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 succccked
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**

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

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

Note: You don’t have to understand all of this right now – we’ll dive into it soon.
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**
- `get(index)` return item at index
- `set(item, index)` replace item at index
- `append(item)` add item to end of list
- `insert(item, index)` add item at index
- `delete(index)` delete item at index
- `size()` count of items

**ArrayList**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
</table>
| `ArrayList<E>` | \n| `data[]` | return data[index] 
| `size` | set data[index] = value 
| `append` data[size] = value, if out of space 
| `insert` shift values to make hole at index, data[index] = value, if out of space 
| `delete` shift following values forward 
| `size` return size |

**LinkedLIst**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
</table>
| `LinkedList<E>` | \n| `Node front` | get loop until index, return node’s value 
| `size` | set loop until index, update node’s value 
| `append` create new node, loop until index, update next fields 
| `insert` create new node, loop until index, update next fields 
| `delete` loop until index, skip node 
| `size` return size |

**ArrayList**

- uses an Array as underlying storage

**LinkedList**

- uses nodes as underlying storage

---

![Diagram of ArrayList](image1)

![Diagram of LinkedList](image2)

```plaintext
<table>
<thead>
<tr>
<th></th>
<th><code>ArrayList&lt;E&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>88.6</td>
</tr>
<tr>
<td>1</td>
<td>26.1</td>
</tr>
<tr>
<td>2</td>
<td>94.4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th><code>LinkedList&lt;E&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>88.6</td>
</tr>
<tr>
<td></td>
<td>26.1</td>
</tr>
<tr>
<td></td>
<td>94.4</td>
</tr>
<tr>
<td></td>
<td>list</td>
</tr>
<tr>
<td></td>
<td>free space</td>
</tr>
</tbody>
</table>
```
**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

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
<td>push(item) add item to top</td>
</tr>
<tr>
<td>Number of items</td>
<td>pop() return and remove item at top</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pop()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>peek()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>push()</code></td>
<td>O(N) linear if you have to resize O(1) otherwise</td>
</tr>
</tbody>
</table>

---

**Push Operations**

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

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>peek()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>push()</td>
<td>O(1) Constant</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Queue ADT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>state</strong></td>
</tr>
<tr>
<td>Set of ordered items</td>
</tr>
<tr>
<td>Number of items</td>
</tr>
<tr>
<td><strong>behavior</strong></td>
</tr>
<tr>
<td>add(item) add item to back</td>
</tr>
<tr>
<td>remove() remove and return item at front</td>
</tr>
<tr>
<td>peek() return item at front</td>
</tr>
<tr>
<td>size() count of items</td>
</tr>
<tr>
<td>isEmpty() count of items is 0?</td>
</tr>
</tbody>
</table>

**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

---

0 1 2 3 4

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`numberOfItems = 3`

`front = 1`

`back = 2`

---

Take 1 min to respond to activity

What do you think the worst possible runtime of the “add()” operation will be?

www.pollev.com/cse373activity

CSE 373 19 WI - KASEY CHAMPION
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

What do you think the worst case runtime of the “add()” operation will be?

www.pollev.com/cse373activity
Review: Dictionaries

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>HashTable</th>
<th>BST</th>
<th>AVLTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>get(key)</td>
<td>best</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>
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

---

| 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  |

---

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

```plaintext
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.

---

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>('a', 1)</td>
<td>('b', 97)</td>
<td>('c', 3)</td>
<td>('d', 4)</td>
<td>('e', 20)</td>
</tr>
</tbody>
</table>
```
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>**

```java
LinkedMap<K, V>...
```

**state**
- `front`
- `size`

**behavior**
- `put(key, item)` 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

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

\[
f(n) = 2
\]

\[
f(n) = 3n + 1
\]

Asymptotic Analysis

1. **BEST CASE FUNCTION**
   \[
   f(n) = 2
   \]

2. **WORST CASE FUNCTION**
   \[
   f(n) = 3n + 1
   \]

**CODE**

**Case Analysis**

**TIGHT BIG-OH**

**BIG-Omega**

**BIG-THETA**

**O(n)**

**\( \Theta(n) \)**

**\( \Omega(n) \)**
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
}
**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**

\[
f(n) \text{ is } O(g(n)) \text{ if there exist positive constants } c,n_0 \text{ such that for all } n \geq n_0,
\]

\[
f(n) \leq c \cdot g(n)
\]

**Big-Omega**

\[
f(n) \text{ is } \Omega(g(n)) \text{ if there exist positive constants } c,n_0 \text{ such that for all } n \geq n_0,
\]

\[
f(n) \geq c \cdot g(n)
\]

**Big-Theta**

\[
f(n) \text{ is } \Theta(g(n)) \text{ if } f(n) \text{ is } O(g(n)) \text{ and } f(n) \text{ is } \Omega(g(n)).
\]

(in other words: there exist positive constants \(c_1, c_2, n_0\) such that for all \(n \geq n_0\))

\[
c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
\]
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. However, as the input size increases, the growth rate for both \( f(n) \) and \( g(n) \) eventually look similar at large input sizes. Whereas \( h(n) \) has a distinctly different growth rate than either \( f(n) \) or \( g(n) \). But for very small input values, \( h(n) \) actually has a slower growth rate than either \( f(n) \) or \( g(n) \).
Examples

4n² ∈ Ω(1)
true

4n² ∈ Ω(n)
true

4n² ∈ Ω(n²)
true

4n² ∈ Ω(n³)
false

4n² ∈ Ω(n⁴)
false

4n² ∈ O(1)
false

4n² ∈ O(n)
false

4n² ∈ O(n²)
true

4n² ∈ O(n³)
true

4n² ∈ O(n⁴)
true

Big-O

f(n) ∈ O(g(n)) if there exist positive constants c, n₀ such that for all n ≥ n₀,

f(n) ≤ c · g(n)

Big-Omega

f(n) ∈ Ω(g(n)) if there exist positive constants c, n₀ such that for all n ≥ n₀,

f(n) ≥ c · g(n)

Big-Theta

f(n) ∈ Θ(g(n)) if

f(n) is O(g(n)) and f(n) is Ω(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 \(\rightarrow\) function step instead of function \(\rightarrow\) \(O/\Omega/\Theta\) 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!

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);
        }
    }
}

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

\[ a = 16, \quad b = 4, \quad c = 0 \]

\[ \log_4 16 = 2 \Rightarrow \theta(n^{\log_4 16}) = \theta(n^2) \]

Master Theorem

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

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

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} \]
Tree Method

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

- **Input**: \( n \)
- **Work**: \( n \)

---

**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 | \( \frac{n}{8} \) | \( n \)
... | ... | ... | ...
Tree Method Practice

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? \(\frac{n}{2^i}\)

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?

\[
\text{numNodes} \times \text{workPerNode} = 2^i \left(\frac{n}{2^i}\right) = n
\]

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?

\[
\text{numNodes} \times \text{workPerNode} = 2^{\log_2 n} (1) = n
\]

Combining it all together...

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

\[
T(n) = \sum_{i=0}^{\log_2 n - 1} n + n = n \log_2 n + n = \Theta(n \log n)
\]
Separate chaining

// 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)
}

runtime analysis
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.
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

\[
\lambda = \frac{n}{c}
\]

LOAD FACTOR $\lambda$

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

\[
\lambda = \frac{n}{c} = 1.6
\]
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

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
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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>58</td>
<td>79</td>
<td></td>
<td></td>
<td>27</td>
<td>18</td>
<td>49</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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 λ ≥ ½ 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>best</th>
<th>Array w/ indices as keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(1 + (\lambda))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(n)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>best</th>
<th>Array w/ indices as keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(key)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(1 + (\lambda))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(n)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>best</th>
<th>Array w/ indices as keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove(key)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(1 + (\lambda))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(n)</td>
<td></td>
</tr>
</tbody>
</table>

**Average Case:**
Depends on average number of elements per chain.

**Load Factor \(\lambda\)**
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 Adelson-Velsky and Landis (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
Is this a valid AVL tree?

Is it...
- Binary \textbf{yes}
- BST \textbf{yes}
- Balanced? \textbf{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?
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?

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

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” |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>behavior</strong></td>
<td></td>
</tr>
</tbody>
</table>
- **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
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()
**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

How do we find the minimum node?

\[ \text{peekMin}() = \text{arr}[0] \]

How do we find the last node?

\[ \text{lastNode}() = \text{arr}[\text{size} - 1] \]

How do we find the next open space?

\[ \text{openSpace}() = \text{arr}[\text{size}] \]

How do we find a node’s left child?

\[ \text{leftChild}(i) = 2i + 1 \]

How do we find a node’s right child?

\[ \text{rightChild}(i) = 2i + 2 \]

How do we find a node’s parent?

\[ \text{parent}(i) = \frac{(i - 1)}{2} \]

Fill array in level-order from left to right

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implement Heaps with an array

How do we find the minimum node?

\[ \text{peekMin}() = \text{arr}[1] \]

How do we find the last node?

\[ \text{lastNode}() = \text{arr}[\text{size}] \]

How do we find the next open space?

\[ \text{openSpace}() = \text{arr}[\text{size} + 1] \]

How do we find a node’s left child?

\[ \text{leftChild}(i) = 2i \]

How do we find a node’s right child?

\[ \text{rightChild}(i) = 2i + 1 \]

How do we find a node’s parent?

\[ \text{parent}(i) = \frac{i}{2} \]

Fill array in **level-order** from left to right
Array-Implemented MinHeap Runtimes

<table>
<thead>
<tr>
<th>Operation</th>
<th>Case</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeMin()</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(log n)</td>
</tr>
<tr>
<td></td>
<td>in practice</td>
<td>Θ(log n)</td>
</tr>
<tr>
<td>add(key)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(log n)</td>
</tr>
<tr>
<td></td>
<td>in practice</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>peekMin()</td>
<td>all cases</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>

- 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!