CS61B - LAB REVIEW

Author: Kefan Li

1. Setting Up Your Computer(Lab 1)

This is an enviorment setting based lab.

This lab teaches us how to install Java environment variable and git in Windows, Mac and Linux.

My personal option is Mac and Linux. The operations are quite clear and easy to follow even if you have a little previous programming knowledge. By the way, the options in Windows are tricky contrasting to Mac and Linux.

We could learn how to use terminal to control our computer by learning lab1. For more details, please refer lab1Setting Up Your Computer

We could learn how to use **git** in lab1 javac, java, git. **Git** is crucial in our future working cases. Please learn that in a down to earth manner.

2. IntelliJ Set Up & Debugging learning and IntList(Lab 2)

The first part of this lab is to teach us how to set up IntelliJ, the most important IDE for our future CS61B learning. Please refer details in lab2 IntelliJ Set Up

The second part of this lab is to show using Debugger in IntelliJ. Debugger will accompany us for the whole programming carrer. We could learn **setting break points in our program and basic operations of step over, step into, step out etc.** Please refer details in lab2 Debugger Using.

The third part of this lab is to write several methods of IntList.

Codes for this lab is lab2 codes

A. Destructively Square the List

Destructive means we change the origin list

```
    public static void dSquareList(IntList L) {

        while (L != null) {
            L.first = L.first * L.first;
            L = L.rest;
        }
    }
```

Since `L` is not a primitive type of Java, it is *pass by reference*, so we decstructively square the list.

B. Non-Destructive Square the List

Non-destructive means we don't change the origin list.

```
    public static IntList squareListIterative(IntList L) {
        if (L == null) {
            return null;
        }
        IntList res = new IntList(L.first * L.first, null);
        IntList ptr = res;
        L = L.rest;
        while (L != null) {
            ptr.rest = new IntList(L.first * L.first, null);
            L = L.rest;
            ptr = ptr.rest;
        }
        return res;
    }
```

We use a very delicate way by creating a new pointer `res`, so that the following operation does not change the value of `L`. We create a pointer `ptr` to perform the squaring so that we can return res correctly.

C. Non-Destructive Square(Recursive)

```
    public static IntList squareListRecursive(IntList L) {
        if (L == null) {
            return null;
        }
        return new IntList(L.first * L.first,         squareListRecursive(L.rest));
    }
```

The recursive method uses `new` to create a new list, so `L` is never changed.

D. Destructively Catenate Lists

```
public static IntList dcatenate(IntList A, IntList B) {
    if (A == null) {
        return B;
    }

    IntList L;
    L = A;
    while (L.rest != null) { // XC: Why not L != null plus L = B ---> null is not an
address, cannot pass by reference;
        L = L.rest;
    }
    L.rest = B;
    return A;
}
```

This task is easy, the only tricky point is that we should set the while loop condition as `L.rest != null`, so that we can pass by reference. Otherwise, when `L = null`, the `null` value will be passed, which voids the function.

E. Non-Destructive Catenate Lists

```
   public static IntList catenate(IntList A, IntList B) {
        if (A == null) {
            return B;
        }
        return new IntList(A.first, catenate(A.rest, B));
    }
```

Again, since we return a new `IntList`, the original variables cannot be changed.

In summary, use `new` and not using any statments with repect to `L.first` usually make sure that the method is non-destructive.

3. JUnit Testing and Debugging(Lab 3)

Please refer Lab3 for more details about this lab. By the way, the Hug61B book did not mention how to use the *style checker*. The instruction is: Right click any files in IDEA and click *check style*, then the *CS61B Plugin* will do the job.

What is JUnit Testing?

The answer is the "Unit" part of Unit Testing comes from the idea that you can break your program down into units, or the smallest testable part of an application. Therefore, Unit Testing enforces good code structure (each method should only do "One Thing"), and allows you to consider all of the edge cases for each method and test for them individually.

Add Reverse() Method in IntList.java

The reverse() is quite tricky. We wrote this method in discussion 3. So this time I chose copy-past manner to fill out the requirement for this lab.

However, I still would like to show the codes for this method.

```
public static IntList reverse(IntList A) {
    if (A == null || A.rest == null) {
        return null;
    }
    IntList current = A;
    IntList previous = null;
    while (current != null) {
        IntList nextCurrent = current.rest;
        current.rest = previous;
        previous = current;
        current = nextCurrent;
    }
    A = previous;
    return A;
}
```

经过深思熟虑，克凡认为上述方法的返回值和输入值不相同，比如调用`IntList.reverse(B)`所得到的返回值和`B`的值不相同。

This function will destructively reverse the input. But it actually destory the input, not only reverse it. This point is rather tricky. This is because the statement `IntList current = A` Let `current` points to the same address, and destories `A` in the while loop. The statement `A = previous` cannot save our input, becauese this is actually a new `A` since previous is not as similar as input value. This issue is very similar to The Pokemon Problem. In this lab, the author found that the only way out is to use deep copy at the statement `IntList current = A` .But by doing so, the method will be non-destructive. Anyway, you can get full marks for this lab without all these hassle stuffs. The author just intends to demonstrate that the **GRoE** for classes in functions are far more complicated than we thought.

4. Getting Started on Project 2(Lab 5)

This lab mainly wants to introduce some basic conceptions and skills that will be useful in the following proj2. This lab teaches us how to use the Tile Rendering Engine and show two demos of worlds created by the tiles. For the original instructions, please refer Lab5.

The most tricky part of this lab is to self-design and develop a hexagon drawing method.

> Start by trying to create a method `addHexagon` that adds a hexagon of side length s to a given position in the world This is surprisingly tricky! There are many many ways of doing this, but however you do it, **the most important thing is to break the task down into smaller pieces.** This will probably involve creating helper methods. Example hexagons of size 2, 3, 4, and 5 are shown below. Note that your hexagon should always have a “middle” that is two rows thick, no matter how large the size (ie. the widest part consists of two rows of equal length). This is important so that the hexagons tesselate nicely.

The most useful strategy of solving this problem is to define the private helper method. These helper methods might involve drawing tasks, or they might involve calculating useful quantities.

In my solution, I wrote three helper methods.

First: `private static int hexRowWidth(int size, int i)` which is aim to calculate the width of ith row of a hexagon with specific size.

```
/**
     * compute the width of ith row of a size hexagon
     *
     * @param size the size of the hexagon
     * @param i    the row number of hexagon where i=0 is the
     *             bottom row
     * @return
     */
    private static int hexRowWidth(int size, int i) {
        if (size == 0) {
            return 0;
        }
        int returnValue;
        if (i >= size) {
            returnValue = size + 2 * (2 * size - i - 1);
        } else {
            returnValue = size + 2 * i;
        }
        return returnValue;
    }
```

Second: `private static Position findPosition(Position position, int i, int size)` which is aim to find the position of start point(left) of its row by using the given position(left bottom position of hexagon)

```
/**
     * find the position of start point(left) of one tile
     * by using the given position(left bottom position of hexagon
     *
     * @param position start position(left bottom position of hexagon
     * @param i        the ith line of a hexagon
     * @param size     the size of a hexagon
```

```
     * @return
     */
    private static Position findPosition(Position position, int i, int size) {
        Position p = new Position(position.x, position.y);
        if (i >= size) {
            p.x = position.x - (2 * size - i - 1);
            p.y = position.y + i;
        } else {
            p.x = position.x - i;
            p.y = position.y + i;
        }
        return p;
    }
```

Third: `private static void addRow(Position position, int width, TETile[][] world)` which is aim to add a row of a hexagon at given position.

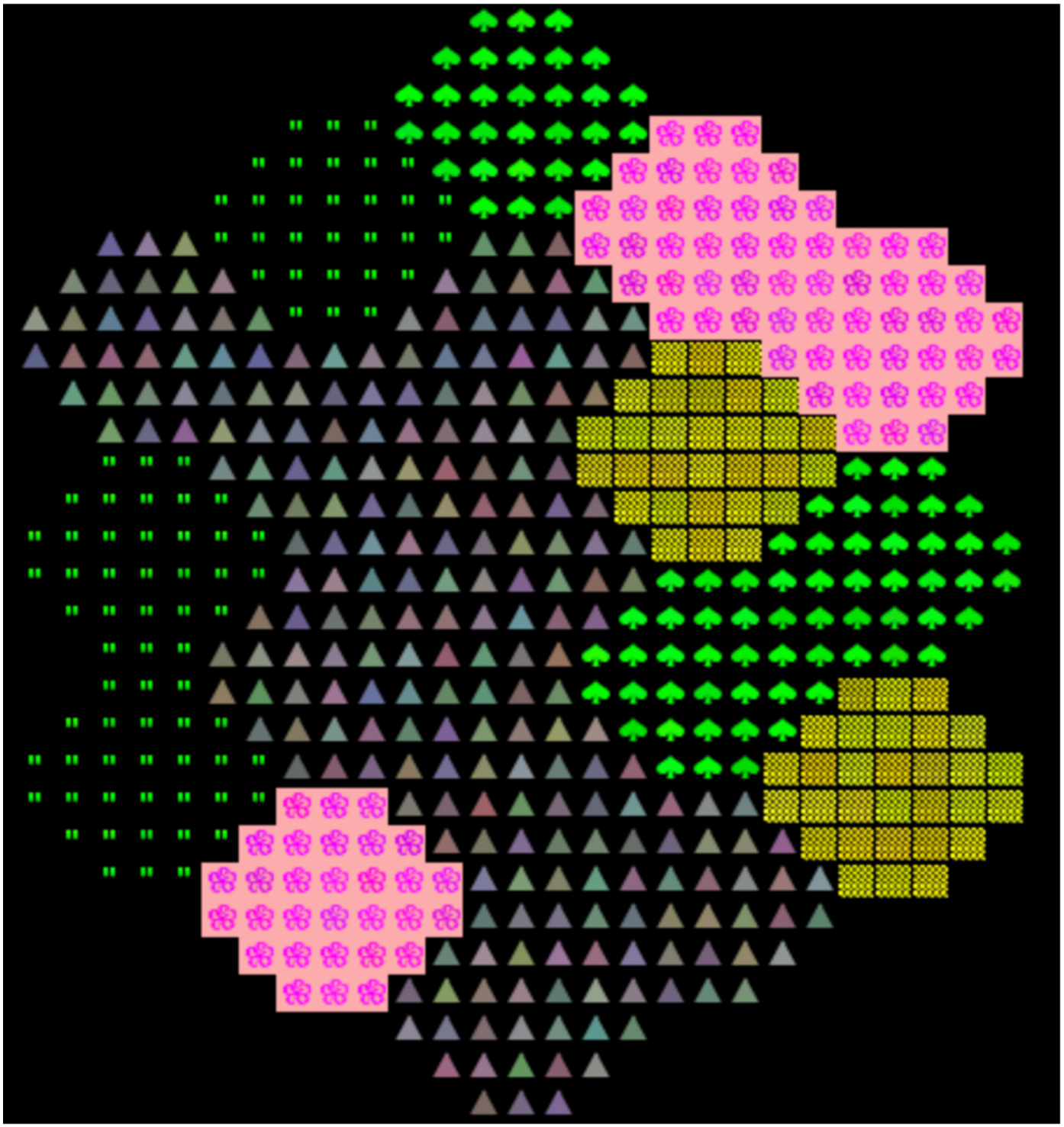
```
/**
     * add a row of a hexagon at given position
     *
     * @param position the start position(left) of the row of this hexagon
     * @param width    the width of this row
     * @param world    the given world in which we draw this row
     */
    private static void addRow(Position position, int width, TETile[][] world) {
        int boundary = world[0].length;
        if (width + position.x > boundary) {
            throw new IllegalArgumentException("Out of boundary");
        }
        for (int i = 0; i < width; i++) {
            world[position.x + i][position.y] = Tileset.FLOWER;
        }
    }
```

Final: `public static void addHexagon(int size, Position position, TETile[][] world)` which is aim to add a hexagon at given position.

```
    /**
     * add a hexagon at given position
     *
     * @param size     the size of this hexagon
     * @param position the position(left bottom) of this hexagon
     * @param world    the world in which we draw this heaxgon
     */
    public static void addHexagon(int size, Position position, TETile[][] world) {
        for (int i = 0; i < 2 * size; i++) {
            Position p = findPosition(position, i, size);
            addRow(p, hexRowWidth(size, i), world);
        }
    }
```

Please be noted that you should absolutely not try to do something like do everything in a nested for loop with no helper methods.

The remaining part of this lab is to arrange some hexagons in the pattern shown below, consisting of 19 total hexagons.



The key point to solve this problem is to define:

```
private static Position findTop(Position position, int size)
```

```
    /**
     * find the top hexagon's start point(left bottom)
     *
     * @param position the start point under this hexagon
     * @param size     the size of the hexagon
     * @return
     */
     private static Position findTop(Position position, int size) {
        Position p = new Position(position.x, position.y);
        p.y = p.y + 2 * size;
        return p;
    }
```

Which is use to find the top hexagon's start point(left bottom).

The remaining part is quite straightforward, however, the author defines:

```
/**
     * Draw three hexagons
     *
     * @param p     the start point of these three hexagons
     * @param size  the size of each hexagons
     * @param world the world in which we draw the hexagons
     */
    private static void threeHexagon(Position p, int size, TETile[][] world) {
        for (int i = 0; i < 3; i++) {
            addHexagonTree(size, p, world);
            p = findTop(p, size);
        }
    }

    /**
     * Draw three hexagons
     *
     * @param p     the start point of these three hexagons
     * @param size  the size of each hexagons
     * @param world the world in which we draw the hexagons
     */
    private static void fourHexagon(Position p, int size, TETile[][] world) {
        for (int i = 0; i < 4; i++) {
            addHexagonFlower(size, p, world);
            p = findTop(p, size);
        }
    }

    /**
     * Draw three hexagons
     *
     * @param p     the start point of these three hexagons
```

```
     * @param size  the size of each hexagons
     * @param world the world in which we draw the hexagons
     */
    private static void fiveHexagon(Position p, int size, TETile[][] world) {
        for (int i = 0; i < 5; i++) {
            addHexagonMountain(size, p, world);
            p = findTop(p, size);
        }
    }
```

to improve the overall efficiency of the whole program since we know the invariant that the number of hexagons we should draw at each different start point.

For the entire program, please refer DrawingHexagons.

5. Tree Maps vs. Hash Maps(Lab 9)

The original tasks and instructions could be found at Lab 9. In this lab, you'll create **BSTMap**, a BST-based implementation of the Map61B interface, which represents a basic map. Then, you'll create **MyHashMap**, another implementation of the Map61B interface, which instead represents a Hash Map rather than a Tree Map.

A. BSTMap

Kefan knows that the `remove(K key)`, `remove(K key, V value)` and `keySet()` are quite tricky to implement. So I decide to temporarily skip this part. The remaining part is straightforward, just refer the text book and lecture code, it is not hard to finish.

The code for BSTMap is:

```
import java.util.Iterator;
import java.util.Set;

/**
 * Implementation of interface Map61B with BST as core data structure.
 *
 * @author Your name here
 */
public class BSTMap<K extends Comparable<K>, V> implements Map61B<K, V> {

    private class Node {
        /* (K, V) pair stored in this Node. */
        private K key;
        private V value;

        /* Children of this Node. */
        private Node left;
        private Node right;
```

```
        private Node(K k, V v) {
            key = k;
            value = v;
        }
    }

    private Node root;  /* Root node of the tree. */
    private int size; /* The number of key-value pairs in the tree */

    /* Creates an empty BSTMap. */
    public BSTMap() {
        this.clear();
    }

    /* Removes all the mappings from this map. */
    @Override
    public void clear() {
        root = null;
        size = 0;
    }

    /**
     * Returns the value mapped to by KEY in the subtree rooted in P.
     * or null if this map contains no mapping for the key.
     */
    private V getHelper(K key, Node p) {
        if (p == null) {
            return null;
        }
        int cmp = key.compareTo(p.key);
        if (cmp < 0) {
            return getHelper(key, p.left);
        } else if (cmp > 0) {
            return getHelper(key, p.right);
        } else {
            return p.value;
        }
    }

    /**
     * Returns the value to which the specified key is mapped, or null if this
     * map contains no mapping for the key.
     */
    @Override
    public V get(K key) {
        return getHelper(key, root);
    }

    /**
```

```
     * Returns a BSTMap rooted in p with (KEY, VALUE) added as a key-value mapping.
     * Or if p is null, it returns a one node BSTMap containing (KEY, VALUE).
     */
    private Node putHelper(K key, V value, Node p) {
        if (p == null) {
            return new Node(key, value);
        }
        int cmp = key.compareTo(p.key);
        if (cmp < 0) {
            p.left = putHelper(key, value, p.left);
        } else if (cmp > 0) {
            p.right = putHelper(key, value, p.right);
        } else {
            p.value = value;
        }
        return p;
    }

    /**
     * Inserts the key KEY
     * If it is already present, updates value to be VALUE.
     */
    @Override
    public void put(K key, V value) {
        if (getHelper(key, root) != null) {
            root = putHelper(key, value, root);
        } else {
            root = putHelper(key, value, root);
            size += 1;
        }
    }

    /* Returns the number of key-value mappings in this map. */
    @Override
    public int size() {
        return size;
    }
}
```

B. MyHashMap

All the parts of MyHashMap are straightforward. Just one thing that is worth to mention: Unlike lecture (where each bucket was represented as a naked recursive linked list), each bucket in this lab is implemented as an `ArrayMap`.

The code for MyHashMap is:

```
import java.util.HashSet;
import java.util.Iterator;
```

```java
import java.util.Set;

/**
 *  A hash table-backed Map implementation. Provides amortized constant time
 *  access to elements via get(), remove(), and put() in the best case.
 *
 *  @author Your name here
 */
public class MyHashMap<K, V> implements Map61B<K, V> {

    private static final int DEFAULT_SIZE = 16;
    private static final double MAX_LF = 0.75;

    private ArrayMap<K, V>[] buckets;
    private int size;

    private int loadFactor() {
        return size / buckets.length;
    }

    public MyHashMap() {
        buckets = new ArrayMap[DEFAULT_SIZE];
        this.clear();
    }

    public MyHashMap(int initialSize) {
        buckets = new ArrayMap[initialSize];
        this.clear();
    }

    /* Removes all the mappings from this map. */
    @Override
    public void clear() {
        this.size = 0;
        for (int i = 0; i < this.buckets.length; i += 1) {
            this.buckets[i] = new ArrayMap<>();
        }
    }

    /** Computes the hash function of the given key. Consists of
     *  computing the hashcode, followed by modding by the number of buckets.
     *  To handle negative numbers properly, uses floorMod instead of %.
     */
    private int hash(K key) {
        if (key == null) {
            return 0;
        }

        int numBuckets = buckets.length;
```

```
        return Math.floorMod(key.hashCode(), numBuckets);
    }

    /* Returns the value to which the specified key is mapped, or null if this
     * map contains no mapping for the key.
     */
    @Override
    public V get(K key) {
        int hashCode = hash(key);
        return buckets[hashCode].get(key);
    }

    /* Private Helper method to resize the buckets */
    private void resize() {
        MyHashMap<K, V> resizedMap = new MyHashMap<>(buckets.length * 2);
        for (int i = 0; i < buckets.length; i++) {
            for (K key : buckets[i]) {
                V value = buckets[i].get(key);
                resizedMap.put(key, value);
            }
        }
        this.buckets = resizedMap.buckets;
    }

    /* Associates the specified value with the specified key in this map. */
    @Override
    public void put(K key, V value) {
        int hashCode = hash(key);
        if (get(key) != null) {
            buckets[hashCode].put(key, value);
        } else {
            buckets[hashCode].put(key, value);
            size += 1;
        }
        if (loadFactor() > MAX_LF) {
            resize();
        }
    }

    /* Returns the number of key-value mappings in this map. */
    @Override
    public int size() {
        return size;
    }

    ///////////////// EVERYTHING BELOW THIS LINE IS OPTIONAL ////////////////

    /* Returns a Set view of the keys contained in this map. */
    @Override
```

```java
    public Set<K> keySet() {
        Set<K> set = new HashSet<>();
        for (int i = 0; i < buckets.length; i++) {
            for (K key : buckets[i]) {
                set.add(key);
            }
        }
        return set;
    }

    /* Removes the mapping for the specified key from this map if exists.
     * Not required for this lab. If you don't implement this, throw an
     * UnsupportedOperationException. */
    @Override
    public V remove(K key) {
        if (get(key) == null) {
            return null;
        }
        int hashCode = hash(key);
        size -= 1;
        return buckets[hashCode].remove(key);
    }

    /* Removes the entry for the specified key only if it is currently mapped to
     * the specified value. Not required for this lab. If you don't implement this,
     * throw an UnsupportedOperationException.*/
    @Override
    public V remove(K key, V value) {
        if (get(key) == null || get(key) != value) {
            return null;
        }
        int hashCode = hash(key);
        size -= 1;
        return buckets[hashCode].remove(key);
    }

    @Override
    public Iterator<K> iterator() {
        return new MyHashMapIterator();
    }

    private class MyHashMapIterator implements Iterator<K> {
        private int wizPos;
        private Set<K> set;
        private K[] array = (K[]) new Object[size];

        public MyHashMapIterator() {
            this.wizPos = 0;
            set = keySet();
```

```
            array = (K[]) set.toArray();
        }

        @Override
        public boolean hasNext() {
            return (wizPos < size);
        }

        public K next() {
            K returnItem = array[wizPos];
            wizPos += 1;
            return returnItem;
        }
    }
}
```

6. Priority Queues(Lab 10)

The original questions and instructions could be found at the following link Lab10.

A. Representing a Tree With an Array

You've seen two approaches to implementing a sequence data structure: either using an array, or using linked nodes. Wth BSTs, we extended our idea of linked nodes to implement a tree data structure. As discussed in the heaps lecture, we can also use an array to represent a complete tree.

Here's how we implement a complete binary tree:

- The root of the tree will be in position 1 of the array (nothing is at position 0). We can define the position of every other node in the tree recursively:
- The left child of a node at position `n` is at position `2n`.
- The right child of a node at position `n` is at position `2n + 1`.
- The parent of a node at position `n` is at position `n/2`.

B. Working With Binary Heaps

Defination

In this lab, you will be making a priority queue using a binary min-heap (where smaller values correspond to higher priorities). Recall from lecture: Binary min-heaps are basically just binary trees (but *not* binary search trees) – they have all of the same invariants of binary trees, with two extra invariants:

- **Invariant 1:** the tree must be *complete* (more on this later)
- **Invariant 2:** every node is smaller than its descendants (there is another variation called a binary *max* heap where every node is greater than its descendants)

Invariant 2 guarantees that the min element will always be at the root of the tree. This helps us access that item quickly, which is what we need for a priority queue.

We need to make sure binary min-heap methods maintain the above two invariants. Here's how we do it:

Add an item

1. Put the item you're adding in the left-most open spot in the bottom level of the tree.
2. Swap the item you just added with its parent until it is larger than its parent, or until it is the new root. This is called *bubbling up* or *swimming*.

Remove the min item

1. Swap the item at the root with the item of the right-most leaf node.
2. Remove the right-most leaf node, which now contains the min item.
3. *Bubble down* the new root until it is smaller than both its children. If you reach a point where you can either bubble down through the left or right child, you must choose the smaller of the two. This process is also called *sinking*.

C. Complete Trees

There are a couple different notions of what it means for a tree to be well balanced. A binary heap must always be what is called *complete* (also sometimes called *maximally balanced*).

A **complete tree** has all available positions for nodes filled, except for possibly the last row, which must be filled from left-to-right.

D. Implementation

The implementation process is quite straightforward since I have already practiced it before lab. The only thing changed in this lab is we use the instance variable `contents` to store the `Node` object. The priority of each `Node` object is defined as the instance variable `myPriority` of `Node` object.

```
mport org.junit.Test;

import java.util.NoSuchElementException;

import static org.junit.Assert.*;

/**
 * A Generic heap class. Unlike Java's priority queue, this heap doesn't just
 * store Comparable objects. Instead, it can store any type of object
 * (represented by type T), along with a priority value. Why do it this way? It
 * will be useful later on in the class...
 */
public class ArrayHeap<T> implements ExtrinsicPQ<T> {
    private Node[] contents;
    private int size;

    public ArrayHeap() {
        contents = new ArrayHeap.Node[16];

        /* Add a dummy item at the front of the ArrayHeap so that the getLeft,
         * getRight, and parent methods are nicer. */
```

```
        contents[0] = null;

        /* Even though there is an empty spot at the front, we still consider
         * the size to be 0 since nothing has been inserted yet. */
        size = 0;
    }

    /**
     * Returns the index of the node to the left of the node at i.
     */
    private static int leftIndex(int i) {
        /* TODO: Your code here! */
        return 2 * i;
    }

    /**
     * Returns the index of the node to the right of the node at i.
     */
    private static int rightIndex(int i) {
        /* TODO: Your code here! */
        return 2 * i + 1;
    }

    /**
     * Returns the index of the node that is the parent of the node at i.
     */
    private static int parentIndex(int i) {
        /* TODO: Your code here! */
        return i / 2;
    }

    /**
     * Gets the node at the ith index, or returns null if the index is out of
     * bounds.
     */
    private Node getNode(int index) {
        if (!inBounds(index)) {
            return null;
        }
        return contents[index];
    }

    /**
     * Returns true if the index corresponds to a valid item. For example, if
     * we have 5 items, then the valid indices are 1, 2, 3, 4, 5. Index 0 is
     * invalid because we leave the 0th entry blank.
     */
    private boolean inBounds(int index) {
        if ((index > size) || (index < 1)) {
```

```
            return false;
        }
        return true;
    }

    /**
     * Swap the nodes at the two indices.
     */
    private void swap(int index1, int index2) {
        Node node1 = getNode(index1);
        Node node2 = getNode(index2);
        contents[index1] = node2;
        contents[index2] = node1;
    }


    /**
     * Returns the index of the node with smaller priority. Precondition: not
     * both nodes are null.
     */
    private int min(int index1, int index2) {
        Node node1 = getNode(index1);
        Node node2 = getNode(index2);
        if (node1 == null) {
            return index2;
        } else if (node2 == null) {
            return index1;
        } else if (node1.myPriority < node2.myPriority) {
            return index1;
        } else {
            return index2;
        }
    }

    /**
     * Bubbles up the node currently at the given index.
     */
    private void swim(int index) {
        // Throws an exception if index is invalid. DON'T CHANGE THIS LINE.
        validateSinkSwimArg(index);
        while (index > 1 && min(index, parentIndex(index)) == index) {
            swap(index, parentIndex(index));
            index = parentIndex(index);
        }
        return;
    }

    /**
     * Bubbles down the node currently at the given index.
```

```
     */
    private void sink(int index) {
        // Throws an exception if index is invalid. DON'T CHANGE THIS LINE.
        validateSinkSwimArg(index);
        while (2 * index <= size) {
            int j = 2 * index;
            if (j < size && min(j, j + 1) == j + 1) {
                j += 1;
            }
            if (min(index, j) == index) {
                break;
            }
            swap(index, j);
            index = j;
        }
        return;
    }

    /**
     * Inserts an item with the given priority value. This is enqueue, or offer.
     * To implement this method, add it to the end of the ArrayList, then swim it.
     */
    @Override
    public void insert(T item, double priority) {
        /* If the array is totally full, resize. */
        if (size + 1 == contents.length) {
            resize(contents.length * 2);
        }
        Node nodeToInsert = new Node(item, priority);
        size += 1;
        contents[size] = nodeToInsert;
        validateSinkSwimArg(size);
        swim(size);
    }

    /**
     * Returns the Node with the smallest priority value, but does not remove it
     * from the heap. To implement this, return the item in the 1st position of the
ArrayList.
     */
    @Override
    public T peek() {
        if (contents[1] == null) {
            return null;
        } else {
            return contents[1].item();
        }
    }
```

```java
/**
 * Returns the Node with the smallest priority value, and removes it from
 * the heap. This is dequeue, or poll. To implement this, swap the last
 * item from the heap into the root position, then sink the root. This is
 * equivalent to firing the president of the company, taking the last
 * person on the list on payroll, making them president, and then demoting
 * them repeatedly. Make sure to avoid loitering by nulling out the dead
 * item.
 */
@Override
public T removeMin() {
    if (size() == 0) {
        throw new NoSuchElementException("Priority queue under flow");
    }
    if (contents[1] == null) {
        return null;
    }
    T min = contents[1].item();
    validateSinkSwimArg(size);
    validateSinkSwimArg(1);
    swap(1, size);
    size -= 1;
    sink(1);
    contents[size + 1] = null;

    /*
    if ((size > 0) && (size == (contents.length) / 4)) {
        resize(contents.length / 2);
    } */
    return min;
}

/**
 * Returns the number of items in the PQ. This is one less than the size
 * of the backing ArrayList because we leave the 0th element empty. This
 * method has been implemented for you.
 */
@Override
public int size() {
    return size;
}

/**
 * Change the node in this heap with the given item to have the given
 * priority. You can assume the heap will not have two nodes with the same
 * item. Check item equality with .equals(), not ==. This is a challenging
 * bonus problem, but shouldn't be too hard if you really understand heaps
 * and think about the algorithm before you start to code.
 */
```

```
    @Override
    public void changePriority(T item, double priority) {
        int index = 0;
        for (int i = 1; i <= size; i++) {
            if (contents[i].item() == item) {
                contents[i].myPriority = priority;
                index = i;
                break;
            }
        }
        if (index == 0) {
            return;
        }
        int leftIndex = leftIndex(index);
        int rightIndex = rightIndex(index);
        int parent = parentIndex(index);
        if (min(index, leftIndex) == leftIndex ||
                min(index, rightIndex) == rightIndex) {
            sink(index);
        }
        if (min(index, parent) == index) {
            swim(index);
        }
    }

    /**
     * Prints out the heap sideways. Provided for you.
     */
    @Override
    public String toString() {
        return toStringHelper(1, "");
    }

    /* Recursive helper method for toString. */
    private String toStringHelper(int index, String soFar) {
        if (getNode(index) == null) {
            return "";
        } else {
            String toReturn = "";
            int rightChild = rightIndex(index);
            toReturn += toStringHelper(rightChild, "         " + soFar);
            if (getNode(rightChild) != null) {
                toReturn += soFar + "    /";
            }
            toReturn += "\n" + soFar + getNode(index) + "\n";
            int leftChild = leftIndex(index);
            if (getNode(leftChild) != null) {
                toReturn += soFar + "    \\";
            }
```

```
            toReturn += toStringHelper(leftChild, "        " + soFar);
            return toReturn;
        }
    }


    /**
     * Throws an exception if the index is invalid for sinking or swimming.
     */
    private void validateSinkSwimArg(int index) {
        if (index < 1) {
            throw new IllegalArgumentException("Cannot sink or swim nodes with index 0
or less");
        }
        if (index > size) {
            throw new IllegalArgumentException("Cannot sink or swim nodes with index
greater than current size.");
        }
        if (contents[index] == null) {
            throw new IllegalArgumentException("Cannot sink or swim a null node.");
        }
    }

    private class Node {
        private T myItem;
        private double myPriority;

        private Node(T item, double priority) {
            myItem = item;
            myPriority = priority;
        }

        public T item() {
            return myItem;
        }

        public double priority() {
            return myPriority;
        }

        @Override
        public String toString() {
            return myItem.toString() + ", " + myPriority;
        }
    }


    /**
     * Helper function to resize the backing array when necessary.
```

```
     */
    private void resize(int capacity) {
        Node[] temp = new ArrayHeap.Node[capacity];
        for (int i = 1; i < this.contents.length; i++) {
            temp[i] = this.contents[i];
        }
        this.contents = temp;
    }
}
```

Nothing tricky, just following the comments lines.

A special point that is worth to be mentioned is the Autograder shows:

```
est Failed!
    java.lang.IllegalArgumentException: Cannot sink or swim nodes with index greater
than current size.
    at ArrayHeap.validateSinkSwimArg:280 (ArrayHeap.java)
    at ArrayHeap.sink:123 (ArrayHeap.java)
    at ArrayHeap.removeMin:190 (ArrayHeap.java)
    at AGTestArrayHeap.testBetterPriority:52 (AGTestArrayHeap.java)
```

Because we use `validateSinkSwimArg(int index)`

```
  /**
     * Throws an exception if the index is invalid for sinking or swimming.
     */
    private void validateSinkSwimArg(int index) {
        if (index < 1) {
            throw new IllegalArgumentException("Cannot sink or swim nodes with index 0
or less");
        }
        if (index > size) {
            throw new IllegalArgumentException("Cannot sink or swim nodes with index
greater than current size.");
        }
        if (contents[index] == null) {
            throw new IllegalArgumentException("Cannot sink or swim a null node.");
        }
    }
```

So I discard the Autograder and the Gradescope gives me the fully scores.

7. Graphs(Lab 11)

The original questions and tips could be found at Lab 11.

A. Breadth First Search

> BFS and DFS are very similar. BFS uses a queue (First In First Out, a.k.a. FIFO) to store the fringe, whereas DFS uses a stack (First In Last Out, a.k.a. FILO). Naturally, programmers often use recursion for DFS, since we can take advantage of and use the implicit recursive call stack as our fringe. For BFS, there is no implicit data structures that we can use. We must instead use an explicit data structure, i.e. some sort of instance of a queue.

The BFS algorithm is similar as what I learn in the class. Just review the former code.

```
mport edu.princeton.cs.algs4.Queue;

/**
 * @author Josh Hug
 */
public class MazeBreadthFirstPaths extends MazeExplorer {
    /* Inherits public fields:
    public int[] distTo;
    public int[] edgeTo;
    public boolean[] marked;
    */
    private int s;
    private int t;
    private boolean targetFound = false;
    private Maze maze;

    public MazeBreadthFirstPaths(Maze m, int sourceX, int sourceY, int targetX, int
targetY) {
        super(m);
        // Add more variables here!
        maze = m;
        s = maze.xyTo1D(sourceX, sourceY);
        t = maze.xyTo1D(targetX, targetY);
        distTo[s] = 0;
        edgeTo[s] = s;
    }

    /**
     * Conducts a breadth first search of the maze starting at the source.
     */
    private void bfs(int v) {
        Queue<Integer> queue = new Queue<>();
        queue.enqueue(v);
        marked[v] = true;
        announce();
        while (!queue.isEmpty()) {
            int w = queue.dequeue();
            if (w == t) {
                targetFound = true;
```

```
            }
            if (targetFound) {
                return;
            }
            for (int i : maze.adj(w)) {
                if (!marked[i]) {
                    edgeTo[i] = w;
                    announce();
                    distTo[i] = distTo[w] + 1;
                    marked[i] = true;
                    announce();
                    queue.enqueue(i);
                }
            }
        }
    }

    @Override
    public void solve() {
        bfs(s);
    }
}
```

B. Depth First Search & Cycle Check

> In the world of graph theory, there exist many cycle detection algorithms. For example, a weighted-quick union object (without path compression) can be used to detect cycles in `O(E * logV)` time. For today's exercise, we will use DFS to detect cycles in this maze (an undirected graph) in `O(V + E)`. The idea is relatively simple: For every visited vertex `v`, if there is an adjacent `u` such that `u` is already visited and `u` is not parent of `v`, then there is a cycle in graph.
>
> For this part of the lab, you'll write a cycle detection algorithm. When you compile and run `CylesDemo`, you should see your algorithm crawl the graph. If it identifies any cycles, it should connect the vertices of the cycle using purple lines (by setting values in the `edgeTo` array and calling `announce()`) and terminating immediately. All visited vertices should marked, but there should be no edges connecting the part of the graph that doesn't contain a cycle. Instead, the only edges that should be drawn are the ones connecting the cycle.
>
> Recall from last section, you can use either recursion or a `Stack` class for DFS. If you decide to go with latter, we recommend using the Princeton standard libary's `Stack` class rather than `java.util.Stack`, which has some fundamental flaws. Alternaetly, you can instead use some linear structure in a FILO fashion.

This implementation is similar as the former Cycle check algorithm of directed graph.

```
public class MazeCycles extends MazeExplorer {
    /* Inherits public fields:
    public int[] distTo;
    public int[] edgeTo;
```

```
    public boolean[] marked;
    */
    public MazeCycles(Maze m) {
        super(m);
        distTo[0] = 0;
        edgeTo[0] = 0;
    }

    @Override
    public void solve() {
        // TODO: Your code here!
        dfs(0);
    }

    // Helper methods go here
    private void dfs(int v) {
        marked[v] = true;
        announce();
        for (int w : maze.adj(v)) {
            if (marked[w] && w != edgeTo[v]) {
                return;
            }
            if (!marked[w]) {
                edgeTo[w] = v;
                announce();
                distTo[w] = distTo[v] + 1;
                dfs(w);
            }
        }
    }
}
```

Please be noted that we need to make sure `!marked[w]` before we check next vertic using DFS.

C. A*

> Earlier, you created the `MazeBreathFirstPaths` class to find the shortest path from the source to the target.
>
> We've discussed other algorithms for finding shortest paths, specifically Dijkstra's algorithm and A*. For this lab, Dijkstra's algorithm would behave exactly the same as `MazeBreathFirstPaths` (B level question: figure out why), so it's not particularly interesting.
>
> However, A* does give us one way to potentially improve upon the performance of BFS.

```
import edu.princeton.cs.algs4.IndexMinPQ;

/**
 * @author Josh Hug
```

```
 */
public class MazeAStarPath extends MazeExplorer {
    private int s;
    private int t;
    private boolean targetFound = false;
    private Maze maze;
    private IndexMinPQ<Integer> pq;

    public MazeAStarPath(Maze m, int sourceX, int sourceY, int targetX, int targetY) {
        super(m);
        maze = m;
        s = maze.xyTo1D(sourceX, sourceY);
        t = maze.xyTo1D(targetX, targetY);
        distTo[s] = 0;
        edgeTo[s] = s;
        pq = new IndexMinPQ<>(maze.V());
    }

    /**
     * Estimate of the distance from v to the target.
     */
    private int h(int v) {
        return Math.abs(maze.toX(v) - maze.toX(t)) + Math.abs(maze.toY(v) -
maze.toY(t));
    }

    /**
     * Finds vertex estimated to be closest to target.
     */
    private int findMinimumUnmarked() {
        return -1;
        /* You do not have to use this method. */
    }

    /**
     * Performs an A star search from vertex s.
     */
    private void astar(int s) {
        // TODO
        pq.insert(s, h(s));
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            marked[v] = true;
            announce();
            if (v == t) {
                return;
            }
            for (int w : maze.adj(v)) {
                if (distTo[w] > distTo[v] + 1) {
```

```
                    distTo[w] = distTo[v] + 1;
                    edgeTo[w] = v;
                    if (pq.contains(w)) {
                        pq.changeKey(w, distTo[w] + h(w));
                    } else {
                        pq.insert(w, distTo[w] + h(w));
                    }
                }
            }
        }
    }

    @Override
    public void solve() {
        astar(s);
    }
}
```

Nothing tricky, just copying some codes from class materials.

Please be noted that we must check whether `distTo[w] > distTo[v] + 1` before changing the priority or insert a new node in `pq`.

8. Merge Sort and Quick Sort(Lab 12)

The original questions and tips could be found at Lab12.

> In this week's lab, you'll implement two of the sorting algorithms that we learned about in lecture this week. In lecture, we focused on sorting arrays. In this lab, you'll instead focus on sorting linked lists, which requires some cleverness and a good understanding of how merge sort and quick sort operate.
>
> All of the functions that you write will operate on the Princeton Queue Implementation, which implements a queue using a linked list. You should implement sorting using the public methods in the `Queue` class.

A. Merge Sort

> If you need to review how mergesort works, you may find the Merge sort demo from lecture or this Merge sort demo to be useful.
>
> To help you implement merge sort, start by implementing two helper methods:
>
> - Implement `makeSingleItemQueues`. This method takes in a `Queue` called `items`, and should return a `Queue` of `Queues` that each contain one item from `items`. For example, if you called `makeSingleItemQueues` on the `Queue` `("Alice" > "Vanessa" > "Ethan")`, it should return `(("Alice") > ("Vanessa") > ("Ethan"))`.
> - Implement `mergeSortedQueues`. This method takes two sorted queues `q1` and `q2` as parameters, and returns a new queue that has all of the items in `q1` and `q2` in sorted order. For example, `mergeSortedQueues(("Alice" > "Vanessa"), ("Ethan"))` should return `("Alice" > "Ethan" > "Vanessa")`. The provided `getMin` heper method may be helpful in implementing

> `mergeSortedQueues`. Your implementation should take time linear in the total number of items in `q1` and `q2` (it should be Θ(q1.size() + q2.size()).
>
> Once you've finished implementing these helper methods, use them to implement `mergeSort`. With the help of the two methods above, your `mergeSort` method should be short (fewer than 15 lines of code).

The implementation of Merge Sort is:

```
public class MergeSort {
    /**
     * Removes and returns the smallest item that is in q1 or q2.
     * <p>
     * The method assumes that both q1 and q2 are in sorted order, with the smallest
item first. At
     * most one of q1 or q2 can be empty (but both cannot be empty).
     *
     * @param q1 A Queue in sorted order from least to greatest.
     * @param q2 A Queue in sorted order from least to greatest.
     * @return The smallest item that is in q1 or q2.
     */
    private static <Item extends Comparable> Item getMin(
            Queue<Item> q1, Queue<Item> q2) {
        if (q1.isEmpty()) {
            return q2.dequeue();
        } else if (q2.isEmpty()) {
            return q1.dequeue();
        } else {
            // Peek at the minimum item in each queue (which will be at the front,
since the
            // queues are sorted) to determine which is smaller.
            Comparable q1Min = q1.peek();
            Comparable q2Min = q2.peek();
            if (q1Min.compareTo(q2Min) <= 0) {
                // Make sure to call dequeue, so that the minimum item gets removed.
                return q1.dequeue();
            } else {
                return q2.dequeue();
            }
        }
    }

    /**
     * Returns a queue of queues that each contain one item from items.
     */
    private static <Item extends Comparable> Queue<Queue<Item>>
        makeSingleItemQueues(Queue<Item> items) {
        // Your code here!
        if (items == null) {
```

```
            return null;
        }
        Queue<Queue<Item>> queue = new Queue<>();
        for (Item item : items) {
            Queue<Item> q = new Queue<>();
            q.enqueue(item);
            queue.enqueue(q);
        }
        return queue;
    }

    /**
     * Returns a new queue that contains the items in q1 and q2 in sorted order.
     * <p>
     * This method should take time linear in the total number of items in q1 and q2.
After
     * running this method, q1 and q2 will be empty, and all of their items will be in
the
     * returned queue.
     *
     * @param q1 A Queue in sorted order from least to greatest.
     * @param q2 A Queue in sorted order from least to greatest.
     * @return A Queue containing all of the q1 and q2 in sorted order, from least to
     * greatest.
     */
    private static <Item extends Comparable> Queue<Item> mergeSortedQueues(
            Queue<Item> q1, Queue<Item> q2) {
        // Your code here!
        Queue<Item> queue = new Queue<>();
        while (!q1.isEmpty() || !q2.isEmpty()) {
            queue.enqueue(getMin(q1, q2));
        }
        return queue;
    }

    /**
     * Returns a Queue that contains the given items sorted from least to greatest.
     */
    public static <Item extends Comparable> Queue<Item> mergeSort(
            Queue<Item> items) {
        // Your code here!
        if (items.size() == 1 || items.isEmpty()) {
            return items;
        }
        Queue<Queue<Item>> singleQueues = makeSingleItemQueues(items);
        while (true) {
            Queue<Item> p = singleQueues.dequeue();
            Queue<Item> q = singleQueues.dequeue();
            if (singleQueues.isEmpty()) {
```

```
                items = mergeSortedQueues(p, q);
                break;
            } else {
                singleQueues.enqueue(mergeSortedQueues(p, q));
            }
        }
        return items;
    }

    public static void main(String[] args) {
        Queue<Integer> q1 = new Queue<>();
        q1.enqueue(1);
        q1.enqueue(7);
        q1.enqueue(9);
        q1.enqueue(2);
        q1.enqueue(5);
        q1.enqueue(10);
        System.out.print(q1);
        System.out.println();
        Queue<Integer> q2 = mergeSort(q1);
        System.out.print(q2);
    }
}
```

Please be noted that we added a `main()` method in order to test our code that is called test driven development.

The most critical part in this implementation is we check `singleQueues.isEmpty()` then we decided whether we should assign `items = mergeSortedQueues(p, q);` or added the merged queue back to `singleQueues`.

B. Quick Sort

> If you need to review how quick sort works, take a look at slides 6 through 10 from this lecture. You'll be using the 3-way merge partitioning process described on slide 10. This partitioning approach, unfortunately, has no Hungarian dance demo (the dancers chose to partition based on the first item in the array, rather than on a random element.).
>
> Begin by implementing the helper function `partition()`. The `partition()` method takes an unsorted queue called `unsorted` and an item to pivot on, and three empty queues called `less`, `equal`, and `greater`. When it returns, `less` should contain all items from `unsorted` that were less than the pivot, `equal` should contain all items from `unsorted` that were equal to the pivot, and `greater` should contain all items that were greater than the pivot.
>
> Once you've implemented `partition()`, use it to implement the `quickSort` function. You may fund the `getRandomItem()` and `catenate()` methods that we've provided to be useful. Using these helper functions, your `quickSort` method should be short (fewer than 15 lines of code).

The implementation of Quick Sort is:

```
public class QuickSort {
    /**
     * Returns a new queue that contains the given queues catenated together.
     * <p>
     * The items in q2 will be catenated after all of the items in q1.
     */
    private static <Item extends Comparable> Queue<Item> catenate(Queue<Item> q1,
Queue<Item> q2) {
        Queue<Item> catenated = new Queue<Item>();
        for (Item item : q1) {
            catenated.enqueue(item);
        }
        for (Item item : q2) {
            catenated.enqueue(item);
        }
        return catenated;
    }

    /**
     * Returns a random item from the given queue.
     */
    private static <Item extends Comparable> Item getRandomItem(Queue<Item> items) {
        int pivotIndex = (int) (Math.random() * items.size());
        Item pivot = null;
        // Walk through the queue to find the item at the given index.
        for (Item item : items) {
            if (pivotIndex == 0) {
                pivot = item;
                break;
            }
            pivotIndex--;
        }
        return pivot;
    }

    /**
     * Partitions the given unsorted queue by pivoting on the given item.
     *
     * @param unsorted A Queue of unsorted items
     * @param pivot    The item to pivot on
     * @param less     An empty Queue. When the function completes, this queue will
contain
     *                 all of the items in unsorted that are less than the given pivot.
     * @param equal    An empty Queue. When the function completes, this queue will
contain
     *                 all of the items in unsorted that are equal to the given pivot.
     * @param greater  An empty Queue. When the function completes, this queue will
contain
```

```
     *                all of the items in unsorted that are greater than the given
pivot.
     */
    private static <Item extends Comparable> void partition(
            Queue<Item> unsorted, Item pivot,
            Queue<Item> less, Queue<Item> equal, Queue<Item> greater) {
        // Your code here!
        for (Item item : unsorted) {
            if (item.compareTo(pivot) > 0) {
                greater.enqueue(item);
            } else if (item.compareTo(pivot) == 0) {
                equal.enqueue(item);
            } else {
                less.enqueue(item);
            }
        }
    }

    /**
     * Returns a Queue that contains the given items sorted from least to greatest.
     */
    public static <Item extends Comparable> Queue<Item> quickSort(
            Queue<Item> items) {
        // Your code here!
        if (items.size() == 1 || items.size() == 0) {
            return items;
        }
        Queue<Item> less = new Queue<>();
        Queue<Item> equal = new Queue<>();
        Queue<Item> greater = new Queue<>();
        Item pivot = getRandomItem(items);
        partition(items, pivot, less, equal, greater);
        less = quickSort(less);
        greater = quickSort(greater);
        items = catenate(less, equal);
        items = catenate(items, greater);
        return items;
    }

    public static void main(String[] args) {
        Queue<Integer> q1 = new Queue<>();
        q1.enqueue(1);
        q1.enqueue(7);
        q1.enqueue(9);
        q1.enqueue(2);
        q1.enqueue(5);
        q1.enqueue(10);
        System.out.print(q1);
        System.out.println();
```

```
        Queue<Integer> q2 = quickSort(q1);
        System.out.print(q2);
    }
}
```

We used recursive calls to less and greater queues then catenate them together to format the final queue.

9. Radix Sorts(Lab 13)

The original questions and tips could be found at Lab 13.

A. Counting Sort

The counting sort uses `counts[]` to store the total number of each element in targeting array. The mission of this lab is to implement the `betterCountingSort` method so that it still does a counting based sort, but also handles negative numbers gracefully.

The implementation of `betterCountingSort` is:

```
public class CountingSort {
    /**
     * Counting sort on the given int array. Returns a sorted version of the array.
     * Does not touch original array (non-destructive method).
     * DISCLAIMER: this method does not always work, find a case where it fails
     *
     * @param arr int array that will be sorted
     * @return the sorted array
     */
    public static int[] naiveCountingSort(int[] arr) {
        // find max
        int max = Integer.MIN_VALUE;
        for (int i : arr) {
            max = max > i ? max : i;
        }

        // gather all the counts for each value
        int[] counts = new int[max + 1];
        for (int i : arr) {
            counts[i]++;
        }

        // when we're dealing with ints, we can just put each value
        // count number of times into the new array
        int[] sorted = new int[arr.length];
        int k = 0;
        for (int i = 0; i < counts.length; i += 1) {
            for (int j = 0; j < counts[i]; j += 1, k += 1) {
                sorted[k] = i;
            }
```

```
        }

        // however, below is a more proper, generalized implementation of
        // counting sort that uses start position calculation
        int[] starts = new int[max + 1];
        int pos = 0;
        for (int i = 0; i < starts.length; i += 1) {
            starts[i] = pos;
            pos += counts[i];
        }

        int[] sorted2 = new int[arr.length];
        for (int i = 0; i < arr.length; i += 1) {
            int item = arr[i];
            int place = starts[item];
            sorted2[place] = item;
            starts[item] += 1;
        }

        // return the sorted array
        return sorted;
    }

    /**
     * Counting sort on the given int array, must work even with negative numbers.
     * Note, this code does not need to work for ranges of numbers greater
     * than 2 billion.
     * Does not touch original array (non-destructive method).
     *
     * @param arr int array that will be sorted
     */
    public static int[] betterCountingSort(int[] arr) {
        // TODO make counting sort work with arrays containing negative numbers.
        int max = Integer.MIN_VALUE;
        for (int i : arr) {
            if (i > max) {
                max = i;
            }
        }

        int min = Integer.MAX_VALUE;
        for (int i : arr) {
            if (i < min) {
                min = i;
            }
        }
        if (min >= 0) {
            return naiveCountingSort(arr);
        }
```

```
        int move = -min;
        int[] counts = new int[move + max + 1];
        for (int i : arr) {
            counts[i + move] += 1;
        }

        int[] starts = new int[move + max + 1];
        int pos = 0;
        for (int i = 0; i < starts.length; i++) {
            starts[i] = pos;
            pos += counts[i];
        }

        int[] sorted = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
            int item = arr[i];
            int place = starts[item + move];
            sorted[place] = item;
            starts[item + move] += 1;
        }
        return sorted;
    }
}
```

The key philosophy is to add the `-min` in the array to make all the elements in targeting arrary could have a proper index in `counts`. After that, we could construct `starts` to store the starting postion of each element and put the right order of elements into `sorted` array.

B. Radix Sort

> The *radix* of a numeral system is the number of values a single digit can take on. Binary numbers form a radix-2 system since each bit in a number can either be 0 or 1; decimal numbers are radix-10 since each digit in a number can take on values between 0 and 9. Words formed from the lowercase English alphabet are part of a radix-26 number system (yes, words can be enumerated).

> Given a collection of elements all from the same radix numeral system, you can sort it using a radix sort. A radix sort examines elements in passes, one pass for each place in the elements. In other words, a radix sort would make a pass for the rightmost digit, one for the next-to-rightmost digit, and so on. In contrast, comparison based sorts (of which all the sorts you've learned already belong to), compare elements in pairs in order to sort.
>
> A key realization is the following: given two three-digit numbers (say, 536 and 139), it is possible to sort these numbers using a subroutine that sorts one digits-place at a time. There are two ways to do this:
>
> - First use the subroutine to sort everything on the least important (right-most) digit. Then sort everything on the next digit to the left. Continue, until you reach the left-most digit. This strategy requires the subroutine sorts to be stable.
> - First sort everything on the most-important (left-most) place and group all the items into buckets according to the value they take on in that digit. Recursively sort each bucket on the next highest

> digit. After your buckets have been sorted, concatenate your buckets back together.

> Here's an example of using the first strategy. Imagine we have the following numbers we wish to sort:
>
> 356, 112, 904, 294, 209, 820, 394, 810
>
> First we sort them by the first digit:
>
> 82**0**, 81**0**, 11**2**, 90**4**, 29**4**, 39**4**, 35**6**, 20**9**
>
> Then we sort them by the second digit, keeping numbers with the same second digit in the same relative order from the previous step:
>
> 9**0**4, 2**0**9, 8**1**0, 1**1**2, 8**2**0, 3**5**6, 2**9**4, 3**9**4
>
> Finally, we sort by the third digit, keeping numbers with the same third digit in their order from the previous step:
>
> **1**12, **2**09, **2**94, **3**56, **3**94, **8**10, **8**20, **9**04

> Hopefully it's not hard to see how these can be extended to more than three digits. The first strategy is known as *LSD radix sort*, and the second strategy is called *MSD radix sort*. LSD and MSD stand for *least significant digit* and *most significant digit* respectively, reflecting the order in which the digits are considered.

In this part of lab you'll write an implementation of radix sort for ASCII Strings. Normally, if we just had decimal numbers, we would say that we would have a radix of 10 (R = 10) since there are 10 possible digits at each index, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. ASCII Strings have 256 possible characters (numbered 0-255 with the radix R = 256) and are of variable length. In Java, you can get the ASCII value for a character by casting the `char` to an `int` (`int i = (int)'a'`), and get the character from the ASCII value by casting the other way (`char a = (char)97`).

Write the method `sort` in `RadixSort.java` that returns a sorted copy of the input list of ASCII Strings. Make sure the method is NON-destructive (the original list cannot be modified).

Please be noted that **MSD Radix** is conceptually simpler, but a bit uglier and more difficult to code (and slower in practice). LSD Radix is relatively harder to understand intuitively, but easier to implement (and faster in practice).

> Note: A difference between sorting numbers and strings using a radix sort is that we naturally sort Strings lexicographically (a.k.a. dictionary order; "2" is after "100" so "2" is considered equivalently as "2_", *where "*" is a placeholder that comes before any other character), but sort numbers in numerically ascending order (2 is before 100 and is considered equivalently as 002). In other words, to sort Strings, we would pad them on the right with empty values, but to sort numbers, we would pad them on the left with empty values.

Please be noted that the ASCII code for a is 97, not 0.

The implementation of LSD is:

```
public class RadixSort {
    /**
     * Does LSD radix sort on the passed in array with the following restrictions:
     * The array can only have ASCII Strings (sequence of 1 byte characters)
```

```
     * The sorting is stable and non-destructive
     * The Strings can be variable length (all Strings are not constrained to 1 length)
     *
     * @param asciis String[] that needs to be sorted
     * @return String[] the sorted array
     */
    public static String[] sort(String[] asciis) {
        // TODO: Implement LSD Sort
        int R = 256;
        int N = asciis.length;

        String[] arrange = new String[N];
        String[] sorted = new String[N];

        int length = findLongestLength(asciis);
        for (int i = 0; i < N; i++) {
            if (asciis[i].length() < length) {
                int j = asciis[i].length();
                while (j < length) {
                    asciis[i] += "0";
                    j += 1;
                }
            }
            arrange[i] = asciis[i];
        }


        for (int d = length - 1; d >= 0; d--) {
            int[] counts = new int[R + 1];

            for (int i = 0; i < N; i++) {
                counts[arrange[i].charAt(d) + 1] += 1;
            }

            for (int r = 0; r < R; r++) {
                counts[r + 1] += counts[r];
            }

            for (int i = 0; i < N; i++) {
                sorted[counts[arrange[i].charAt(d)]++] = arrange[i];
            }

            for (int i = 0; i < N; i++) {
                arrange[i] = sorted[i];
            }
        }

        for (int i = 0; i < arrange.length; i++) {
            int pos = 0;
```

```
            if (arrange[i].contains("0")) {
                for (int j = 0; j < arrange[i].length(); j++) {
                    if (arrange[i].charAt(j) == '0') {
                        pos = j;
                        break;
                    }
                }
                arrange[i] = arrange[i].substring(0, pos);
            }
        }
        return arrange;
    }

    private static int findLongestLength(String[] str) {
        int length = 0;
        for (String s : str) {
            if (s.length() > length) {
                length = s.length();
            }
        }
        return length;
    }
}
```

Writing test when you finished a part of your programming is a very good habit for developer, since you could find the mistakes immediately!

Nothing tricky, just following what we have learned in *Algorithm* book. However, there are two more steps in our implementation. The first step is to make all the String in `` `asciis `` with same length by adding `"0"` in the end of the strings whose length is smaller than the longest length in `asciis`. The second step is to clean the sorted string which has `"0"`.

10. Fractal Sound(Lab 14)

The original instructions and tips could be found at Lab 14.

A. Generating a SawTooth

The main part of this lab is to implement the wave of each period, we use `normalize` to get values between -1.0 and 1.0. The key philosophy is adding `state` with 1 until it reaches `period - 1` then reset state to 0.

```
public class SawToothGenerator implements Generator {
    private int period;
    private int state;

    public SawToothGenerator(int period) {
        this.state = 0;
        this.period = period;
    }
```

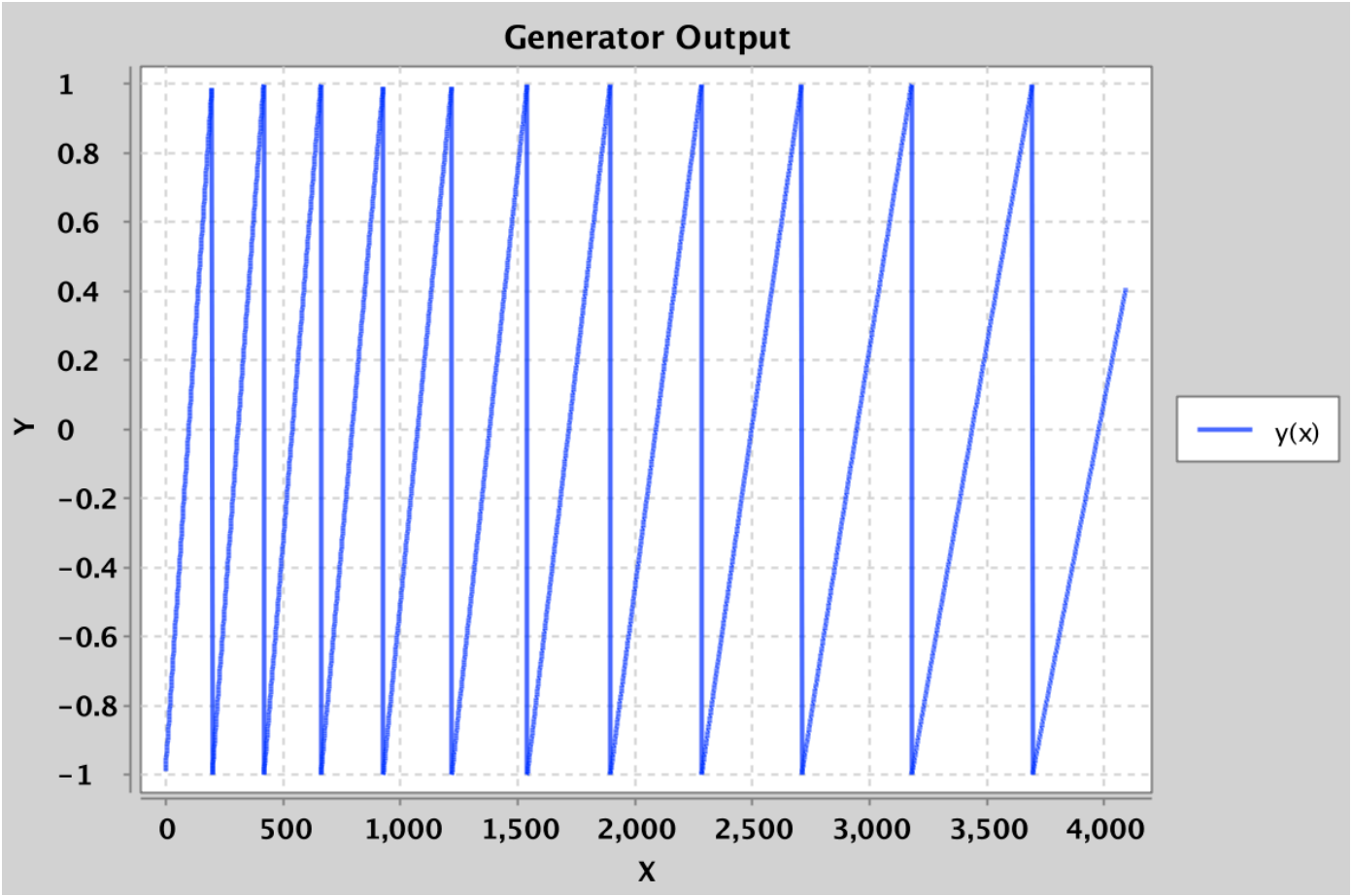
```
    @Override
    public double next() {
        if (state % period != 0) {
            double n = normalize(state);
            state += 1;
            return n;
        }
        state = 0;
        double n = normalize(state);
        state += 1;
        return n;
    }

    private double normalize(int state) {
        double num = (double) state / (period / 2);
        return num - 1;
    }
}
```

B. Generating an AcceleratingSawTooth

```
    Generator generator = new AcceleratingSawToothGenerator(200, 1.1);
    GeneratorAudioVisualizer gav = new GeneratorAudioVisualizer(generator);
    gav.drawAndPlay(4096, 1000000);
```

> What's more, we could create `AcceleratingSawToothGenerator`. This waveform should start at -1.0 and linearly increase towards 1.0, before resetting back to -1.0. After resetting, the period should change by a factor of the second argument, rounded down. The first argument to SawToothGenerator still describes the period of the waveform, i.e. the number of samples before it resets back down to -1.0. So, in the example above, the period of the second sawtooth should be 220 samples, the 3rd should be 242 samples, the 4th should 266 (which is 266.2 with the 0.2 truncated off).




```java
public class AcceleratingSawToothGenerator implements Generator {
    private int period;
    private int state;
    private double factor;

    public AcceleratingSawToothGenerator(int period, double factor) {
        this.period = period;
        this.state = 0;
        this.factor = factor;
    }

    @Override
    public double next() {
        if (state % period != 0) {
            double n = normalize(state);
            state += 1;
            return n;
        }
        if (state % period == 0 && state != 0) {
            state = 0;
            double n = normalize(state);
            state += 1;
            this.period = (int) Math.round(period * factor);
            return n;
        }
```

```
        double n = normalize(state);
        state += 1;
        return n;
    }

    private double normalize(int state) {
        double num = (double) state / (period / 2);
        return num - 1;
    }
}
```

Nothing tricky, copy pass most of codes of `SawToothGenerator`, just modifying the instance variable `period` to `period * factor`.

11. HugLife(Lab 15)

The original questions and tips could be found at Lab 15.

In this lab, we will implement `Plip` and `Clorus` classes, all of which extend `creature` class. We will decide how each `Plip` or Clorus object behave when the simulator call them.

The whole process is not tricky, just following the official examples and guides.

The codes for `Plip.java` is:

```
public class Plip extends Creature {

    /**
     * red color.
     */
    private int r;
    /**
     * green color.
     */
    private int g;
    /**
     * blue color.
     */
    private int b;

    /**
     * creates plip with energy equal to E.
     */
    public Plip(double e) {
        super("plip");
        r = 99;
        g = 0;
        b = 76;
        energy = e;
    }
```

```java
/**
 * creates a plip with energy equal to 1.
 */
public Plip() {
    this(1);
}

/**
 * Should return a color with red = 99, blue = 76, and green that varies
 * linearly based on the energy of the Plip. If the plip has zero energy,
 * it should have a green value of 63. If it has max energy, it should
 * have a green value of 255. The green value should vary with energy
 * linearly in between these two extremes. It's not absolutely vital
 * that you get this exactly correct.
 */
public Color color() {
    if (energy == 0) {
        g = 63;
        return color(r, g, b);
    } else if (energy == 2) {
        g = 255;
        return color(r, g, b);
    } else {
        g = linearGreen();
        return color(r, g, b);
    }
}

private int linearGreen() {
    double fraction = energy / 2;
    double green = fraction * 192 + 63;
    return (int) Math.round(green);
}

/**
 * Do nothing with C, Plips are pacifists.
 */
public void attack(Creature c) {
}

/**
 * Plips should lose 0.15 units of energy when moving. If you want
 * to avoid the magic number warning, you'll need to make a
 * private static final variable. This is not required for this lab.
 */
public void move() {
    energy -= 0.15;
}
```

```java
/**
 * Plips gain 0.2 energy when staying due to photosynthesis.
 */
public void stay() {
    if (energy + 0.2 >= 2) {
        energy = 2;
    } else {
        energy += 0.2;
    }
}

/**
 * Plips and their offspring each get 50% of the energy, with none
 * lost to the process. Now that's efficiency! Returns a baby
 * Plip.
 */
public Plip replicate() {
    Plip offSpring = new Plip(energy / 2);
    energy = energy / 2;
    return offSpring;
}

/**
 * Plips take exactly the following actions based on NEIGHBORS:
 * 1. If no empty adjacent spaces, STAY.
 * 2. Otherwise, if energy >= 1, REPLICATE.
 * 3. Otherwise, if any Cloruses, MOVE with 50% probability.
 * 4. Otherwise, if nothing else, STAY
 * <p>
 * Returns an object of type Action. See Action.java for the
 * scoop on how Actions work. See SampleCreature.chooseAction()
 * for an example to follow.
 */
public Action chooseAction(Map<Direction, Occupant> neighbors) {
    List<Direction> empties = getNeighborsOfType(neighbors, "empty");
    if (empties.size() == 0) {
        return new Action(Action.ActionType.STAY);
    }
    if (energy > 1) {
        Direction moveDir = HugLifeUtils.randomEntry(empties);
        return new Action(Action.ActionType.REPLICATE, moveDir);
    }
    List<Direction> clorus = getNeighborsOfType(neighbors, "clorus");
    if (clorus.size() > 0) {
        Direction moveDir = HugLifeUtils.randomEntry(empties);
        return new Action(Action.ActionType.MOVE, moveDir);
    }
```

```
        return new Action(Action.ActionType.STAY);
    }
}
```

The codes for `Clorus.java` is:

```
public class Clorus extends Creature {
    /**
     * red color.
     */
    private int r;
    /**
     * green color.
     */
    private int g;
    /**
     * blue color.
     */
    private int b;

    /**
     * creates clorus with energy equal to E.
     */
    public Clorus(double e) {
        super("clorus");
        r = 34;
        g = 0;
        b = 231;
        energy = e;
    }

    /**
     * creates a plip with energy equal to 1.
     */
    public Clorus() {
        this(1);
    }

    /**
     * this color method should always return color red = 34, green = 0, blue = 231
     *
     * @return color with property: red = 34, green = 0, blue = 231
     */
    @Override
    public Color color() {
        return new Color(r, g, b);
    }

    @Override
```

```java
    public void attack(Creature c) {
        energy += c.energy();
    }

    /**
     * Cloruses should lose 0.03 units of energy on a MOVE action.
     */
    @Override
    public void move() {
        energy -= 0.15;
    }

    /**
     * Cloruses should lose 0.01 units of energy on a STAY action.
     */
    @Override
    public void stay() {
        energy -= 0.01;
    }

    /**
     * When a Clorus replicates, it keeps 50% of its energy.
     * The other 50% goes to its offspring.
     * No energy is lost in the replication process.
     *
     * @return the offspring of clorus
     */
    @Override
    public Clorus replicate() {
        Clorus offSpring = new Clorus(energy / 2);
        energy = energy / 2;
        return offSpring;
    }

    /**
     * If there are no empty squares, the Clorus will STAY
     * (even if there are Plips nearby they could attack).
     * Otherwise, if any Plips are seen, the Clorus will ATTACK one of them randomly.
     * Otherwise, if the Clorus has energy greater than or equal to one,
     * it will REPLICATE to a random empty square.
     * Otherwise, the Clorus will MOVE to a random empty square.
     *
     * @param neighbors the neighbors information.
     * @return the action of called clorus should do.
     */
    @Override
    public Action chooseAction(Map<Direction, Occupant> neighbors) {
        List<Direction> empties = getNeighborsOfType(neighbors, "empty");
        if (empties.size() == 0) {
```

```
            return new Action(Action.ActionType.STAY);
        }
        List<Direction> plips = getNeighborsOfType(neighbors, "plip");
        if (plips.size() != 0) {
            Direction moveDir = HugLifeUtils.randomEntry(plips);
            return new Action(Action.ActionType.ATTACK, moveDir);
        }
        if (energy >= 1) {
            Direction moveDir = HugLifeUtils.randomEntry(empties);
            return new Action(Action.ActionType.REPLICATE, moveDir);
        }
        Direction moveDir = HugLifeUtils.randomEntry(empties);
        return new Action(Action.ActionType.MOVE, moveDir);
    }
}
```