do we quantify "too full"?

- Look at the average bucket size: number of elements / number of buckets



$$\lambda = \frac{8}{5} = 1.6$$

LOAD FACTOR λ

n: total number of key/value pairs
c: capacity of the array (# of buckets)

$$\lambda = \frac{n}{c}$$

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
38, 19, 8, 109, 10

8

0	1	2	3	4	5	6	7	8	9
10								38	109

Problem:
- Linear probing causes clustering
- Clustering causes more looping when probing

Primary Clustering
When probing causes long chains of occupied slots within a hash table

Quadratic Probing

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions
89, 18, 49, 58, 79, 27

0	1	2	3	4	5	6	7	8	9
		58	79				27	18	89 49

(49 % 10 + 0 * 0) % 10 = 9
(49 % 10 + 1 * 1) % 10 = 0

(58 % 10 + 0 * 0) % 10 = 8
(58 % 10 + 1 * 1) % 10 = 9
(58 % 10 + 2 * 2) % 10 = 2

(79 % 10 + 0 * 0) % 10 = 9
(79 % 10 + 1 * 1) % 10 = 0
(79 % 10 + 2 * 2) % 10 = 3

Now try to insert 9.

Uh-oh

Problems:
If $\lambda \geq \frac{1}{2}$ we might never find an empty spot
    Infinite loop!
Can still get clusters

Review: Handling Collisions

Solution 1: Chaining

Each space holds a "bucket" that can store multiple values. Bucket is often implemented with a LinkedList

Operation		Array w/ indices as keys
put(key,value)	best	O(1)
	average	O(1 + λ)
	worst	O(n)
get(key)	best	O(1)
	average	O(1 + λ)
	worst	O(n)
remove(key)	best	O(1)
	average	O(1 + λ)
	worst	O(n)

Average Case:
Depends on average number of elements per chain

Load Factor λ

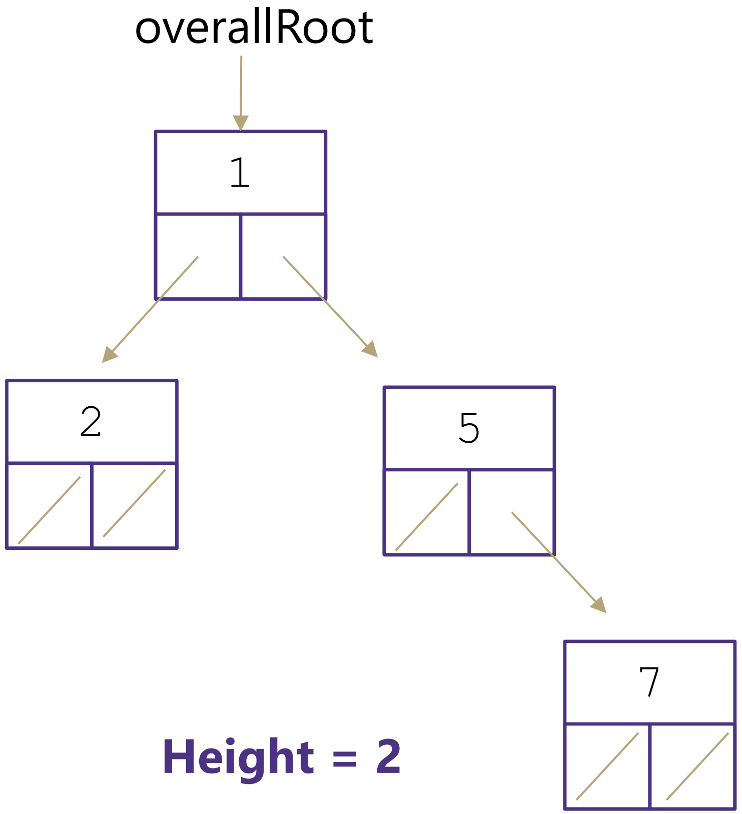
If n is the total number of key-value pairs

Let c be the capacity of array

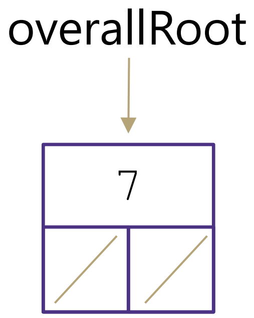
Load Factor $\lambda = \frac{n}{c}$

Tree Height

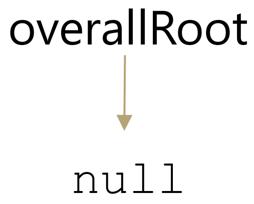
What is the height (the number of edges contained in the longest path from root node to some leaf node) of the following binary trees?



Height = 2



Height = 0



Height = -1 or NA

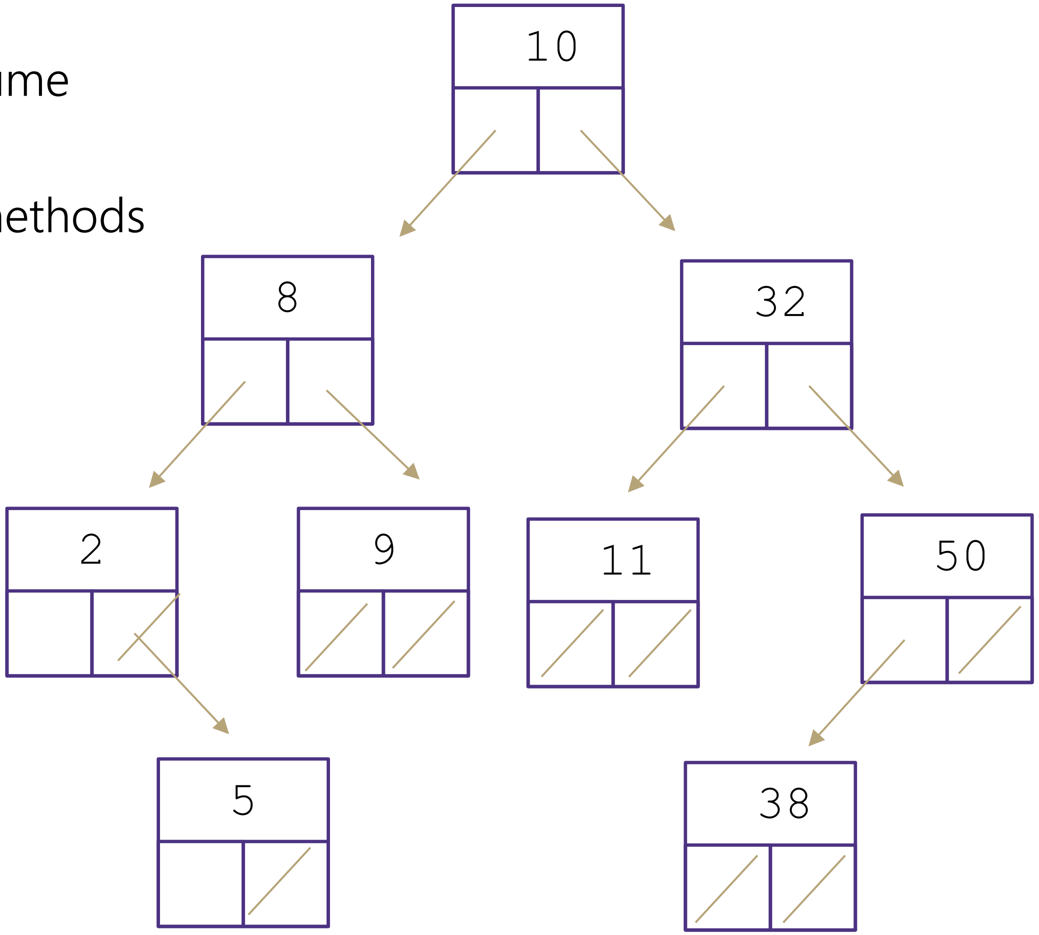
Binary Search Tree (BST)

Invariants (A.K.A. rules for your DS or algorithm)

- Things that are always true. If they're always true, you can assume them so that you can write simpler and more efficient code.
- You can also check invariants at the ends/beginnings of your methods to ensure that your state is valid and that everything is working.

Binary Search Tree invariants:

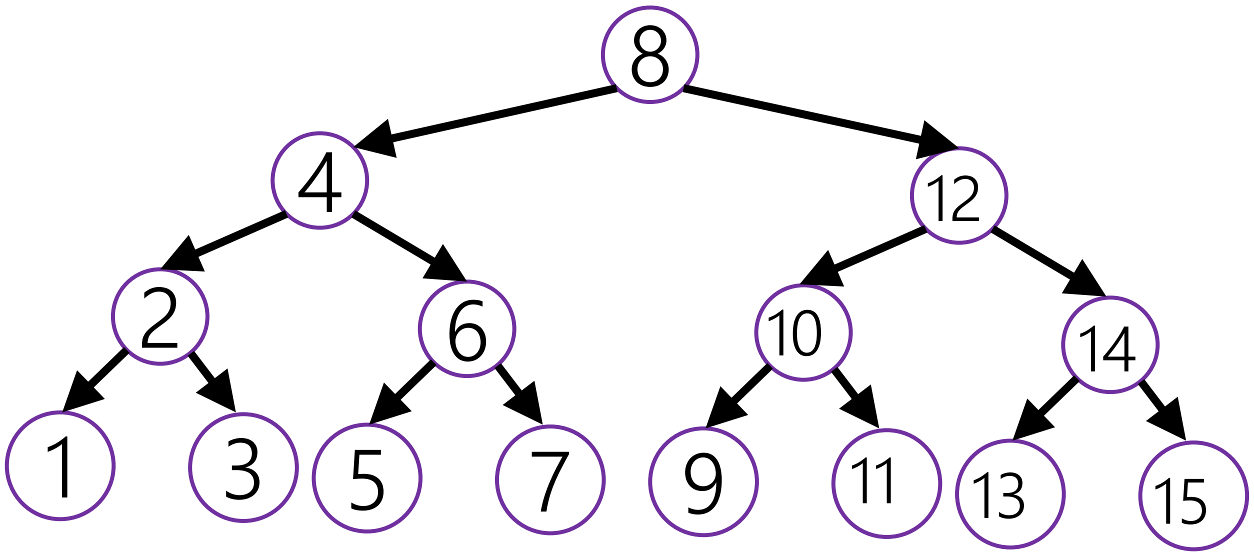
- For every node with key k:
  - The left subtree has only keys smaller than k.
  - The right subtree has only keys greater than k.



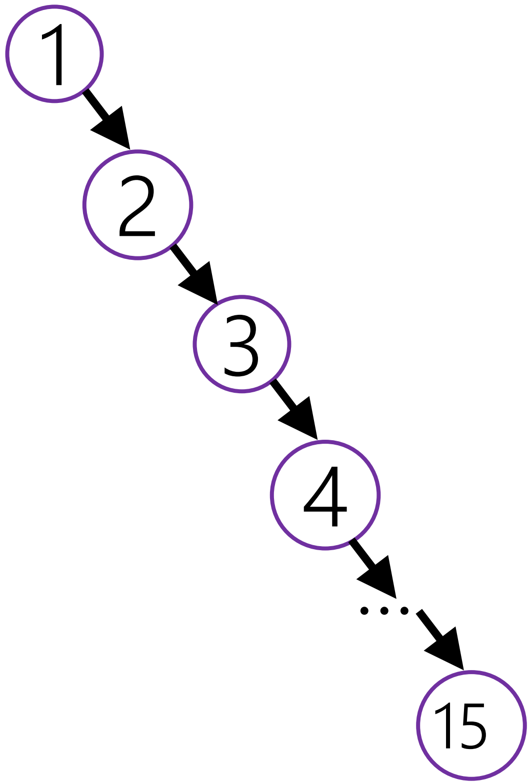
BST different states

Two different extreme states our BST could be in (there's in-between, but it's easiest to focus on the extremes as a starting point). Try containsKey(15) to see what the difference is.

Perfectly balanced – for every node, its descendants are split evenly between left and right subtrees.



Degenerate – for every node, all of its descendants are in the right subtree.



AVL Trees

AVL Trees must satisfy the following properties:

- binary trees: all nodes must have between 0 and 2 children
- binary search tree: for all nodes, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- balanced: for all nodes, there can be no more than a difference of 1 in the height of the left subtree from the right. Math.abs(height(left subtree) – height(right subtree)) ≤ 1

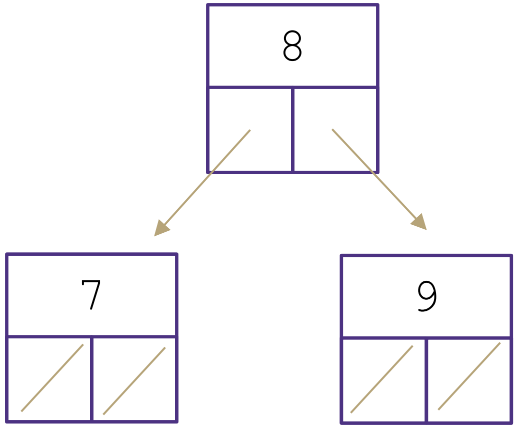
AVL stands for **A**delson-**V**elsky and **L**andis (the inventors of the data structure)

Measuring Balance

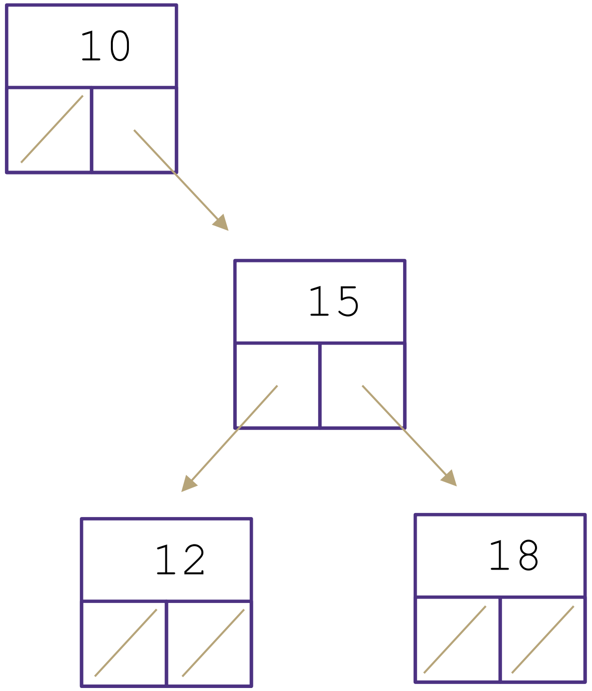
Measuring balance:

For each node, compare the heights of its two sub trees

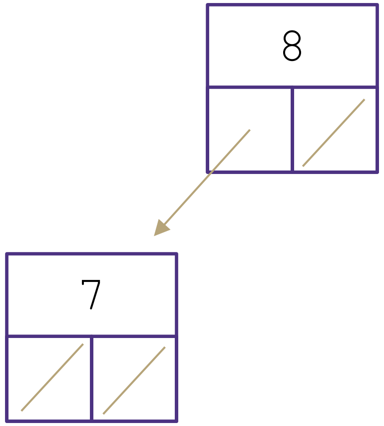
Balanced when the difference in height between sub trees is no greater than 1



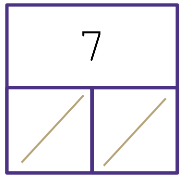
Balanced



Unbalanced



Balanced

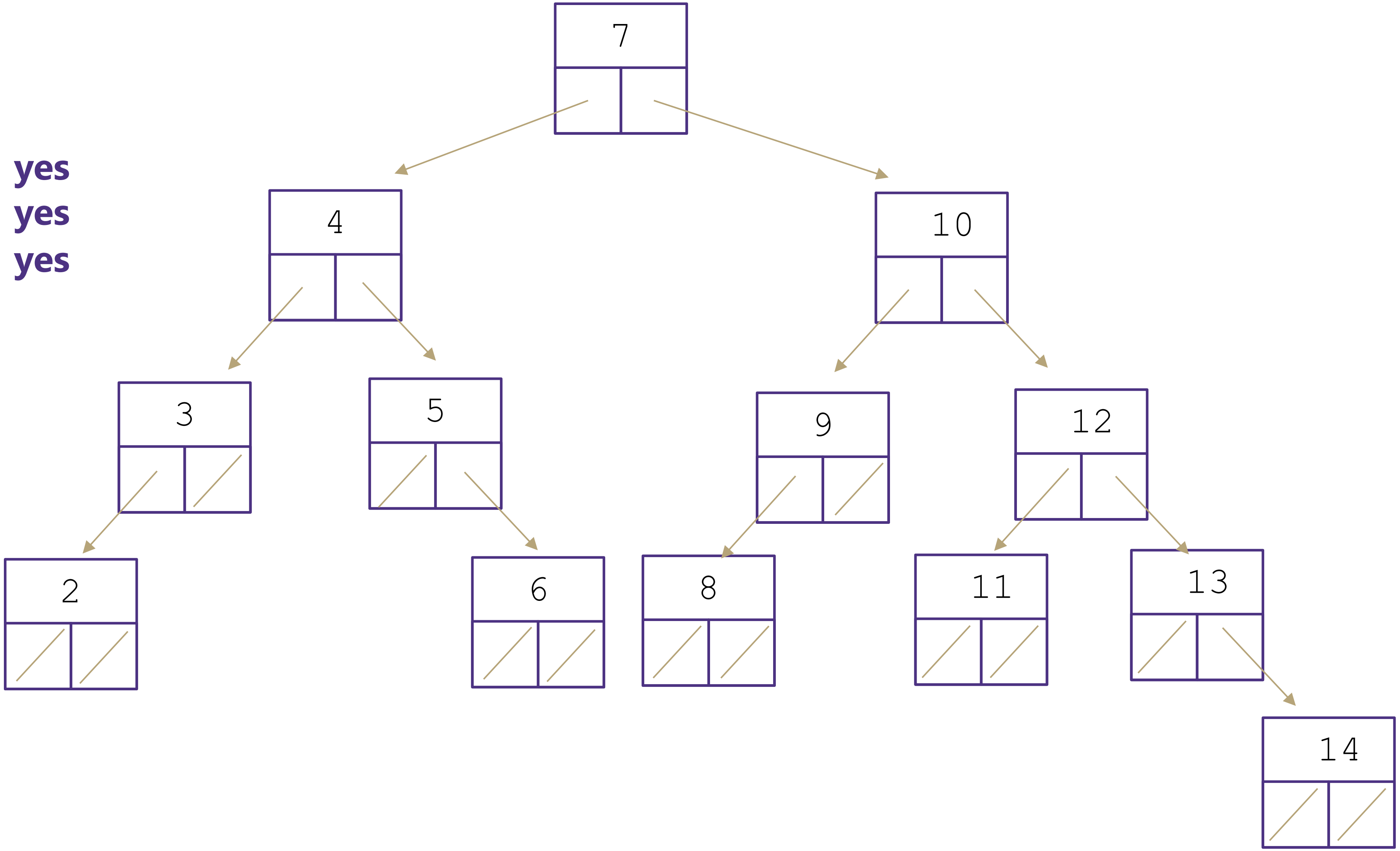


Balanced

Is this a valid AVL tree?

Is it…

- Binary **yes**
- BST **yes**
- Balanced? **yes**



Design Decisions

Before coding can begin engineers must carefully consider the design of their code will organize and manage data

Things to consider:

What functionality is needed?

- What operations need to be supported?
- Which operations should be prioritized?

What type of data will you have?

- What are the relationships within the data?
- How much data will you have?
- Will your data set grow?
- Will your data set shrink?

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?

Practice: Music Storage

You have been asked to create a new system for organizing songs in a music service. For each song you need to store the artist and how many plays that song has.

What functionality is needed?

- What operations need to be supported?
- Which operations should be prioritized?

Update number of plays for a song
Add a new song to an artist's collection
Add a new artist and their songs to the service
Find an artist's most popular song
Find service's most popular artist
more…

What type of data will you have?

- What are the relationships within the data?
- How much data will you have?
- Will your data set grow?
- Will your data set shrink?

Artists need to be associated with their songs, songs need t be associated with their play counts
Play counts will get updated a lot
New songs will get added regularly

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?

Some artists and songs will need to be accessed a lot more than others
Artist and song names can be very similar

Practice: Music Storage

How should we store songs and their play counts?

Hash Table – song titles as keys, play count as values, quick access for updates

Array List – song titles as keys, play counts as values, maintain order of addition to system

How should we store artists with their associated songs?

Hash Table – artist as key,

- Hash Table of their (songs, play counts) as values
- AVL Tree of their songs as values

AVL Tree – artists as key, hash tables of songs and counts as values

Priority Queue ADT

Min Priority Queue ADT
state Set of comparable values - Ordered based on "priority" **behavior** add(value) – add a new element to the collection removeMin() – returns the element with the smallest priority, removes it from the collection peekMin() – find, but do not remove the element with the smallest priority

Imagine you're managing a queue of food orders at a restaurant, which normally takes food orders first-come-first-served. But suddenly, Ana Marie Cauce walks into the restaurant. You know that you should server her as soon as possible (to either suck up or kick her out of the restaurant), and realize other celebrities (CSE 373 staff) could also arrive soon. Your new food management system should rank customers and let us know which food order we should work on next (the most prioritized thing).

Other uses:

- Well-designed printers
- Huffman Coding (see in CSE 143 last hw)
- Sorting algorithms
- Graph algorithms

Binary Heap invariants summary

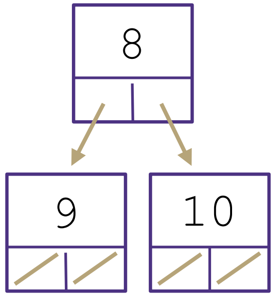
This is a big idea!
(heap invariants!)

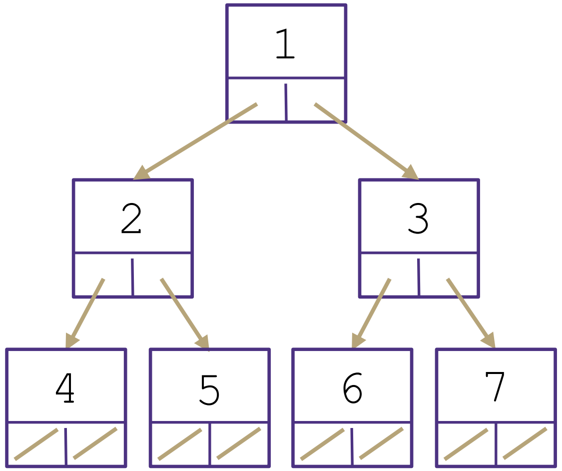
One flavor of heap is a **binary** heap.

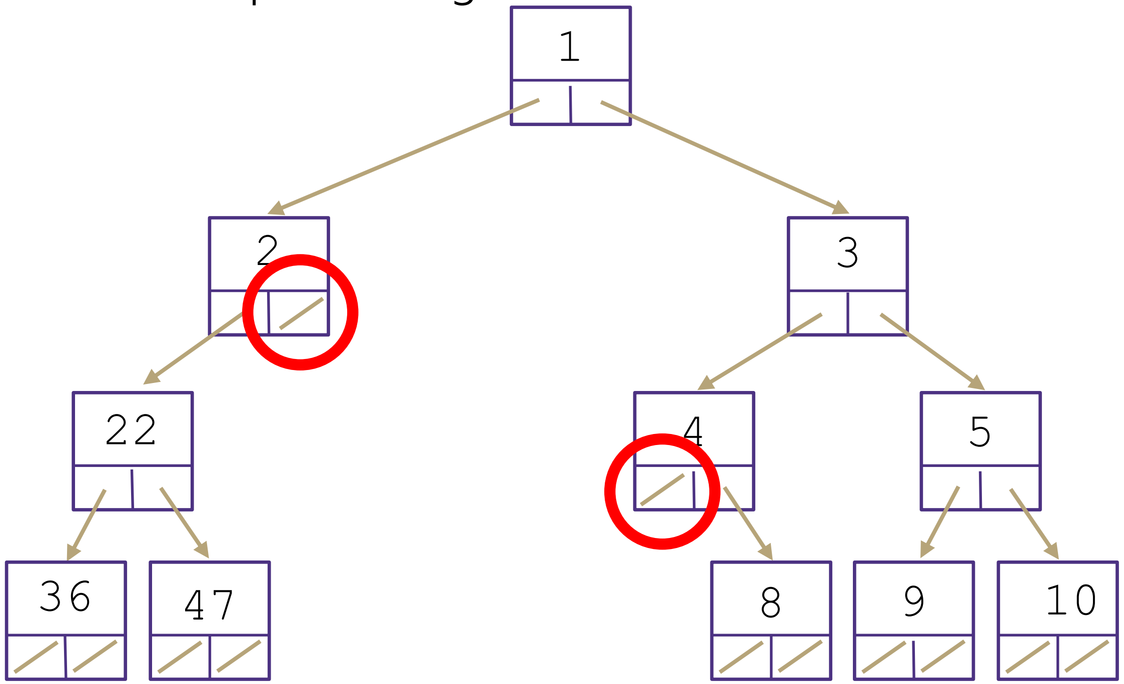
1. Binary Tree: every node has at most 2 children

2. Heap invariant: every node is smaller than (or equal to) its children

3. Heap structure invariant: Each level is "complete" meaning it has no "gaps"
   - Heaps are filled up left to right







Announcements

P2 due today!

Midterm out this Friday – due 1 week later

NO LATE ASSIGNMENTS ACCEPTED

- Group assignment
- Open note/ open internet, closed course staff
- intended to take 1 person 1 hour
- Topics:
  - ADTs
  - Code Modeling
  - Big O, Big Theta, Big Omega
  - Case Analysis
  - Recurrences
  - Master Theorem & Tree Method
  - Hashing
  - BSTs & AVls
  - Heaps
  - Design Decisions

Sorry about OH – we doing out best!

What's NOT on the midterm:

- AVL Rotations
- Big O Proofs (C and N0 style)
- Summation Identities (Limited algebra)

Come to the Midterm Review!

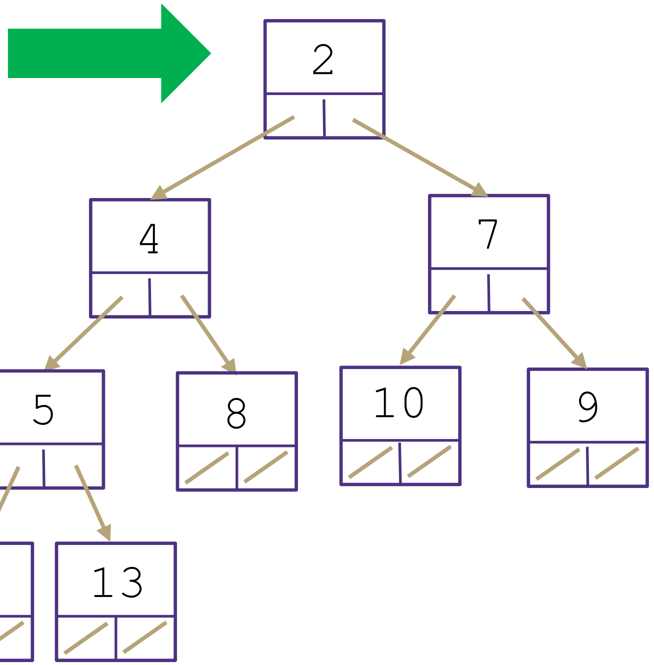
- Thursday (tomorrow) evening 5:30-7:30 pm PST

Mid Quarter Surveys

- Lecture
- Section
- 90% response rate on all- 1 point EC for everyone!

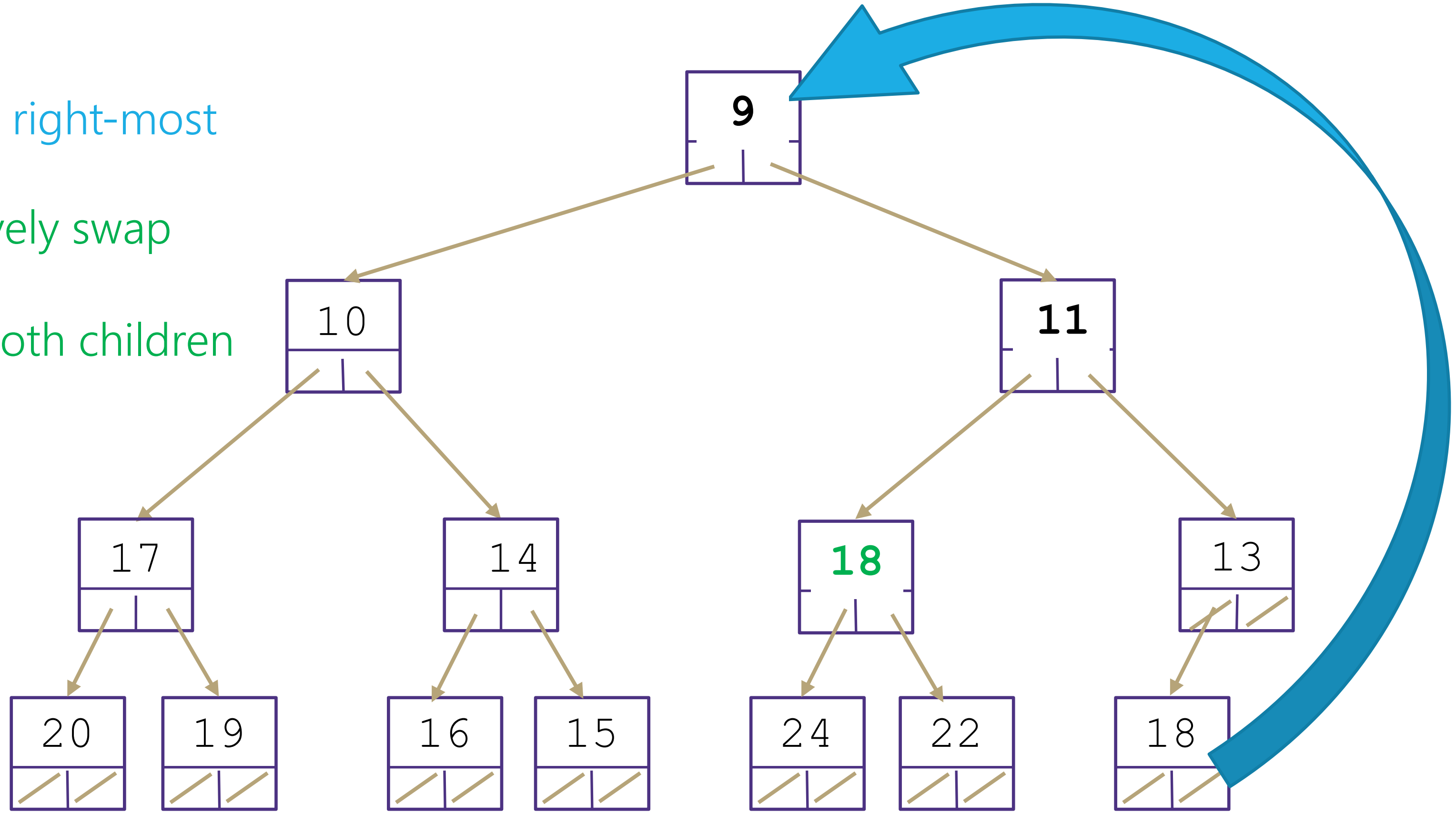
Implementing peekMin()

Runtime: $\Theta(1)$

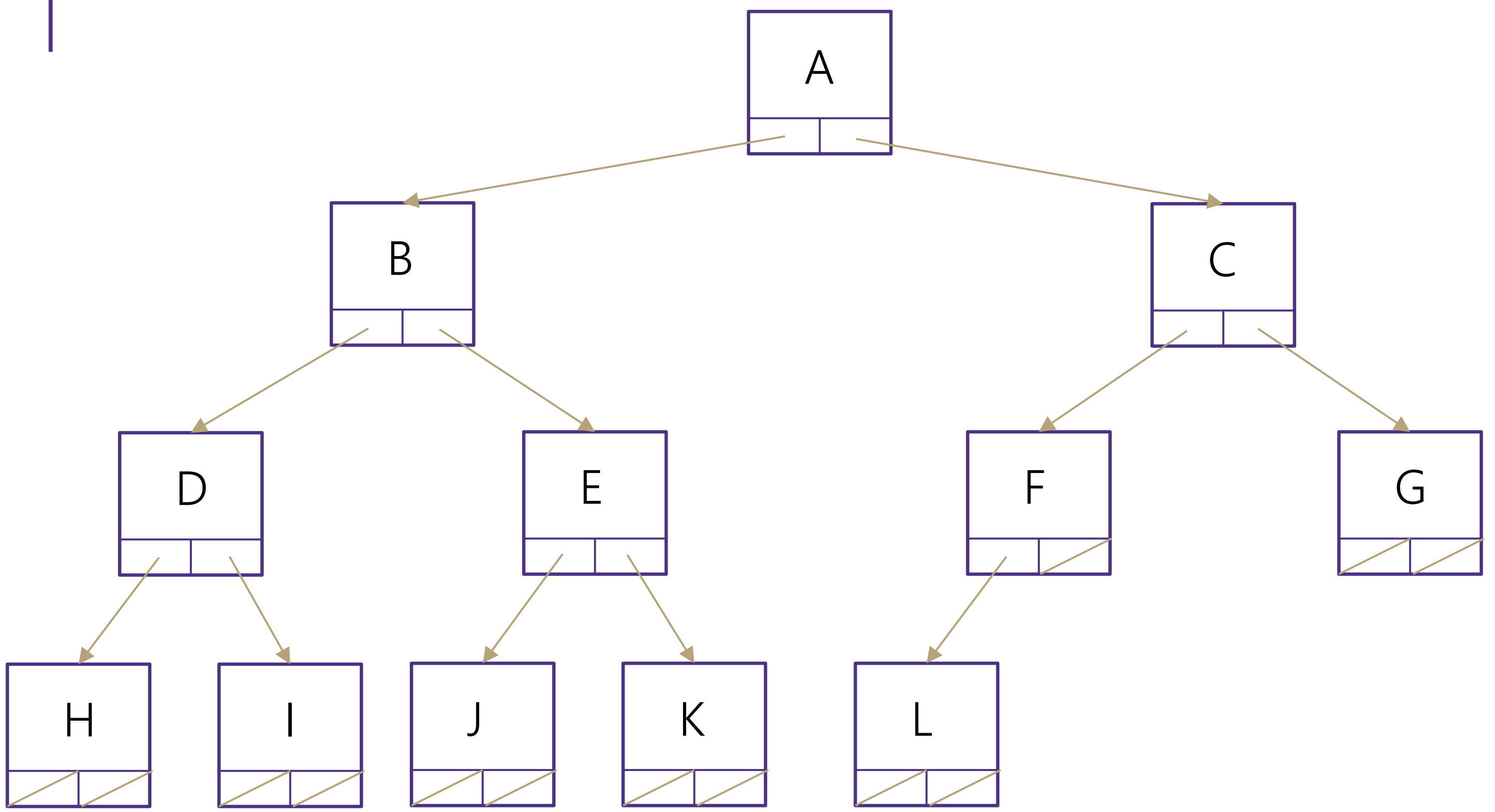


Practice: removeMin()

1.) Remove min node
2.) replace with bottom level right-most node
3.) percolateDown - Recursively swap parent with **smallest** child until parent is smaller than both children (or we're at a leaf).



Implement Heaps with an array



Fill array in level-order from left to right

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	B	C	D	E	F	G	H	I	J	K	L		

How do we find the minimum node?

$$peekMin() = arr[0]$$

How do we find the last node?

$$lastNode() = arr[size\ - 1]$$

How do we find the next open space?

$$openSpace() = arr[size]$$

How do we find a node's left child?

$$leftChild(i) = 2i + 1$$

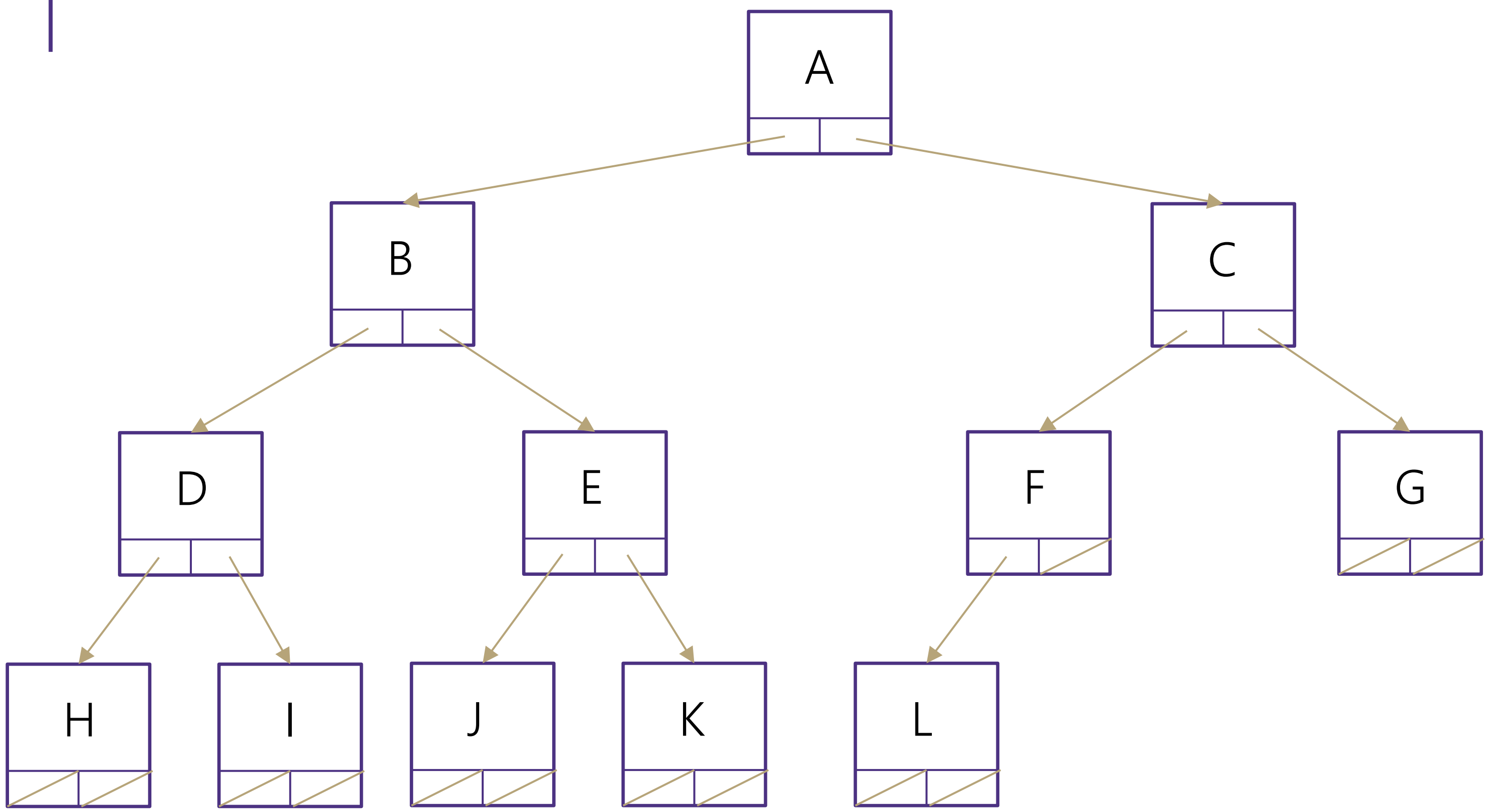
How do we find a node's right child?

$$rightChild(i) = 2i + 2$$

How do we find a node's parent?

$$parent(i) = \frac{(i-1)}{2}$$

Implement Heaps with an array



Fill array in level-order from left to right

0	1	2	3	4	5	6	7	8	9	10	11	12	13
/	A	B	C	D	E	F	G	H	I	J	K	L	

How do we find the minimum node?

$$peekMin() = arr[1]$$

How do we find the last node?

$$lastNode() = arr[size]$$

How do we find the next open space?

$$openSpace() = arr[size + 1]$$

How do we find a node's left child?

$$leftChild(i) = 2i$$

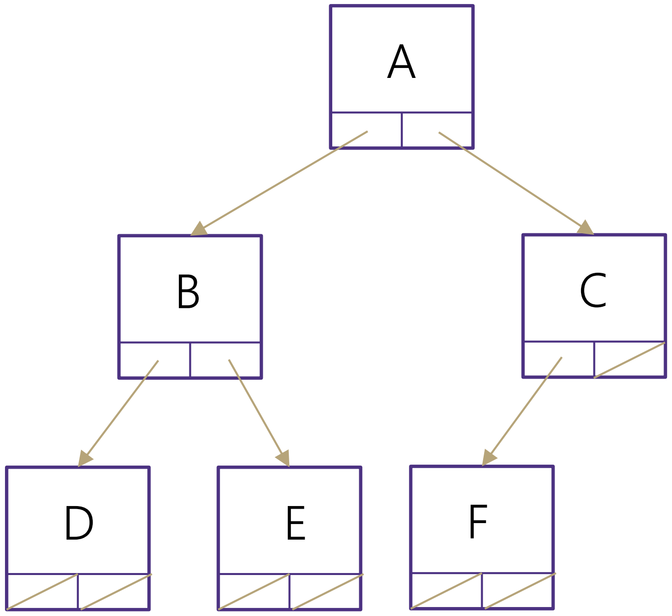
How do we find a node's right child?

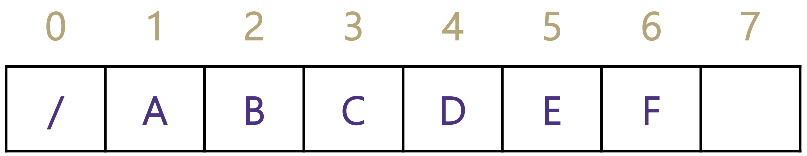
$$rightChild(i) = 2i + 1$$

How do we find a node's parent?

$$parent(i) = \frac{i}{2}$$

Array-Implemented MinHeap Runtimes





Operation	Case	Runtime
`removeMin()`	best	Θ(1)
	worst	Θ(log n)
	in practice	Θ(log n)
`add(key)`	best	Θ(1)
	worst	Θ(log n)
	in practice	Θ(1)
`peekMin()`	all cases	Θ(1)

- With array implementation, heaps match runtime of finding min in AVL trees
- But better in many ways!
  - Constant factors: array accesses give contiguous memory/spatial locality, tree constant factor shorter due to stricter height invariant
  - In practice, add doesn't require many swaps
  - WAY simpler to implement!