Lecture 14: Midterm Review

Data Structures and Algorithms

Administrivia

Snow delays
- you get 3 more late days for the quarter
- you may now turn in any assignment up to 72 hours after deadline instead of 48 hours

HW 4 is due Friday
- turn in via grade scope, you have been added

We are moving to Piazza!
- https://piazza.com/class/js313vs4yym1tg
- You have all been added

Midterm Prep
- Attend section tomorrow, solutions have already been posted
- Erik filmed a review session, check it out
- I am posting another set of “Monday” slides
- HW 4 is midterm review

HW 5 goes out Friday
- Partner form due Thursday 11:59pm

A message about grades…

Midterm Logistics

50 minutes

8.5 x 11 in note page, front and back

Math identities sheet provided (see posted on website)

We will be scanning your exams to grade…

- Do not write on the back of pages
- Try not to cram answers into margins or corners

Midterm Topics

ADTs and Data Structures

- Lists, Stacks, Queues, Maps/Dictionaries
- Array vs Node implementations of each

Asymptotic Analysis

- Proving Big-O by finding a c and N_0
- Modeling code runtime with math functions, including recurrences and summations
- Finding closed form of recurrences using unrolling, tree method and master theorem
- Looking at code models and giving Big O runtimes
- Definitions of Big O, Big Omega, Big Theta

BST and AVL Trees

- Binary Search Property, Balance Property
- Insertions, Retrievals
- AVL rotations

Hashing

- Understanding hash functions
- Insertions and retrievals from a table
- Collision resolution strategies: chaining, linear probing, quadratic probing, double hashing

Heaps

- Heap properties
- Insertions, retrievals while maintaining structure with bubbling up

Homework

- ArrayDictionary
- DoubleLinkedList
- ChainedHashDictionary
- ChainedHashSet

NOT on the exam

- Memory and Locality
- B-Trees
- JUnit specifics and JUnit syntax

Asymptotic Analysis

Asymptotic Analysis

asymptotic analysis – the process of mathematically representing runtime of a algorithm in relation to the number of inputs and how that relationship changes as the number of inputs grow

Two step process

1. **Model** – the process of mathematically representing how many operations a piece of code will run in relation to the number of inputs n
2. **Analyze** – compare runtime/input relationship across multiple algorithms
    1. Graph the model of your code where x = number of inputs and y = runtime
    2. For which inputs will one perform better than the other?

Code Modeling

code modeling – the process of mathematically representing how many operations a piece of code will run in relation to the number of inputs n

Examples:

- Sequential search $f(n) = n$
- Binary search $f(n) = log_2 n$

What counts as an "operation"?

Assume all operations run in equivalent time

Basic operations

- Adding ints or doubles
- Variable assignment
- Variable update
- Return statement
- Accessing array index or object field

Consecutive statements

- Sum time of each statement

Function calls

- Count runtime of function body

Conditionals

- Time of test + worst case scenario branch

Loops

- Number of iterations of loop body x runtime of loop body

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
}
```

$$f(n) = 3 + \frac{3}{4}n = C_1 + \frac{3}{4}n$$

Code Modeling Example

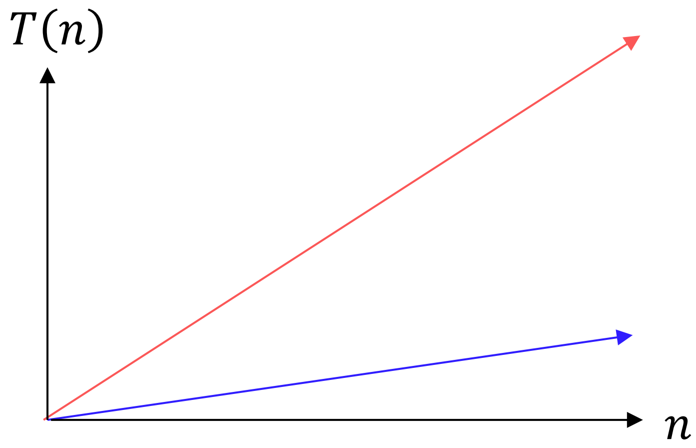
```
     public String mystery (int n) {
+1     ChainedHashDictionary<Integer, Character> alphabet =
                                        new ChainedHashDictionary<Integer, Character>();
       for (int i = 0; i < 26; i++) {
          char c = 'a' + (char)i;                 +26c
          alphabet.put(i, c);
       }
+1     DoubleLinkedList<Character> result = new DoubleLinkedList<Character>();
       for (int i = 0; i < n; i += 2) {
          char c = alphabet.get(i); +26c              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;
     }
```

$$f(n) = 4 + 26 + 27\left(\frac{n}{2}\right) + \frac{n}{2} = C_1 + C_2 n$$

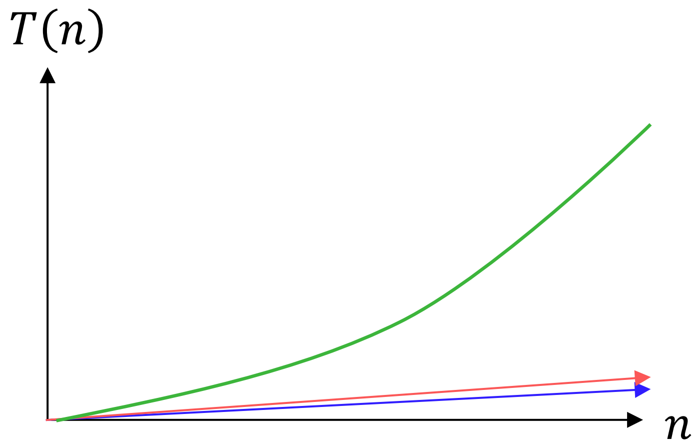
Function growth

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

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

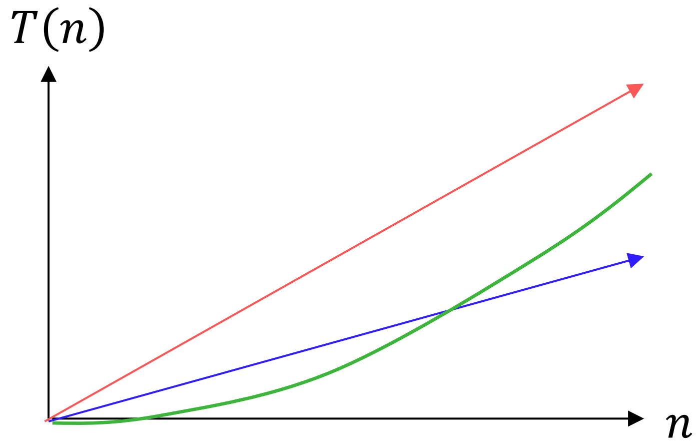


The growth rate for f(n) and g(n) looks very different for small numbers of input



...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)

O, Ω, Θ Definitions

***O*(f(n))** is the "family" or "set" of all functions that are dominated by 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 dominate 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)).

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

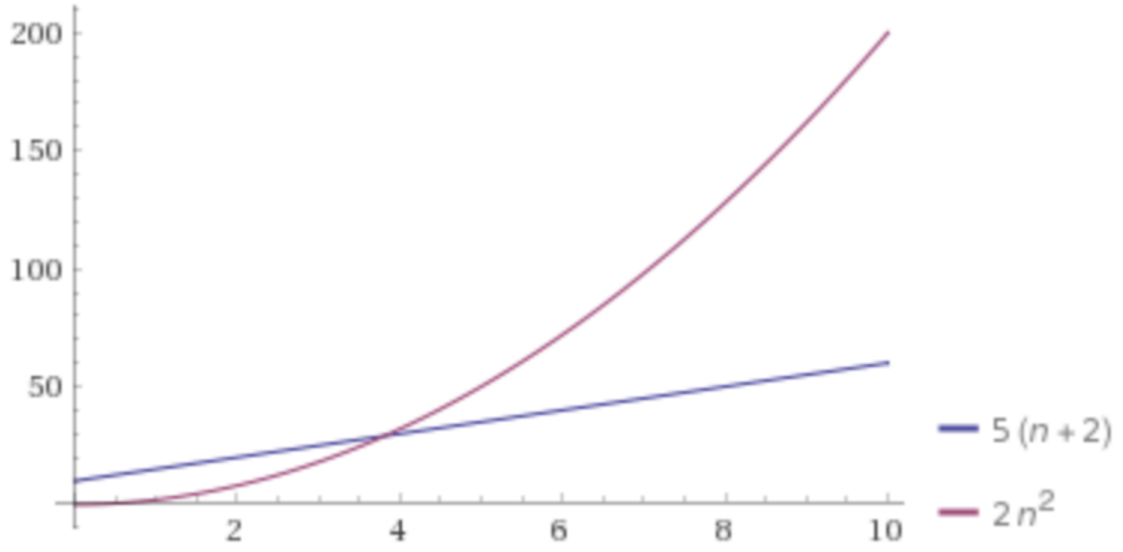
$5n \leq c \cdot 2\text{n}^2$ for c = 3 when n ≥ 1

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

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

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

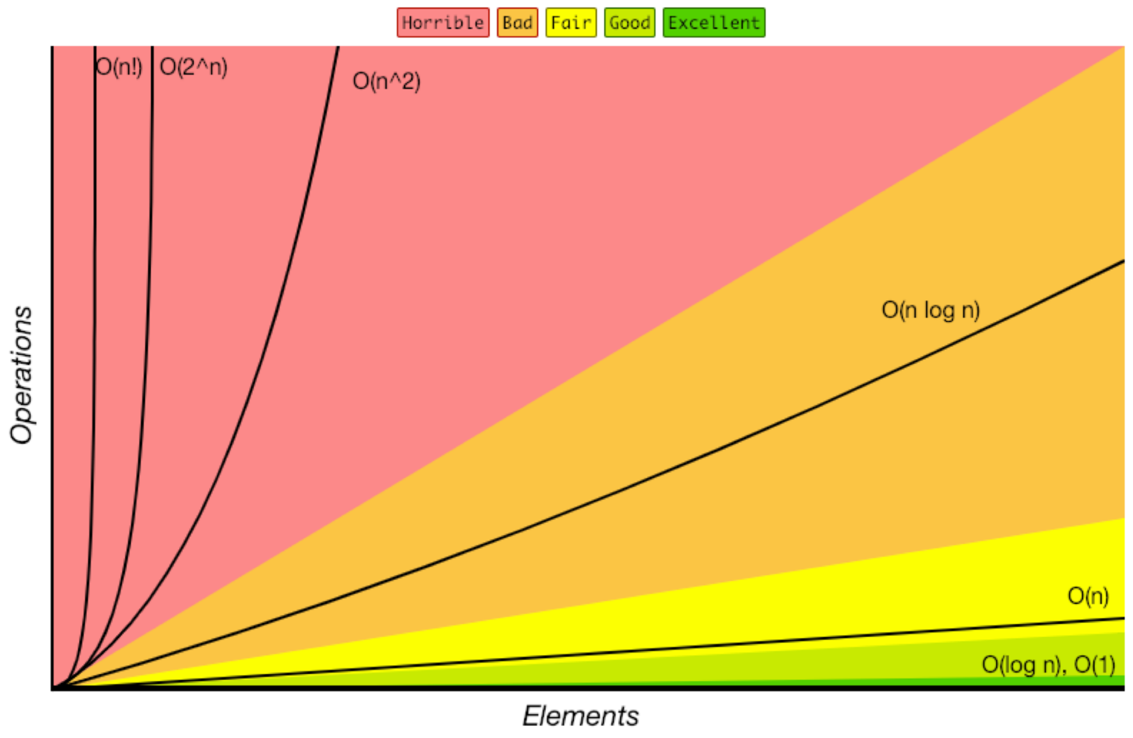
For the above functions indicate whether the following are true or false

$a(n) \in O(b(n))$	TRUE	$b(n) \in O(a(n))$	FALSE	$c(n) \in O(b(n))$	TRUE
$a(n) \in O(c(n))$	TRUE	$b(n) \in O(c(n))$	FALSE	$c(n) \in O(a(n))$	FALSE
$a(n) \in \Omega(b(n))$	FALSE	$b(n) \in \Omega(a(n))$	TRUE	$c(n) \in \Omega(b(n))$	FALSE
$a(n) \in \Omega(c(n))$	FALSE	$b(n) \in \Omega(c(n))$	TRUE	$c(n) \in \Omega(a(n))$	TRUE
$a(n) \in \Theta(b(n))$	FALSE	$b(n) \in \Theta(a(n))$	FALSE	$c(n) \in \Theta(b(n))$	FALSE
$a(n) \in \Theta(c(n))$	FALSE	$b(n) \in \Theta(c(n))$	FALSE	$c(n) \in \Theta(a(n))$	FALSE
$a(n) \in \Theta(a(n))$	TRUE	$b(n) \in \Theta(b(n))$	TRUE	$c(n) \in \Theta(c(n))$	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
log-linear	$O(n\log_2 n)$	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
    }
}
```

+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) {
        return 1;
    } else {
        return n * factorial(n – 1);
}
```

+C_1 (if / return 1)
+$T(n-1)$
+C_2

$$T(n) = \begin{cases} C_1 & \text{when } n = 0 \text{ or } 1 \\ C_2 + T(n-1) & \text{otherwise} \end{cases}$$

Definition: Recurrence

Mathematical equivalent of an if/else statement

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

Unrolling Method

Walk through function definition until you see a pattern

$$T(n) = \begin{cases} 4 & when\ n = 0,1 \\ 2 + T(n-1) & otherwise \end{cases}$$

$T(n) = \boxed{\boxed{}}$ ➡ $T(n) = \begin{cases} 4\ when\ n = 0,1 \\ 2 + T(n-1)\ otherwise \end{cases}$ ➡ $T(n) = 2 + \boxed{2 + T(\boxed{n-1} - 1)}$ *i = 1*

$T(n) = 2 + 2 + \boxed{T(\boxed{n-2})}$ ➡ $T(n) = \begin{cases} 4\ when\ n = 0,1 \\ 1 + T(n-1)\ otherwise \end{cases}$ ➡ $T(n) = 2 + 2 + \boxed{2 + T(\boxed{n-2} - 1)}$ *i = 2*

$T(n) = 2 + 2 + 2 + \boxed{T(\boxed{n-3})}$ ➡ $T(n) = \begin{cases} 4\ when\ n = 0,1 \\ 1 + T(n-1)\ otherwise \end{cases}$ ➡ $T(n) = 2 + 2 + 2 + \boxed{2 + T(\boxed{n-3} - 1)}$ *i = 3*

$T(n) = 2 + 2 + 2 + 2 + \cdots + T(1) = 2 + 2 + 2 + 2 + \ldots + 4$ *T(n-i) = T(1) when i= n-1*

n-1 recursive cases   1 base case

$T(n) = 4 + \sum_{i=1}^{n-1} 2$ ➡ Summation of a constant $\sum_{i=1}^{n} c = cn$ ➡ $\boxed{T(n) = 4 + 2(n-1)}$

$$T(4) = 2 + T(4-1) = 2 + 2 + T(3-1) = 2 + 2 + 2 + T(2-1) = 2 + 2 + 2 + 4 = 3 * 2 + 4$$

Unrolling Example

$$T(n) = \begin{cases} 1 & when\ n = 1 \\ 2T(n-1) + 3 & else \end{cases}$$

$2T(n-1) + 3$
$2(2T(n-2) + 3) + 3$
$2(2(2(T(n-3) + 3) + 3) + 3$
$= 2^3T(n-3) + 2^2 3 + 2^1 3 + 2^0 3$
$$= 2^iT(n-i) + \sum_{k=0}^{i-1} 3(2^k)$$

$base\ case\ when\ n - i = 1 \rightarrow i = n - 1$

$$= 2^{n-1}T(1) + \sum_{k=0}^{n-2} 3\left(2^k\right) = 2^{n-1} + 3(2^{n-1} - 1)$$

$T(n) = 2^{n+1} - 3$

Tree Method Formulas

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

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. How many recursive levels are there?
   - Based on the pattern of how we get down to base case

branchCount = $\log_2 n - 1$

$$Recursive\ work = \sum_{i=0}^{branchCount} branchNum(i) branchWork(i)$$

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

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

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

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

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

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

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

How many levels of the tree? $\log_3(n)$

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? $3n$

$T(n) = n \log_3(n) + 3n$

Reflecting on Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ when } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} \end{cases}$$

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

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

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

$height \approx \log_b a$

$branchWork \approx n^c \log_b a$

$leafWork \approx d\left(n^{\log_b a}\right)$

The $\log_b a < c$ case

- Recursive case conquers work more quickly than it divides work
- Most work happens near “top” of tree
- Non recursive work in recursive case dominates growth, n^c term

The $\log_b a = c$ case

- Work is equally distributed across call stack (throughout the “tree”)
- Overall work is approximately work at top level x height

The $\log_b a > c$ case

- Recursive case divides work faster than it conquers work
- Most work happens near “bottom” of tree
- Leaf work dominates branch work

Master Theorem Example

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

$a = 3$
$b = 3$
$c = 1$
$\log_3 3 = 1$
$T(n)\ is\ in\ \theta(nlogn)$

Given a recurrence of the form:

$$T(n) = \begin{cases} d \ when\ n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \ otherwise \end{cases}$$

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

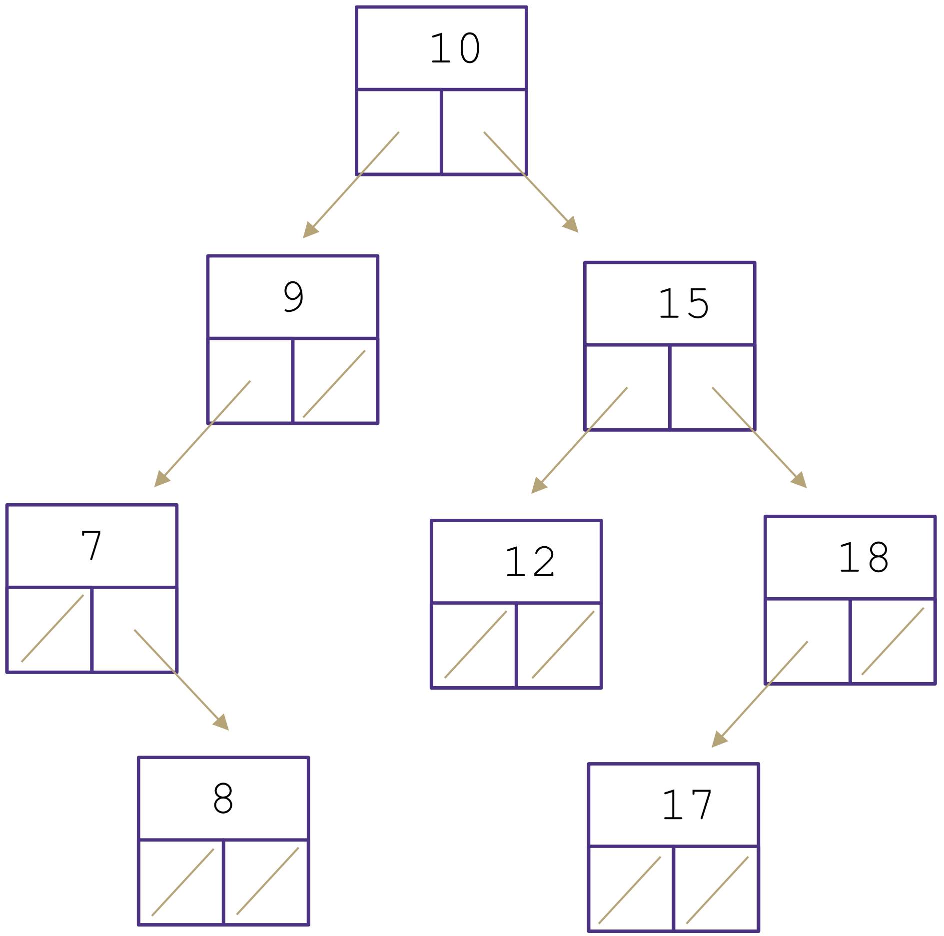
If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

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

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

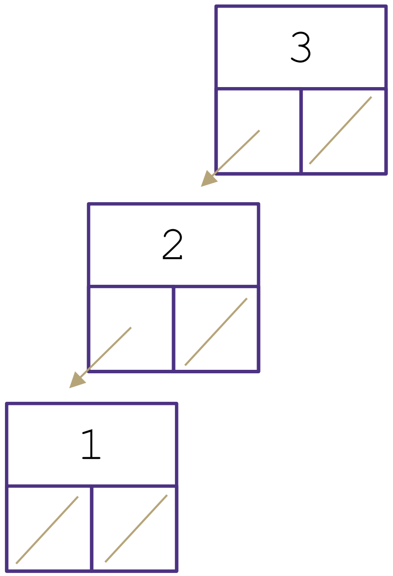
AVL Trees must satisfy the following properties:

- binary 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

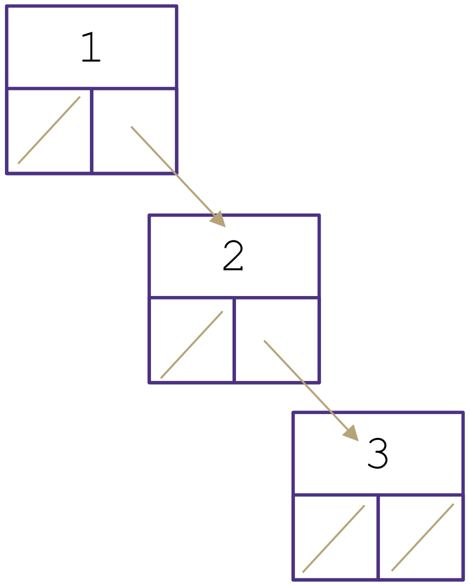
AVL stands for **A**delson-**V**elsky and **L**andis (the inventors of the data structure)

Two AVL Cases

Line Case
Solve with **1** rotation

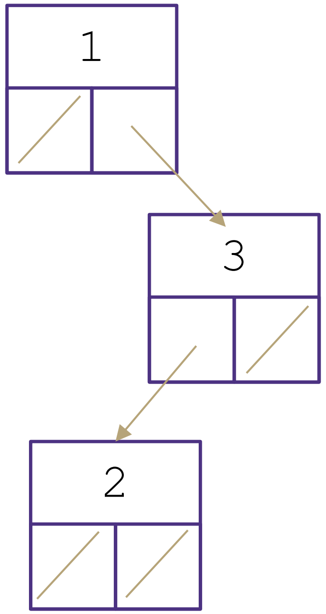


Rotate Right
Parent's left becomes child's right
Child's right becomes its parent

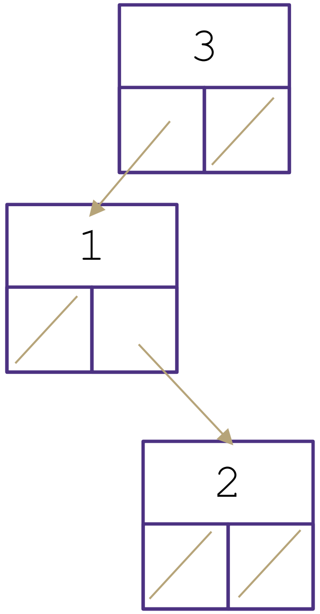


Rotate Left
Parent's right becomes child's left
Child's left becomes its parent

Kink Case
Solve with **2** rotations



Right Kink Resolution
Rotate subtree left
Rotate root tree right



Left Kink Resolution
Rotate subtree right
Rotate root tree left

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	"poo"	"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, "poo"); 20 % 10 = 0
```

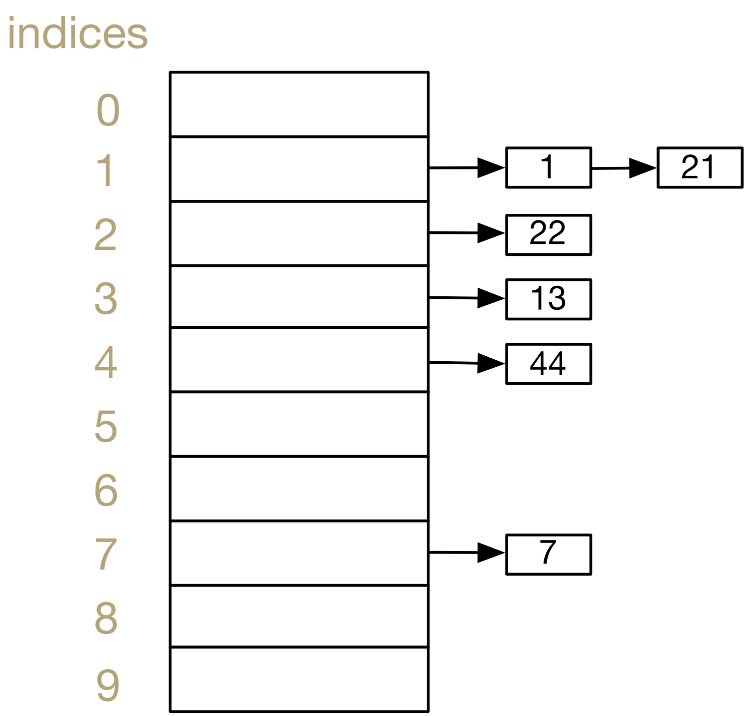
→ Collision!

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)
	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 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 $\lambda \geq \frac{1}{2}$ 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++;
```

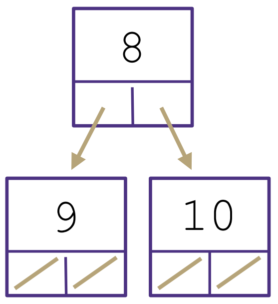
Heaps

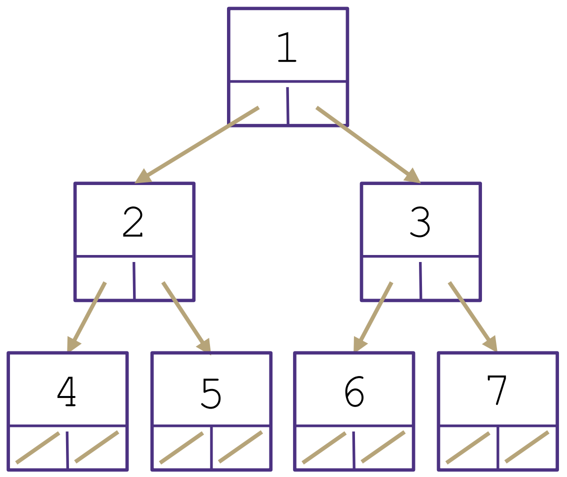
Binary Heap

A type of tree with new set of invariants

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

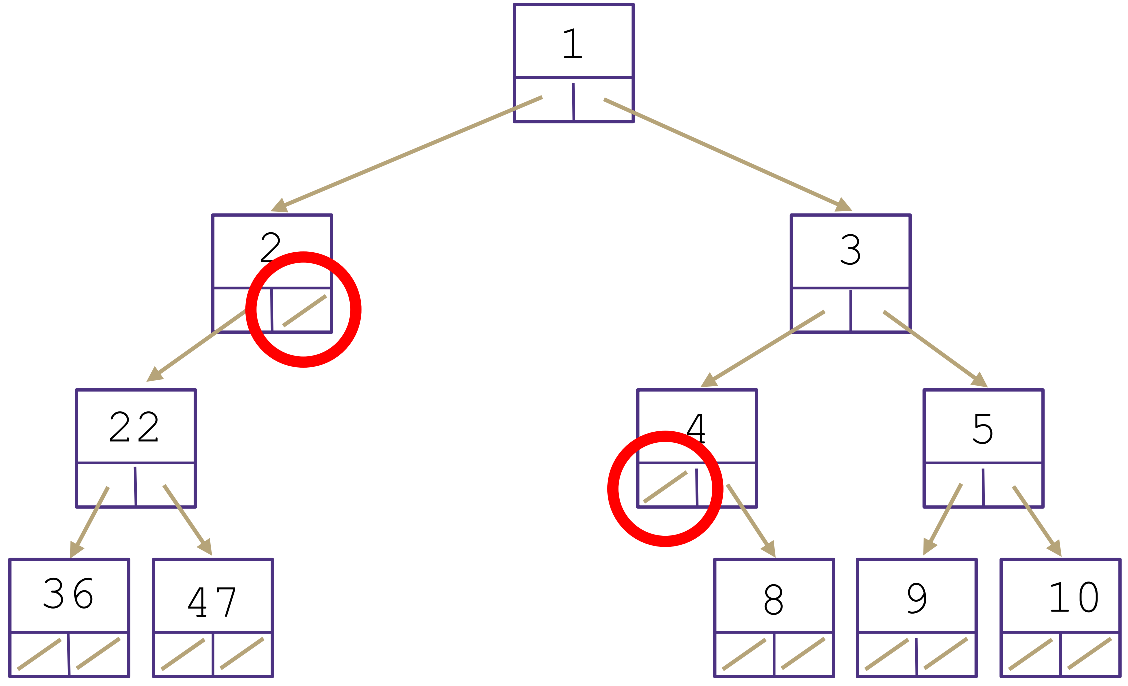
2. **Heap**: every node is smaller than its child



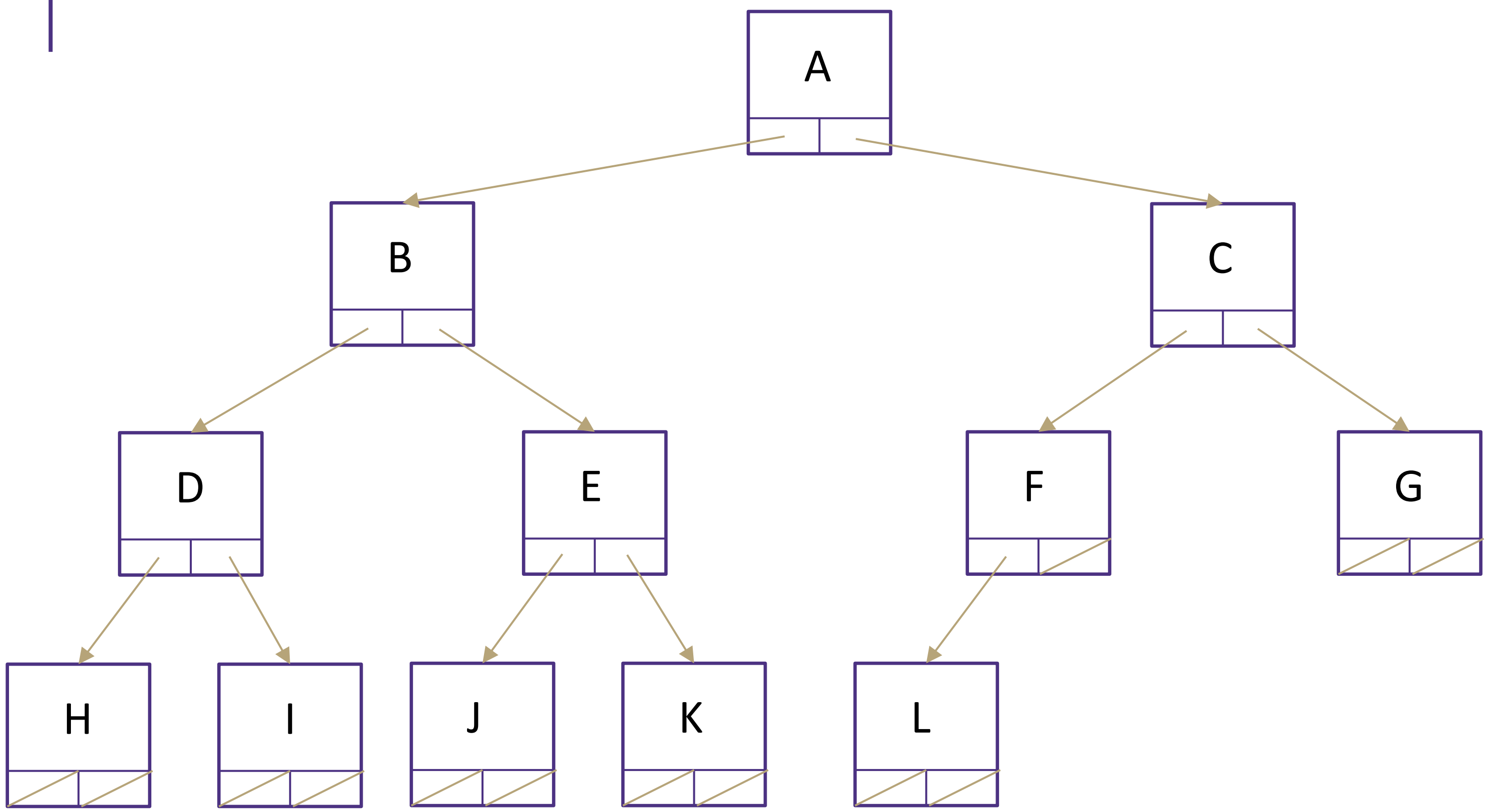


3. **Structure:** Each level is "complete" meaning it has no "gaps"

- Heaps are filled up left to right



Implementing Heaps



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?

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

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

Homework 3

ChainedHashDictionary<K, V>

Function	Best case	Worst case
get(K key)	O(1) Chain size of 1	O(n) Chain size of n
put(K key, V value)	O(1) Add into empty bucket	3N -> O(n) N search in chain, N resizing of chain, N resizing of Hash
remove(K key)	O(1) Chain of size 1	O(n) Chain of size n
containsKey(K key)	O(1) Key is first item in chain / empty chain	O(n) Chain of size n, key not found
size()	O(1) Return field	O(1) Return field

ChainedHashSet<T>

Function	Best case	Worst case
add(T item)	O(1) Add into empty bucket	3N -> O(n) N search in chain, N resizing of chain, N resizing of Hash
remove(T item)	O(1) Chain of size 1	O(n) Chain of size n
contains(T item)	O(1) Item is at front of chain	O(n) Chain of size n, item not found
size()	O(1) Return field	O(1) Return field