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 – 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”?

**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

Assume all operations run in equivalent time
Code Modeling

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

\[ f(n) = 3 + \frac{3}{4}n = C_1 + \frac{3}{4}n \]
public String mystery (int n) {
    ChainedHashDictionary<Integer, Character> alphabet =
        new ChainedHashDictionary<Integer, Character>();
    for (int i = 0; i < 26; i++) {
        char c = ‘a’ + (char)i;
        alphabet.put(i, c);
    }
    DoubleLinkedList<Character> result = new DoubleLinkedList<Character>();
    for (int i = 0; i < n; i += 2) {
        char c = alphabet.get(i);
        result.add(c);
    }
    String final = “”;
    for (int i = 0; i < result.size(); i++) {
        final += result.remove();
    }
    return final;
}
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

<table>
<thead>
<tr>
<th>Big-O</th>
<th>f(n) ∈ O(g(n)) if there exist positive constants c, n₀ such that for all n ≥ n₀, f(n) ≤ c · g(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-Omega</td>
<td>f(n) ∈ Ω(g(n)) if there exist positive constants c, n₀ such that for all n ≥ n₀, f(n) ≥ c · g(n)</td>
</tr>
<tr>
<td>Big-Theta</td>
<td>f(n) ∈ Θ(g(n)) if f(n) is O(g(n)) and f(n) is Ω(g(n)).</td>
</tr>
</tbody>
</table>
Proving Domination

\[ f(n) = 5(n + 2) \]
\[ g(n) = 2n^2 \]

Find a \( c \) and \( n_0 \) that show that \( f(n) \in O(g(n)) \).

\[ f(n) = 5(n + 2) = 5n + 10 \]
\[ 5n \leq c \cdot 2n^2 \text{ for } c = 3 \text{ when } n \geq 1 \]
\[ 10 \leq c \cdot 2n^2 \text{ for } c = 5 \text{ when } n \geq 1 \]
\[ 5n + 10 \leq 3(2n^2) + 5(2n^2) \text{ when } n \geq 1 \]
\[ 5n + 10 \leq 8(2n^2) \text{ when } n \geq 1 \]
\[ f(n) \leq c \cdot g(n) \text{ when } c = 8 \text{ and } n_0 = 1 \]
O, Ω, Θ Examples

For the following functions give the simplest tight O bound

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

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

\[ a(n) \in O(b(n)) \quad \text{TRUE} \]
\[ b(n) \in O(a(n)) \quad \text{FALSE} \]
\[ c(n) \in O(b(n)) \quad \text{TRUE} \]
\[ a(n) \in O(c(n)) \quad \text{TRUE} \]
\[ b(n) \in O(c(n)) \quad \text{FALSE} \]
\[ c(n) \in O(a(n)) \quad \text{FALSE} \]
\[ a(n) \in \Omega(b(n)) \quad \text{FALSE} \]
\[ b(n) \in \Omega(a(n)) \quad \text{TRUE} \]
\[ c(n) \in \Omega(b(n)) \quad \text{FALSE} \]
\[ a(n) \in \Omega(c(n)) \quad \text{FALSE} \]
\[ b(n) \in \Omega(c(n)) \quad \text{TRUE} \]
\[ c(n) \in \Omega(a(n)) \quad \text{TRUE} \]
\[ a(n) \in \Theta(b(n)) \quad \text{FALSE} \]
\[ b(n) \in \Theta(a(n)) \quad \text{FALSE} \]
\[ c(n) \in \Theta(b(n)) \quad \text{FALSE} \]
\[ a(n) \in \Theta(c(n)) \quad \text{FALSE} \]
\[ b(n) \in \Theta(c(n)) \quad \text{FALSE} \]
\[ c(n) \in \Theta(a(n)) \quad \text{FALSE} \]
\[ a(n) \in \Theta(a(n)) \quad \text{TRUE} \]
\[ b(n) \in \Theta(b(n)) \quad \text{TRUE} \]
\[ c(n) \in \Theta(c(n)) \quad \text{TRUE} \]
Review: Complexity Classes

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Big O</th>
<th>If you double N...</th>
<th>Example algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>O(1)</td>
<td>unchanged</td>
<td>Add to front of linked list</td>
</tr>
<tr>
<td>logarithmic</td>
<td>O(log₂n)</td>
<td>Increases slightly</td>
<td>Binary search</td>
</tr>
<tr>
<td>linear</td>
<td>O(n)</td>
<td>doubles</td>
<td>Sequential search</td>
</tr>
<tr>
<td>log-linear</td>
<td>O(nlog₂n)</td>
<td>Slightly more than doubles</td>
<td>Merge sort</td>
</tr>
<tr>
<td>quadratic</td>
<td>O(n²)</td>
<td>quadruples</td>
<td>Nested loops traversing a 2D array</td>
</tr>
<tr>
<td>cubic</td>
<td>O(n³)</td>
<td>Multiplies by 8</td>
<td>Triple nested loop</td>
</tr>
<tr>
<td>polynomial</td>
<td>O(n^c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exponential</td>
<td>O(c^n)</td>
<td>Multiplies drastically</td>
<td></td>
</tr>
</tbody>
</table>

http://bigocheatsheet.com/
Modeling Complex Loops

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

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

```java
generic
public int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

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

**Definition: Recurrence**

Mathematical equivalent of an if/else statement

\[ f(n) = \begin{cases} 
  \text{runtime of base case when conditional} \\
  \text{runtime of recursive case otherwise}
\end{cases} \]
Unrolling Method

Walk through function definition until you see a pattern

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

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

\[ T(n) = 2 + 2 + T(n-1) \]

\[ T(n) = 2 + 2 + 2 + T(n-2) \]

\[ T(n) = 2 + 2 + 2 + 2 + \cdots + T(1) = 2 + 2 + 2 + 2 + \cdots + 4 \]

\[ T(n-i) = T(1) \text{ when } i = n-1 \]

\[ T(n) = 4 + \sum_{i=1}^{n-1} 2 \]

\[ \sum_{i=1}^{n} c = cn \]

\[ T(n) = 4 + 2(n - 1) \]

\[ T(4) = 2 + T(4-1) = 2 + 2 + T(3-1) = 2 + 2 + 2 + T(2-1) = 2 + 2 + 2 + 4 = 3 \times 2 + 4 \]
Unrolling Example

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

\[
2T(n - 1) + 3 = 2(2T(n - 2) + 3) + 3 = 2^3T(n - 3) + 2^23 + 2^13 + 2^03
\]

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

**base case** when \( n - i = 1 \rightarrow i = n - 1 \)

\[
= 2^{n-1}T(1) + \sum_{k=0}^{n-2} 3(2^k) = 2^{n-1} + 3(2^{n-1} - 1)
\]

\[
T(n) = 2^{n+1} - 3
\]
Tree Method Formulas

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

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

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

Recursive work = \[ \sum_{i=0}^{branchCount} branchNum(i) \times branchWork(i) \]

branchCount = \log_2 n - 1

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?

2. How many leaf nodes are there?

NonRecursive work = leafWork \times leafCount

leafCount = 2^{\log_2 n} = n

T(n \leq 1) = 1 \times (2^{\log_2 n}) = n

\[
T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i \left( \frac{n}{2^i} \right) + n = n \log_2 n + n
\]
Tree Method Example

\[ T(n) = \begin{cases} 
3 & \text{when } n = 1 \\
3T \left( \frac{n}{3} \right) + n & \text{otherwise}
\end{cases} \]

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

Number of nodes at level \( i \)? \( 3^i \)

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

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

How much work done in base case? \( 3n \)

Total recursive work

\[
\sum_{i=0}^{\log_3 n - 1} \frac{n}{3^i} \cdot 3^i = n \log_3(n)
\]

\[ T(n) = n \log_3(n) + 3n \]
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 \( T(n) \in \Theta(n^{\log_b a}) \)

**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

**height \( \approx \log_b a \)**

**branchWork \( \approx n^c \log_b a \)**

**leafWork \( \approx d(n^{\log_b a}) \)**
Master Theorem Example

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

\[ a = 3 \quad b = 3 \quad c = 1 \quad \log_3 3 = 1 \quad T(n) \text{ is in } \Theta(n \log n) \]

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 \( T(n) \in \Theta(n^c \log n) \)
- If \( \log_b a > c \) then \( T(n) \in \Theta(n^{\log_b a}) \)
BST & AVL 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 Adelson-Velsky and Landis (the inventors of the data structure)
Two AVL Cases

**Line Case**
Solve with 1 rotation

**Kink Case**
Solve with 2 rotations

**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

**Right Kink Resolution**
Rotate subtree left
Rotate root tree right

**Left Kink Resolution**
Rotate subtree right
Rotate root tree left
Hashing
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.ensureIndexNotNull(key);
    this.data[key] = null;
}

public int getKey(int value) {
    return value % this.data.length;
}
### First Hash Function: % table size

<table>
<thead>
<tr>
<th>indices</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>element</td>
<td>&quot;poo&quot;</td>
<td>&quot;biz&quot;</td>
<td></td>
<td></td>
<td></td>
<td>&quot;bar&quot;</td>
<td></td>
<td></td>
<td></td>
<td>&quot;bop&quot;</td>
</tr>
</tbody>
</table>

```
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
```
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>Array w/ indices as keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key,value)</td>
<td>best: O(1)</td>
</tr>
<tr>
<td></td>
<td>average: O(1 + λ)</td>
</tr>
<tr>
<td></td>
<td>worst: O(n)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best: O(1)</td>
</tr>
<tr>
<td></td>
<td>average: O(1 + λ)</td>
</tr>
<tr>
<td></td>
<td>worst: O(n)</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best: O(1)</td>
</tr>
<tr>
<td></td>
<td>average: O(1 + λ)</td>
</tr>
<tr>
<td></td>
<td>worst: O(n)</td>
</tr>
</tbody>
</table>

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

Solution 2: Open Addressing

Resolves collisions by choosing a different location to store 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.

```java
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

```java
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++;
        }
    }
}
```
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
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

(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. \( h'(k, i) = (h(k) + i \times g(k)) \mod T \)

```java
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);
            } catch (IndexInUseException) {
                i++;
            }
        }
    }
    return naturalHash;
}
```
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

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

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></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<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>
</tr>
</tbody>
</table>
Homework
### ArrayDictionary<K, V>

<table>
<thead>
<tr>
<th>Function</th>
<th>Best case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(K key)</td>
<td>O(1) Key is first item looked at</td>
<td>O(n) Key is not found</td>
</tr>
<tr>
<td>put(K key, V value)</td>
<td>O(1) Key is first item looked at</td>
<td>2n -&gt; O(n) N search, N resizing</td>
</tr>
<tr>
<td>remove(K key)</td>
<td>O(1) Key is first item looked at</td>
<td>O(n) N search, C swapping</td>
</tr>
<tr>
<td>containsKey(K key)</td>
<td>O(1) Key is first item looked at</td>
<td>O(n) Key is not found</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) Return field</td>
<td>O(1) Return field</td>
</tr>
</tbody>
</table>

### DoubleLinkedList<T>

<table>
<thead>
<tr>
<th>Function</th>
<th>Best case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(int index)</td>
<td>O(1) Index is 0 or size</td>
<td>n/2 -&gt; O(n) Index is size/2</td>
</tr>
<tr>
<td>add(T item)</td>
<td>O(1) Item added to back</td>
<td>O(1) Item added to back</td>
</tr>
<tr>
<td>remove()</td>
<td>O(1) Item removed from back</td>
<td>O(1) Item removed from back</td>
</tr>
<tr>
<td>delete(int index)</td>
<td>O(1) Index is 0 or size</td>
<td>n/2 -&gt; O(n) Index is size/2</td>
</tr>
<tr>
<td>set(int index, T item)</td>
<td>O(1) Index is 0 or size</td>
<td>n/2 -&gt; O(n) Index is size/2</td>
</tr>
<tr>
<td>insert(int index, T item)</td>
<td>O(1) Index is 0 or size</td>
<td>n/2 -&gt; O(n) Index is size/2</td>
</tr>
</tbody>
</table>
## Homework 3

### ChainedHashDictionary\(<K, V>\)

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

### ChainedHashSet\(<T>\)

<table>
<thead>
<tr>
<th>Function</th>
<th>Best case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(T item)</td>
<td>O(1) Add into empty bucket</td>
<td>3N -&gt; O(n) N search in chain, N resizing of chain, N resizing of Hash</td>
</tr>
<tr>
<td>remove(T item)</td>
<td>O(1) Chain of size 1</td>
<td>O(n) Chain of size n</td>
</tr>
<tr>
<td>contains(T item)</td>
<td>O(1) Item is at front of chain</td>
<td>O(n) Chain of size n, item not found</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) Return field</td>
<td>O(1) Return field</td>
</tr>
</tbody>
</table>