Lecture 14: buildHeap and Midterm Review

CSE 373 Data Structures and Algorithms

Administrivia

Exercise 3 3d (give an insertion order to cause failed probes) has been changed.

We now require only 4 failed probes not 5.

There is a way to get 5, but the hint wasn't leading you to an answer that gives 5 failures.

Sorry 😔

Section handouts and solutions for tomorrow are published

More Priority Queue Operations

Warm up

We said the height of a heap is always $\Theta(\log n)$.

Let's argue why:

How many nodes are there in a **complete** binary tree of height h?

Above the last level: $\sum_{i=0}^{h-1} 2^i$

At the last level: between 1 and 2^h

Total? Min: $1 + \sum_{i=0}^{h-1} 2^i = 2^h$

Max: $2^h + \sum_{i=0}^{h-1} 2^i = 2^h + 2^h - 1 = 2^{h+1} - 1$

If you have n nodes in a heap, what can you say about the height?

$$2^h \leq n < 2^{h+1}$$
$$h \leq \log_2 n < h + 1$$
$$h \in \Theta(\log n)$$

We can actually write something more specific: $h = \lfloor \log n \rfloor$

More Operations

Min Priority Queue ADT

state

Set of comparable values
- Ordered based on "priority"

behavior

insert(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

We'll use priority queues for lots of things later in the quarter.

Let's add them to our ADT now.

Some of these will be **asymptotically** faster for a heap than an AVL tree!

BuildHeap(elements $e_1, \dots, e_n$)
Given n elements, create a heap containing exactly those n elements.

Even More Operations

BuildHeap(elements $e_1, \ldots, e_n$ **) –** Given n elements, create a heap containing exactly those n elements.

Try 1: Just call insert n times.

Worst case running time?

n calls, each worst case $\Theta(\log n)$. So it's $\Theta(n \log n)$ right?

That proof isn't valid. There's no guarantee that we're getting the worst case every time!

Proof is right if we just want an $O()$ bound

- But it's not clear if it's tight.

BuildHeap Running Time

pollEV.com/cse373su19

What order produces the worst case?

Let's try again for a Theta bound.

The problem last time was making sure we always hit the worst case.

Suppose our priorities are $1,2,3,\ldots,n$ what is the worst-case order?

Insert the elements in decreasing order to hit the worst case each time!

- Every node will have to percolate all the way up to the root.

So we really have $n\ \Theta(\log n)$ operations. Done?

There's still a bug with this proof!

BuildHeap Running Time (again)

Let's try once more.

Saying the worst case was decreasing order was a good start.

What are the actual running times?

It's $\Theta(h)$, where h is the **current** height.

- The tree isn't height $\log n$ at the beginning.

But most nodes are inserted in the last two levels of the tree.

- For most nodes, h is $\Theta(\log n)$.

The number of operations is at least

$$\frac{n}{2} \cdot \Omega(\log n) = \Omega(n \log n).$$

Where Were We?

We were trying to design an algorithm for:

BuildHeap(elements $e_1, \ldots, e_n$ **) –** Given n elements, create a heap containing exactly those n elements.

Just inserting leads to a $\Theta(n \log n)$ algorithm in the worst case.

Can we do better?

Can We Do Better?

What’s causing the n `insert` strategy to take so long?

Most nodes are near the bottom, and they might need to percolate all the way up.

What if instead we dumped everything in the array and then

tried to percolate things down to fix the invariant?

Seems like it might be faster

- The bottom two levels of the tree have $\Omega(n)$ nodes, the top two have 3 nodes.
- Maybe we can make “most nodes” go a constant distance.

Is It Really Faster?

Assume the tree is **perfect**

- the proof for complete trees just gives a different constant factor.

percolateDown() doesn't take $\log n$ steps each time!

Half the nodes of the tree are leaves

- Leaves run percolate down in constant time

1/4 of the nodes have at most 1 level to travel

1/8 the nodes have at most 2 levels to travel

etc…

work(n) $\approx \frac{n}{2} \cdot\ 1\ + \frac{n}{4} \cdot\ 2\ + \frac{n}{8} \cdot\ 3\ + \cdots + 1 \cdot (\log n)$

Closed form Floyd's buildHeap

$n/2 \cdot 1 + n/4 \cdot 2 + n/8 \cdot 3 + \cdots + 1 \cdot (\log n)$

factor out n

work(n) $\approx n\left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{\log n}{n}\right)$ find a pattern -> powers of 2 work(n) $\approx n\left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{\log n}{2^{\log n}}\right)$ Summation!

$$work(n) \approx n \sum_{i=1}^{?} \frac{i}{2^i}$$? = upper limit should give last term

We don't have a summation for this! Let's make it look more like a summation we do know.

Infinite geometric series

$$work(n) \leq n \sum_{i=1}^{\log n} \frac{\left(\frac{3}{2}\right)^i}{2^i}$$

$$if -1 < x < 1\ then \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} = x$$

$$work(n) \approx n \sum_{i=1}^{\log n} \frac{i}{2^i} \leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = n * 4$$

Floyd's buildHeap runs in O(n) time!

Floyd’s BuildHeap

Ok, it’s really faster.
But can we make it **work**?

It’s not clear what order to call the `percolateDown`’s in.

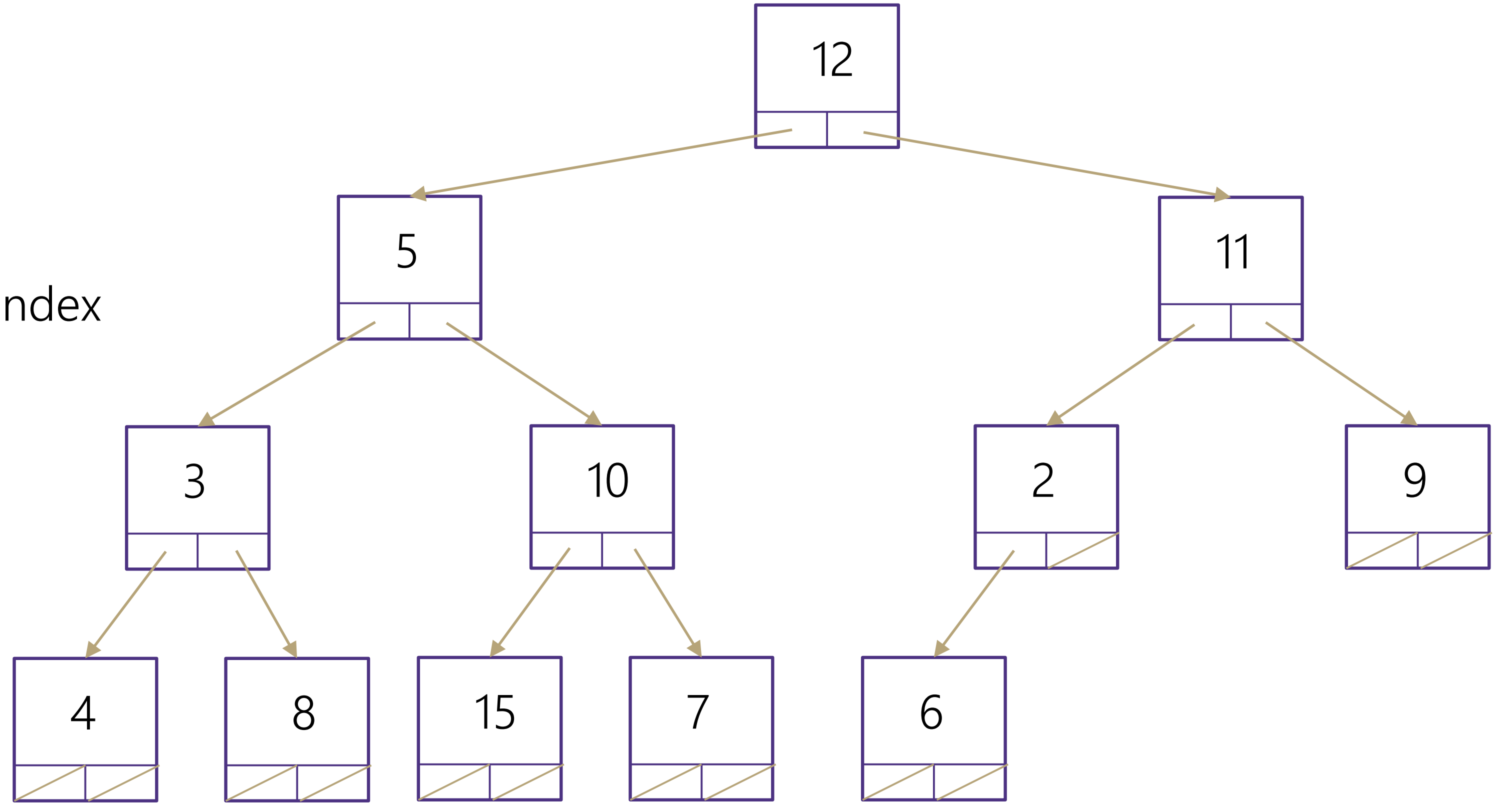
Should we start at the top or bottom? Will one `percolateDown` on each element be enough?

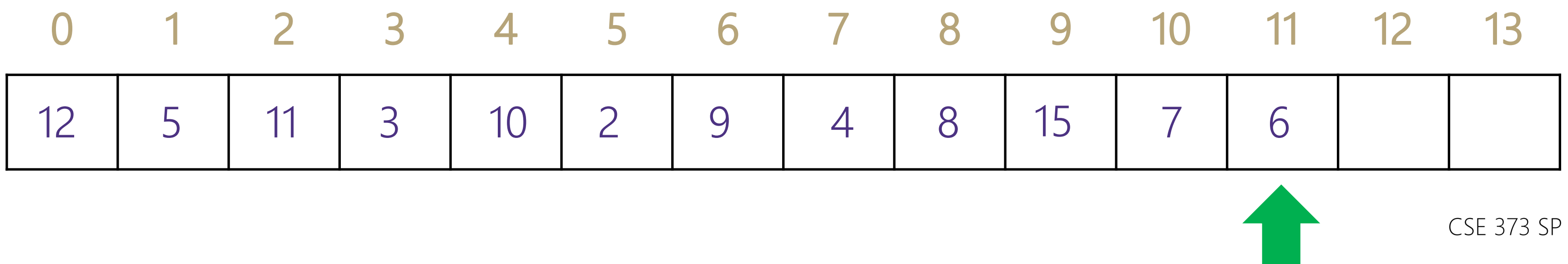
Yes! If we start at the bottom and work up.

Floyd’s buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to array
2. percolateDown(parent) starting at last index



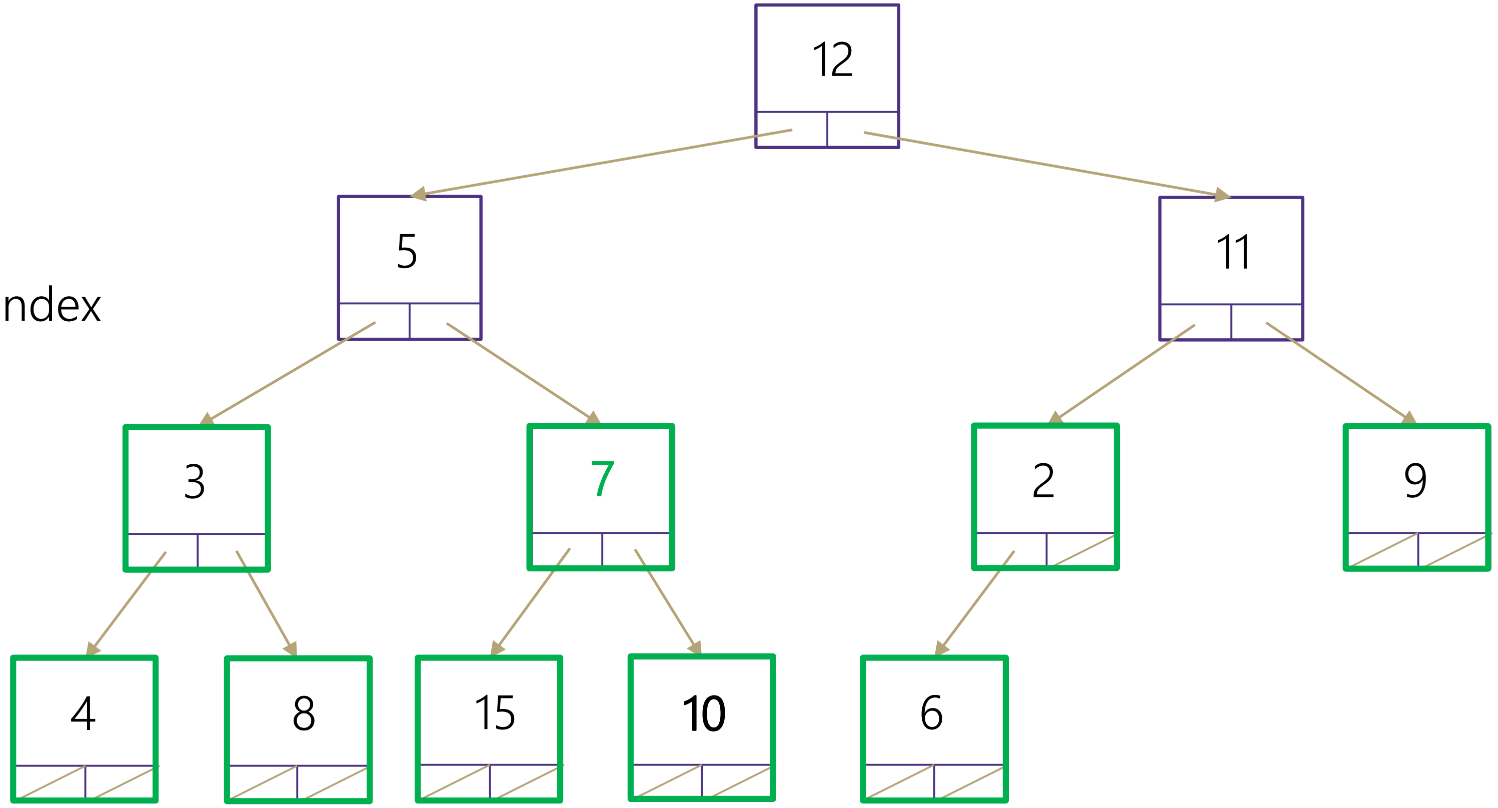


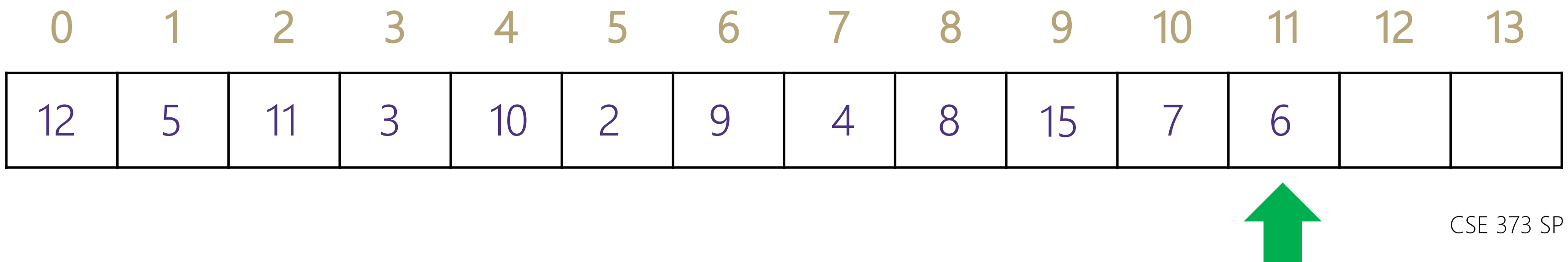
0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	15	7	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to array
2. percolateDown(parent) starting at last index
   1. percolateDown level 3
   2. percolateDown level 2



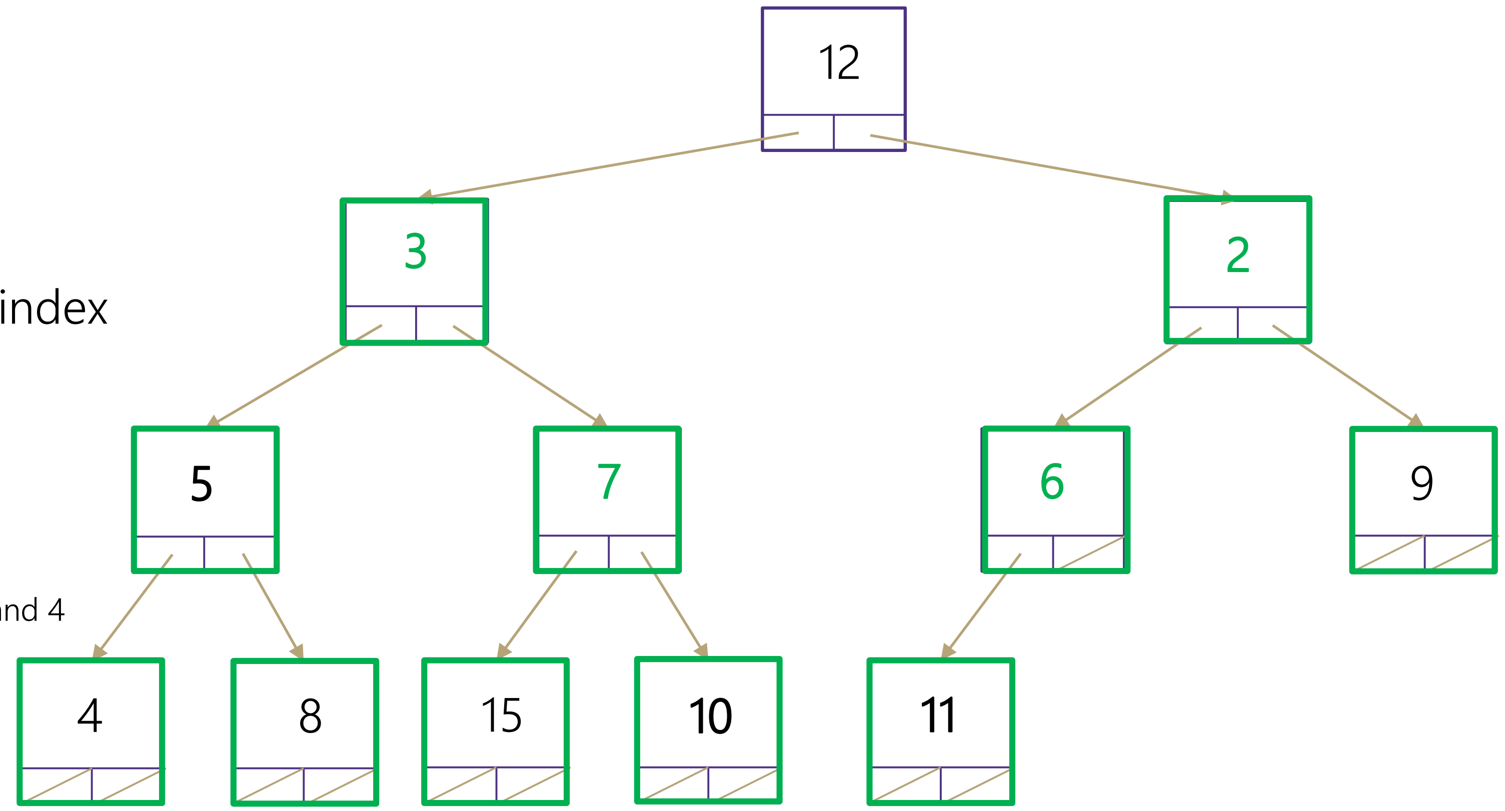


Floyd’s buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to array
2. percolateDown(parent) starting at last index
    1. percolateDown level 3
    2. percolateDown level 2
    3. percolateDown level 1

keep percolating down
like normal here and swap 5 and 4

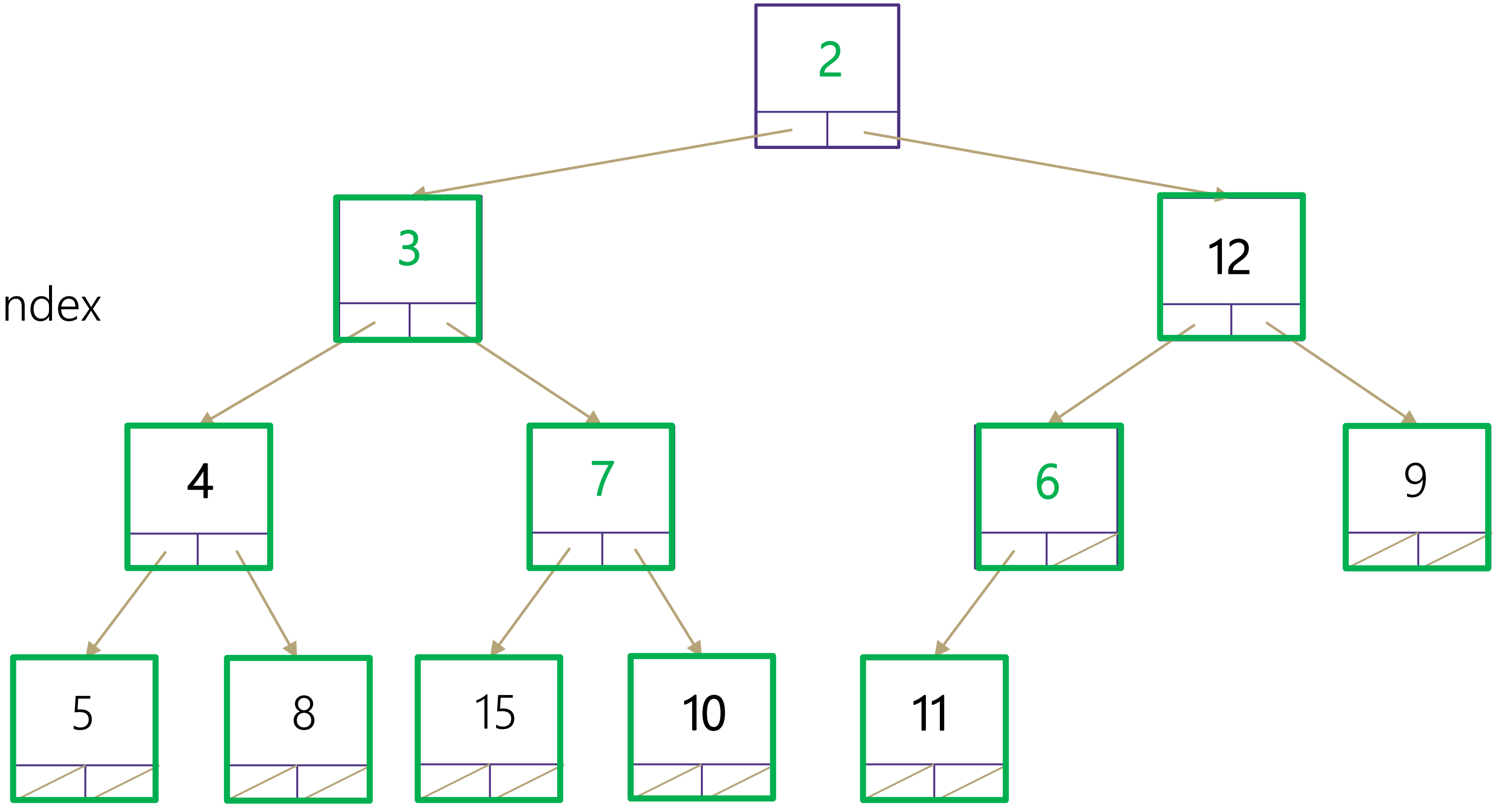


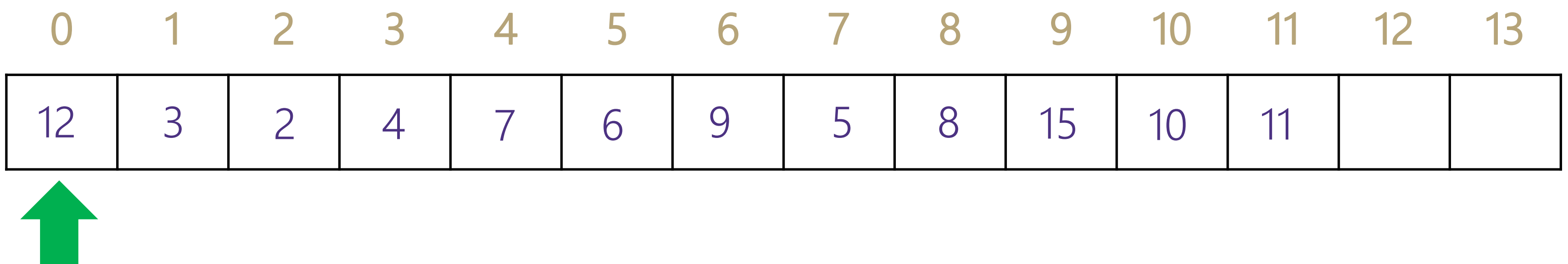
0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	7	2	9	4	8	15	10	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to array
2. percolateDown(parent) starting at last index
    1. percolateDown level 3
    2. percolateDown level 2
    3. percolateDown level 1
    4. percolateDown level 0

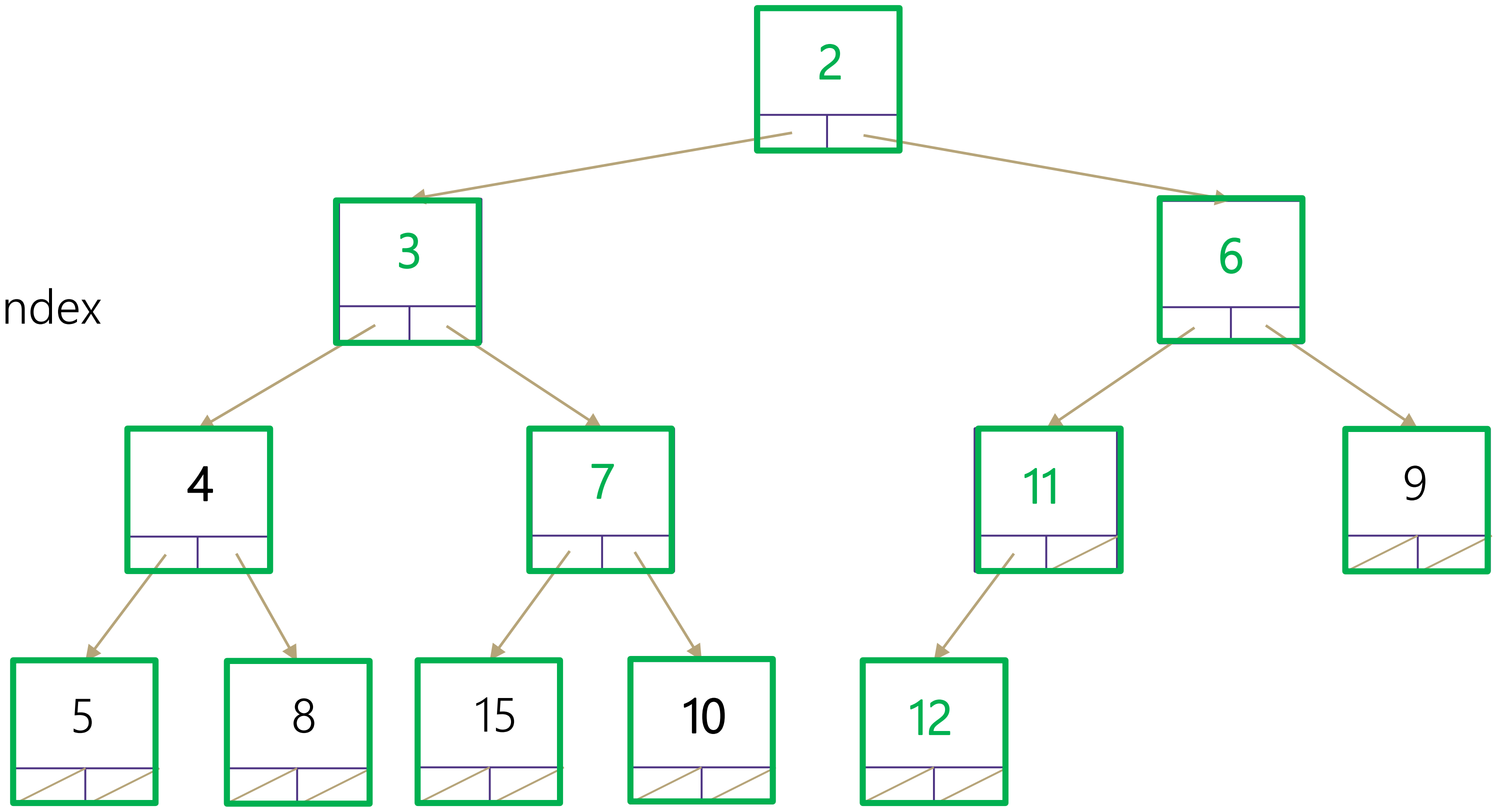


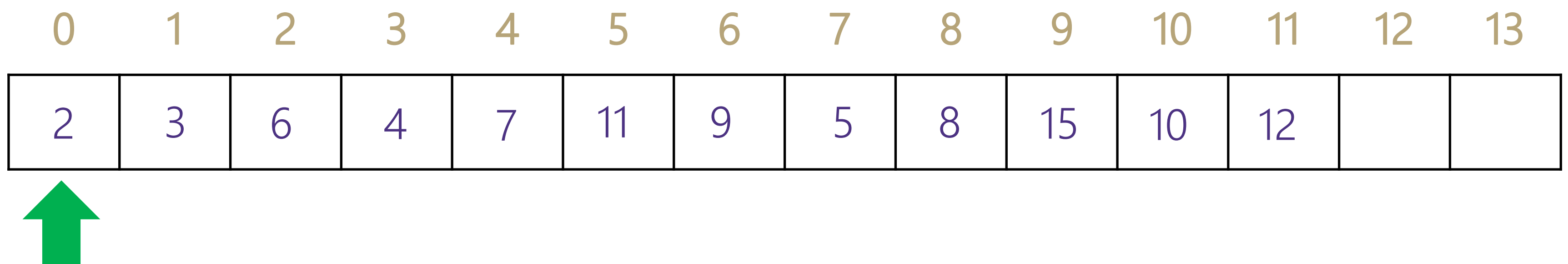


Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to array
2. percolateDown(parent) starting at last index
   1. percolateDown level 3
   2. percolateDown level 2
   3. percolateDown level 1
   4. percolateDown level 0





0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	3	6	4	7	11	9	5	8	15	10	12		

Even More Operations

These operations will be useful in a few weeks…

IncreaseKey(element,priority) Given an element of the heap and a new, larger priority, update that object's priority.

DecreaseKey(element,priority) Given an element of the heap and a new, smaller priority, update that object's priority.

Delete(element) Given an element of the heap, remove that element.

Should just be going to the right spot and percolating…

Going to the right spot is the tricky part.

In the programming projects, you'll use a dictionary to find an element quickly.

Midterm Review

Midterm Logistics

60 minutes

8.5 x 11 inch note page, front and back

Math identities sheet provided (posted on website)

We will be scanning your exams to grade…

- Try not to cram answers into margins or corners
- If you want us to look on the back, tell us *in each problem part* where you want us to look at the back.

We'll mark some rows to not sit in during the exam

- So TAs can answer questions without climbing over anyone.

There aren't enough seats for an empty spot between everyone…

- But still spread out.

Asymptotic Analysis

Asymptotic Analysis

asymptotic analysis – the process of mathematically representing the runtime of an algorithm in relation to the size of the input

- Don't care about constant factors or small n.

Two step process

1. Model – come up with a function to describe the running time
2. Analyze – compare runtime/input relationship across multiple algorithms/data structures

    For which inputs will one perform better than the other?

Code Modeling

```
public int mystery(int n) {
   int result = 0;   +1
   for (int i = 0; i < n/2; i++) {     } n/2
      result++;  +1
   }
   for (int i = 0; i < n/2; i+=2) {    } n/4
      result++;  +1
   }
   result * 10;  +
                 1
   return result;  +1
}
```

$$\boxed{f(n) = 3 + \frac{3}{4}n}$$

Code Modeling Example

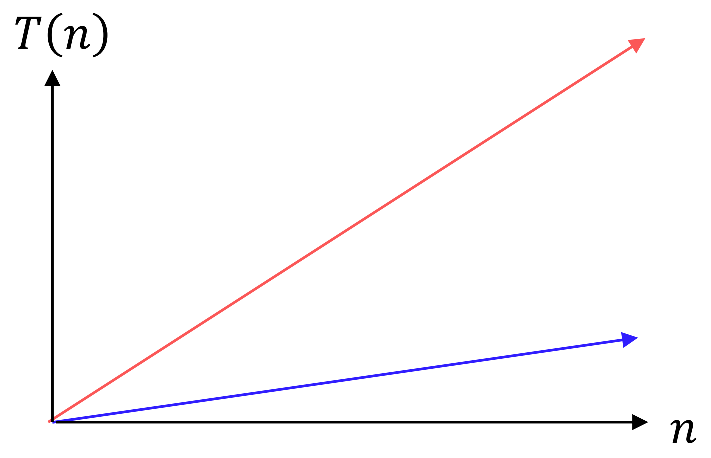
```
    public String mystery (int n) {
+1     ChainedHashDictionary<Integer, Character> alphabet =
                                  new ChainedHashDictionary<Integer, Character>();
       for (int i = 0; i < 26; i++) {
          char c = ‘a’ + (char)i;                    +26
          alphabet.put(i, c);
       }
+1     DoubleLinkedList<Character> result = new DoubleLinkedList<Character>();
       for (int i = 0; i < n; i += 2) {
          char c = alphabet.get(i); +26                  n/2
          result.add(c); +1
       }
+1     String final = “”;
       for (int i = 0; i < result.size(); i++) {
          final += result.remove(); +1                       n/2
       }
+1     return final;
    }
```

$$\boxed{f(n) = 4 + 26 + 27\left(\frac{n}{2}\right) + \frac{n}{2}}$$

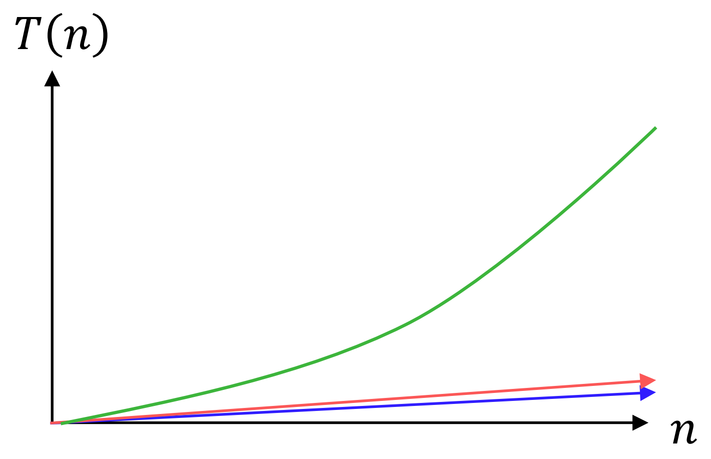
Function growth

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

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

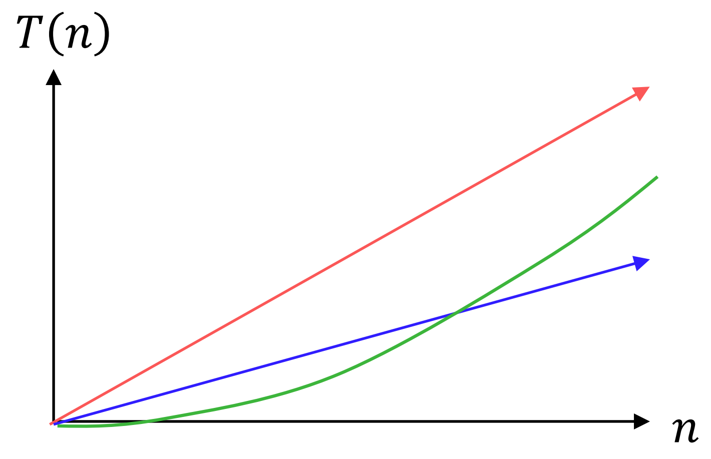


The growth rate for f(n) and g(n) looks very different for small numbers of input



...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)

O, Ω, Θ Definitions

O(f(n)) is the "family" or "set" of all functions that <u>are dominated by</u> f(n)

- f(n) ∈ O(g(n)) when f(n) <= g(n)
- The upper bound of an algorithm's function

Ω(f(n)) is the family of all functions that <u>dominate</u> f(n)

- f(n) ∈ Ω(g(n)) when f(n) >= g(n)
- The lower bound of an algorithm's function

Θ(f(n)) is the family of functions that are equivalent to f(n)

- We say f(n) ∈ Θ(g(n)) when both
- f(n) ∈ O(g(n)) and f(n) ∈ Ω (g(n)) are true
- A direct fit of an algorithm's function

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 $\Omega(g(n))$.

Proving Domination

Big-O

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

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

f(n) = 5(n + 2)

g(n) = 2n^2

Find a c and n$_0$ that show that f(n) ∈ O(g(n)).

$f(n) = 5(n + 2) = 5n + 10$

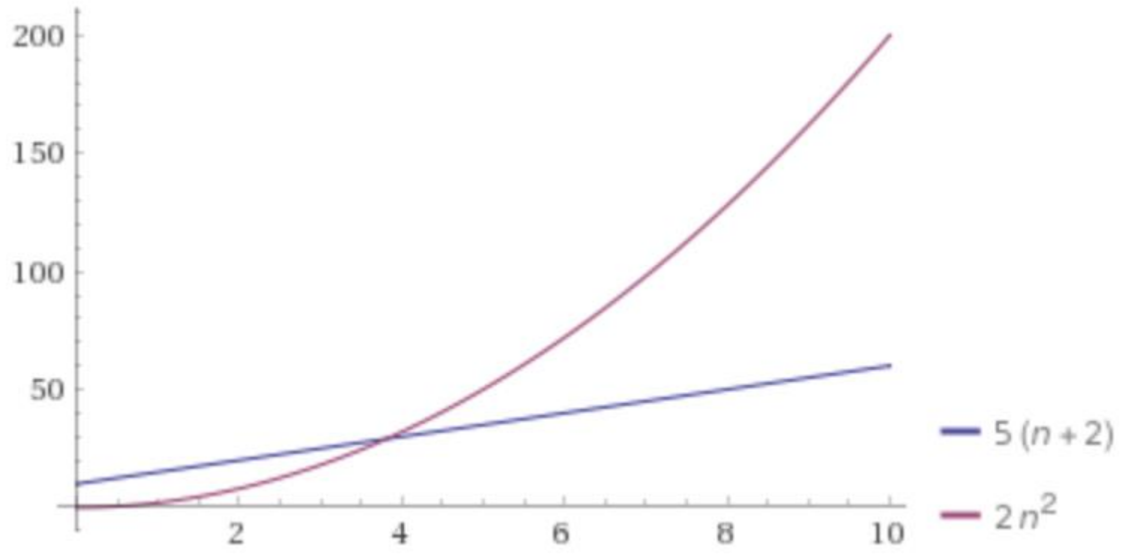
$5n \leq c \cdot 2\text{n}^2 \text{ for c} = 3 \text{ when n} \geq 1$

$10 \leq c \cdot 2\text{n}^2 \text{ for c} = 5 \text{ when n} \geq 1$

$5n + 10 \leq 3(2\text{n}^2) + 5(2\text{n}^2) \text{ when n} \geq 1$

$5n + 10 \leq 8(2\text{n}^2) \text{ when n} \geq 1$

$f(n) \leq c \cdot g(n) \; when \; c = 8 \; and \; n_0 = 1$



O, Ω, Θ Examples

For the following functions give the simplest tight O bound

$a(n) = 10\log n + 5$	O(logn)
$b(n) = 3^n - 4n$	$O(3^n)$
$c(n) = \frac{n}{2}$	O(n)

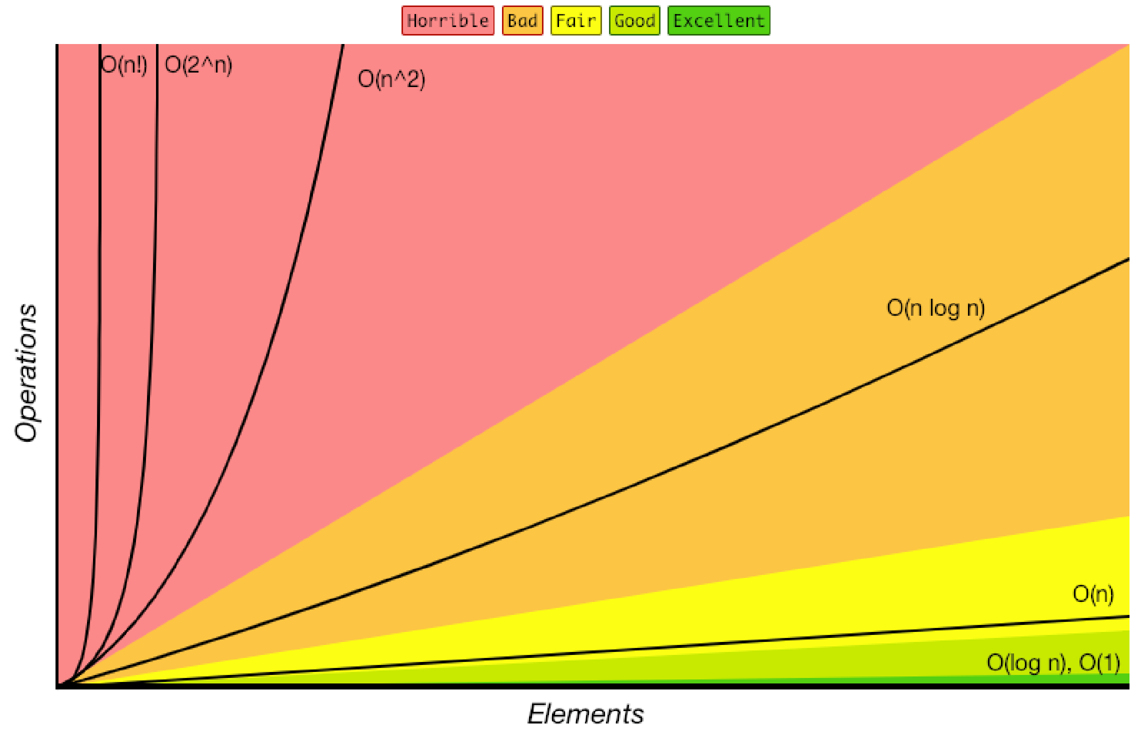
For the above functions indicate whether the following are true or false

a(n) ∈ O(b(n))	TRUE	b(n) ∈ O(a(n))	FALSE	c(n) ∈ O(b(n))	TRUE
a(n) ∈ O(c(n))	TRUE	b(n) ∈ O(c(n))	FALSE	c(n) ∈ O(a(n))	FALSE
a(n) ∈ Ω(b(n))	FALSE	b(n) ∈ Ω(a(n))	TRUE	c(n) ∈ Ω(b(n))	FALSE
a(n) ∈ Ω(c(n))	FALSE	b(n) ∈ Ω(c(n))	TRUE	c(n) ∈ Ω(a(n))	TRUE
a(n) ∈ Θ(b(n))	FALSE	b(n) ∈ Θ(a(n))	FALSE	c(n) ∈ Θ(b(n))	FALSE
a(n) ∈ Θ(c(n))	FALSE	b(n) ∈ Θ(c(n))	FALSE	c(n) ∈ Θ(a(n))	FALSE
a(n) ∈ Θ(a(n))	TRUE	b(n) ∈ Θ(b(n))	TRUE	c(n) ∈ Θ(c(n))	TRUE

Review: Complexity Classes

complexity class – a category of algorithm efficiency based on the algorithm's relationship to the input size N

Class	Big O	If you double N…	Example algorithm
constant	O(1)	unchanged	Add to front of linked list
logarithmic	$O(\log_2 n)$	Increases slightly	Binary search
linear	O(n)	doubles	Sequential search
"n log n"	$O(n\log_2 n)$	Slightly more than doubles	Merge sort
quadratic	$O(n^2)$	quadruples	Nested loops traversing a 2D array
cubic	$O(n^3)$	Multiplies by 8	Triple nested loop
polynomial	$O(n^c)$		
exponential	$O(c^n)$	Multiplies drastically	



http://bigocheatsheet.com/

Modeling Complex Loops

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!"); +c
    }
}
```

0 + 1 + 2 + 3 +…+ n-1

n

Summation
1 + 2 + 3 + 4 +… + n = $\sum_{i=1}^{n} i$

Definition: Summation

$\sum_{i=a}^{b} f(i)$ = f(a) + f(a + 1) + f(a + 2) + … + f(b-2) + f(b-1) + f(b)

T(n) = $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} c$

Function Modeling: Recursion

```
public int factorial(int n) {
    if (n == 0 || n == 1) {          +2
        return 1;
    } else {
        return n * factorial(n - 1);  +T(n-1)
}                  +2
```

$$T(n) = \begin{cases} 2 & \text{when } n = 0 \text{ or } 1 \\ 3 + T(n-1) & \text{otherwise} \end{cases}$$

Definition: Recurrence

Mathematical equivalent of an if/else statement

$$f(n) = \begin{cases} \mathit{runtime\ of\ base\ case\ when\ conditional} \\ \mathit{runtime\ of\ recursive\ case\ otherwise} \end{cases}$$

Tree Method Formulas

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

How much work is done by recursive levels (branch nodes)?

1. How many recursive calls are on the i-th level of the tree?
   - i = 0 is overall root level

numberNodesPerLevel(i) = 2^i

2. At each level i, how many inputs does a single node process?

inputsPerRecursiveCall(i) = $(n/ 2^i)$

3. What is the last level?
   - Based on the pattern of how we get down to base case

i= $\log_2 n$

$$Recursive\ work = \sum_{i=0}^{last\ level-1} branchNum(i) branchWork(i)$$

$$T(n > 1) = \sum_{i=0}^{\log_2 n-1} 2^i \left(\frac{n}{2^i}\right)$$

How much work is done by the base case level (leaf nodes)?

1. How much work is done by a single leaf node?

leafWork = 1

2. How many leaf nodes are there?

leafCount = $2^{\log_2 n}$ = n

$$NonRecursive\ work = leafWork \times leafCount = leafWork \times branchNum^{numLevels}$$

$$T(n \le 1) = 1\left(2^{\log_2 n}\right) = n$$

$$total\ work = recursive\ work + nonrecursive\ work =$$

$$\boxed{T(n) = \sum_{i=0}^{\log_2 n-1} 2^i \left(\frac{n}{2^i}\right) + n = n \log_2 n + n}$$

Tree Method Example

$$T(n) = \begin{cases} 3 \; when \; n = 1 \\ 3T\left(\frac{n}{3}\right) + n \; otherwise \end{cases}$$

Size of input at level i? $\frac{n}{3^i}$

Number of nodes at level i? 3^i

What is the last level? $\log_3(n)$

Total recursive work $\sum_{i=0}^{\log_3 n - 1} \frac{n}{3^i} 3^i = \mathrm{n} \log_3(n)$

How many nodes are on the bottom level? $3^{\log_3(n)} = n$

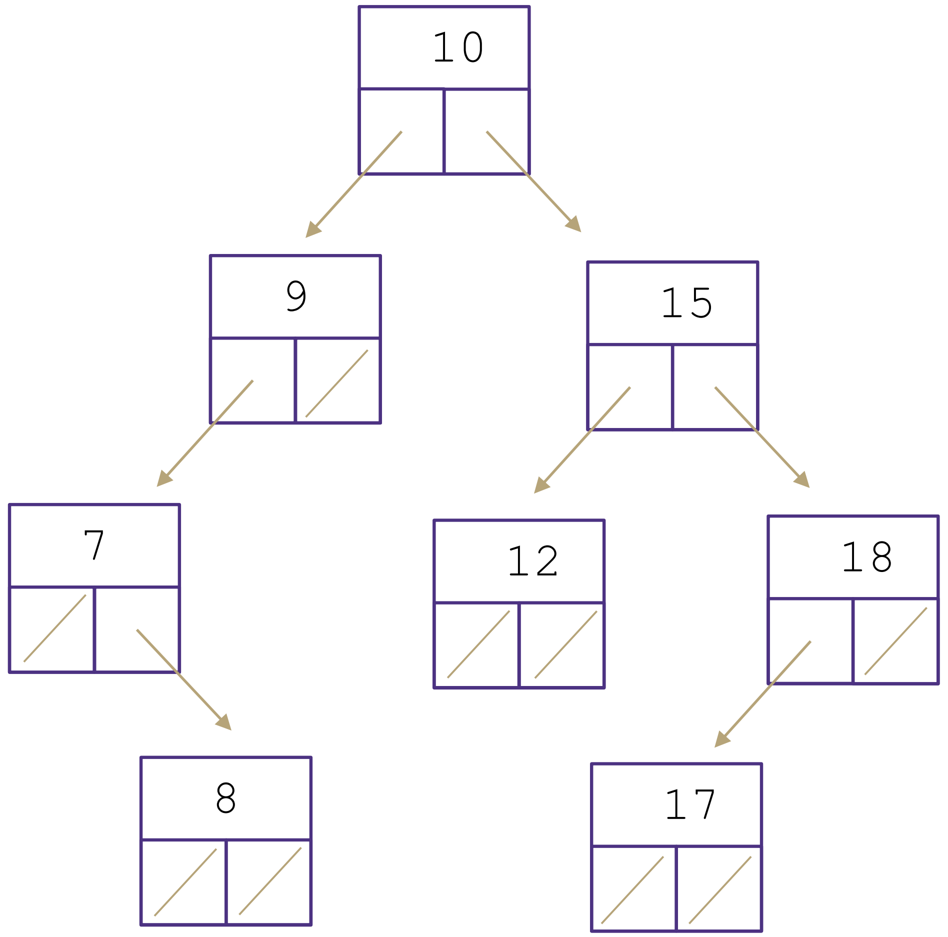
How much work done in base case? $3n$

$$\boxed{T(n) = n \log_3(n) + 3n}$$

BST & AVL Trees

Binary Search Trees

A binary search tree is a binary tree that contains comparable items such that for every node, all children to the left contain smaller data and all children to the right contain larger data.



Meet AVL Trees

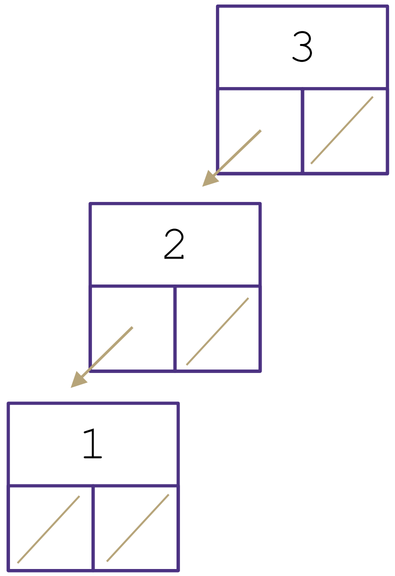
AVL Trees must satisfy the following properties:

- binary search tree: for all nodes, u, all keys in the left subtree must be smaller than u's key and all keys in the right subtree must be larger than u's key.
- AVL condition: for all nodes, the difference between the height of the left subtree and the right subtree is at most one. i.e. Math.abs(height(left subtree) – height(right subtree)) ≤ 1

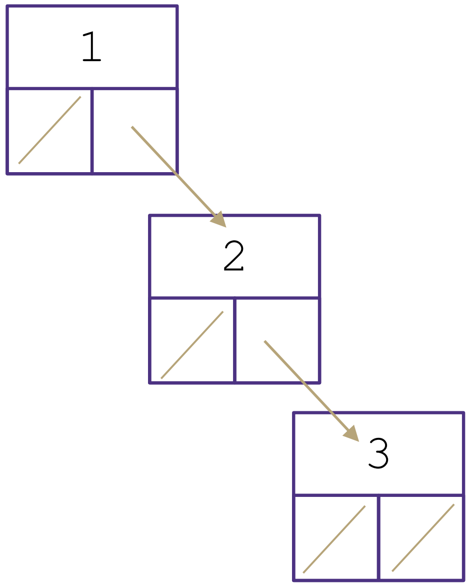
AVL stands for **A**delson-**V**elsky and **L**andis (the inventors of the data structure)

AVL Cases

Line Case
Solve with **1** rotation

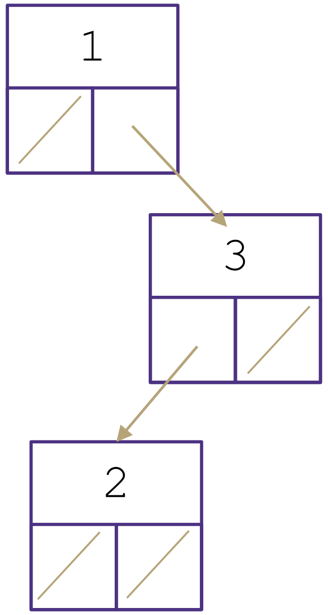


Rotate Right
Parent's left becomes child's right
Child's right becomes its parent

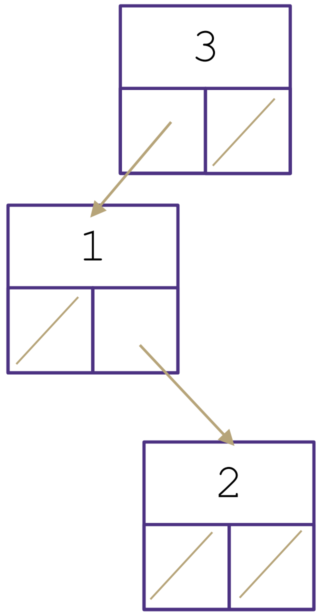


Rotate Left
Parent's right becomes child's left
Child's left becomes its parent

Kink Case
Solve with **2** rotations



Right Kink Resolution
Rotate subtree right
Rotate root tree left



Left Kink Resolution
Rotate subtree left
Rotate root tree right

Rebalancing Steps

1. Recurse up the tree until you find the **lowest** imbalanced node, u.

2. From u, take two steps toward where the insertion happened.
   - i.e. toward the tallest subtree.
   - Call the two nodes you visited v, w

3. Both steps same direction?
   - Single rotation with u, v, w

4. steps going different directions?
   - Double rotation!
   - Start by rotating, v, w and the child of w in a line with v and w. Rotate so u, v, w are in a line
   - Now rotate u, v, w the other way to create balance.

Remember to reconnect orphaned subtrees in BST order.

BST/AVL true/false

Which of the following is true for BSTs and for AVL trees?

	BSTs	AVL
All leaves are distance $\Omega(\log n)$ from the root	False	True
Traversal takes $\Theta(n)$ time	True	True
`containsKey` is $O(n)$ worst case	True	True (only technically – it's $\Theta(\log n)$)

Hashing

Implement First Hash Function

```
public V get(int key) {
   int newKey = getKey(key);
   this.ensureIndexNotNull(key);
   return this.data[key].value;
}

public void put(int key, int value) {
    this.array[getKey(key)] = value;
}
public void remove(int key) {
   int newKey = getKey(key);
   this.entureIndexNotNull(key);
   this.data[key] = null;
}
public int getKey(int value) {
   return value % this.data.length;
}
```

SimpleHashMap<Integer>

state

Data[]
size

behavior

put mod key by table size, put item at result
get mod key by table size, get item at result
containsKey mod key by table size, return data[result] == null remove mod key by table size, nullify element at result
size return count of items in dictionary

First Hash Function: % table size

indices	0	1	2	3	4	5	6	7	8	9
elements	“:(” ’	“biz”				“bar”			“bop”	

```
put(0, “foo”);  0 % 10 = 0
put(5, “bar”);  5 % 10 = 5
put(11, “biz”)  11 % 10 = 1
put(18, “bop”); 18 % 10 = 8
put(20, “:(”);  20 % 10 = 0
```

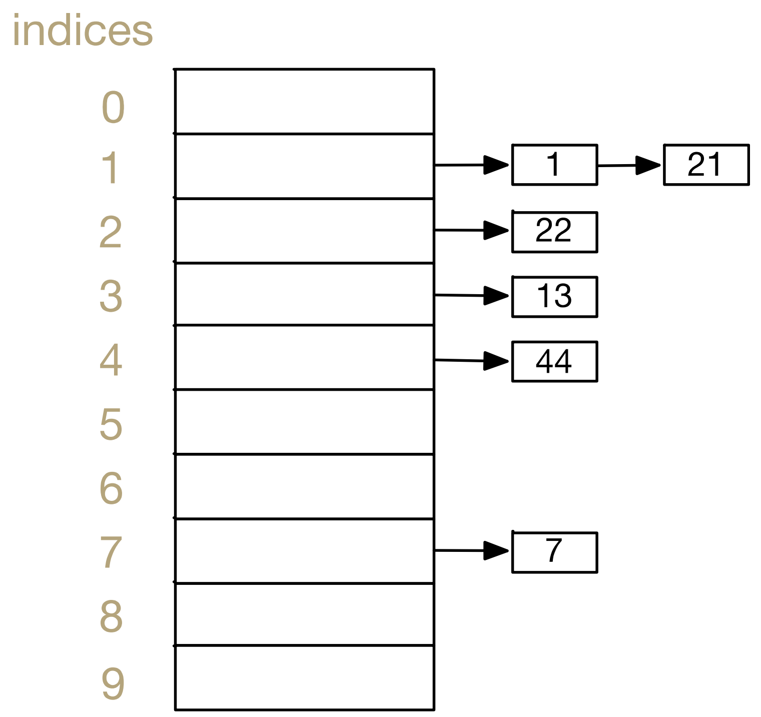
$\longrightarrow$ Collision!

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)
	In-practice	O(1 + λ)
	worst	O(n)
get(key)	best	O(1)
	In-practice	O(1 + λ)
	worst	O(n)
remove(key)	best	O(1)
	In-practice	O(1 + λ)
	worst	O(n)



In-Practice 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 λ = $\frac{n}{c}$

Handling Collisions

Solution 2: Open Addressing

Resolves collisions by choosing a different location to tore a value if natural choice is already full.

Type 1: Linear Probing

If there is a collision, keep checking the next element until we find an open spot.

```
public int hashFunction(String s)
    int naturalHash = this.getHash(s);
    if(natural hash in use) {
        int i = 1;
        while (index in use) {
            try (naturalHash + i);
            i++;
```

Type 2: Quadratic Probing

If we collide instead try the next i^2 space

```
public int hashFunction(String s)
    int naturalHash = this.getHash(s);
    if(natural hash in use) {
        int i = 1;
        while (index in use) {
            try (naturalHash + i * i);
            i++;
```

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
1, 5, 11, 7, 12, 17, 6, 25

0	1	2	3	4	5	6	7	8	9
	11	12			25	6	17		

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

0	1	2	3	4	5	6	7	8	9
		58	79					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

Problems:
If λ≥ ½ we might never find an empty spot
    Infinite loop!
Can still get clusters

Handling Collisions

Solution 3: Double Hashing

If the natural hash location is taken, apply a second and separate hash function to find a new location. h’(k, i) = (h(k) + i * g(k)) % T

```
public int hashFunction(String s)
    int naturalHash = this.getHash(s);
    if(natural hash in use) {
        int i = 1;
        while (index in use) {
            try (naturalHash + i * jump_Hash(key));
            i++;
```

Homework

Homework 2

ArrayDictionary<K, V>

Function	Best case	Worst case
get(K key)	O(1) Key is first item looked at	O(n) Key is not found
put(K key, V value)	O(1) Key is first item looked at	2n -> O(n) N search, N resizing
remove(K key)	O(1) Key is first item looked at	O(n) N search, C swapping
containsKey(K key)	O(1) Key is first item looked at	O(n) Key is not found
size()	O(1) Return field	O(1) Return field

DoubleLinkedList<T>

Function	Best case	Worst case
get(int index)	O(1) Index is 0 or size	n/2 -> O(n) Index is size/2
add(T item)	O(1) Item added to back	O(1) Item added to back
remove()	O(1) Item removed from back	O(1) Item removed from back
delete(int index)	O(1) Index is 0 or size	n/2 -> O(n) Index is size/2
set(int index, T item)	O(1) Index is 0 or size	n/2 -> O(n) Index is size/2
insert(int index, T item)	O(1) Index is 0 or size	n/2 -> O(n) Index is size/2