# Data Structures in Java - Homework 7

## Problem 1

**5.1** Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \bmod 10$, show the resulting:

a. Separate chaining hash table.
b. Hash table using linear probing.
c. Hash table using quadratic probing.

**Answer:**

| (a) Index | | (b) Index | | (c) Index | |
|---|---|---|---|---|---|
| 0 | | 0 | 9679 | 0 | 9679 |
| 1 | 4371 | 1 | 4371 | 1 | 4371 |
| 2 | | 2 | 1989 | 2 | |
| 3 | 1323 → 6173 | 3 | 1323 | 3 | 1323 |
| 4 | 4344 | 4 | 6173 | 4 | 6173 |
| 5 | | 5 | 4344 | 5 | 4344 |
| 6 | | 6 | | 6 | |
| 7 | | 7 | | 7 | |
| 8 | | 8 | | 8 | 1989 |
| 9 | 4199 → 9679 → 1989 | 9 | 4199 | 9 | 4199 |

## Problem 2

**5.2** Show the result of rehashing the hash tables in Exercise 5.1.

**Use Table size of 23**

**Answer:**

Same for all (a), (b) and (c):

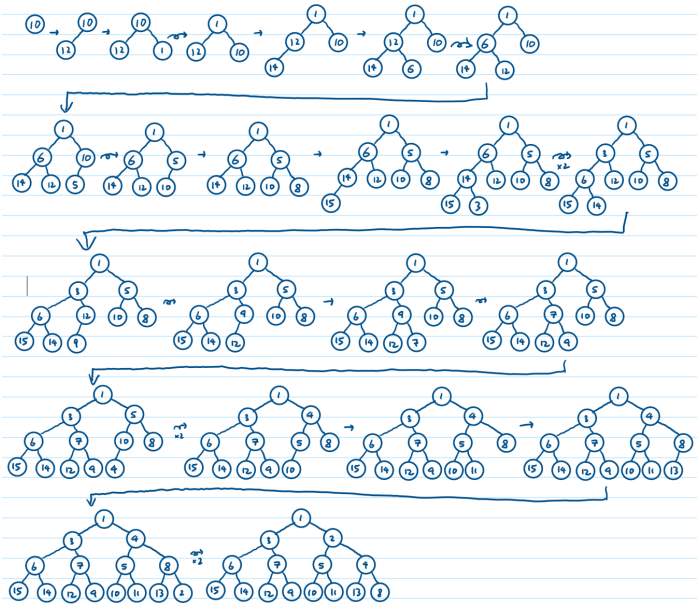
| Index | | | | | |
|---|---|---|---|---|---|
| 0 | | 8 | | 16 | |
| 1 | 4371 | 9 | 6173 | 17 | |
| 2 | | 10 | | 18 | |
| 3 | | 11 | 1989 | 19 | 9679 |
| 4 | | 12 | 1323 | 20 | 4344 |
| 5 | | 13 | 4199 | 21 | |
| 6 | | 14 | | 22 | |
| 7 | | 15 | | | |

## Problem 3

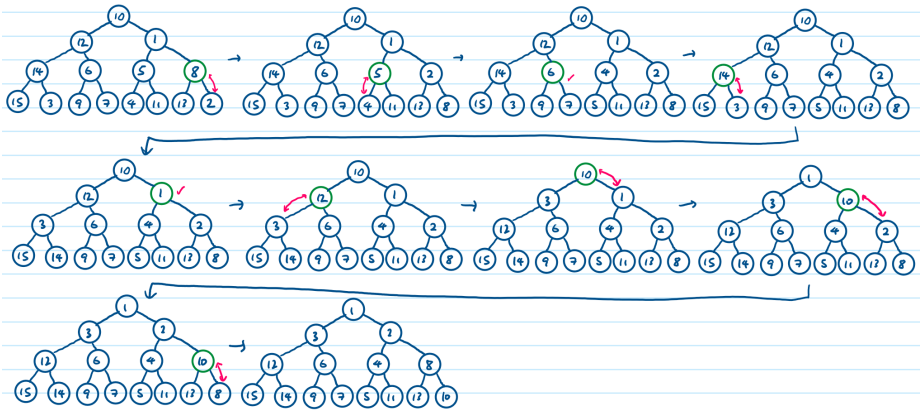
**6.2** a. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.

b. Show the result of using the linear-time algorithm to build a binary heap using the same input.

**a) Answer:**



**b) Answer:**



## Problem 4

**6.16** Suppose that binary heaps are represented using explicit links. Give a simple algorithm to find the tree node that is at implicit position $i$.

**Answer:**

Breadth-first traversal method and incrementing counter.

```
public BinaryNode get(int i)
{
    if (root == null || i < 1) return null;

    ArrayList<BinaryNode> queue = new ArrayList<BinaryNode>();
    queue.add(root); // access to root through class

    int currPos = 0;

    while (!queue.isEmpty()) {
        currPos++;
        BinaryNode currNode = queue.remove(0);

        if (currPos == i) {
            return currNode;
        }

        if (currNode.left != null) {
            queue.add(currNode.left);
        }
        if (currNode.right != null) {
            queue.add(currNode.right);
        }
    }

    return null;
}
```

## Problem 5

6.18 A **min-max heap** is a data structure that supports both `deleteMin` and `deleteMax` in $O(\log N)$ per operation. The structure is identical to a binary heap, but the heap-order property is that for any node, $X$, at even depth, the element stored at $X$ is smaller than the parent but larger than the grandparent (where this makes sense), and for any node $X$ at odd depth, the element stored at $X$ is larger than the parent but smaller than the grandparent. See Figure 6.57.

a. How do we find the minimum and maximum elements?

*b. Give an algorithm to insert a new node into the min-max heap.

### a) Answer:

Minimum is the root. Maximum is the larger child of the root.

### b) Answer:

Assuming heap is implemented in an array with extra space and the variable `size` is accessible through the class:

```
public void insert(Comparable item) {
    heap[size+1] = item; // assuming there is extra space
    size++;

    int idx = size;
    if (idx == 1) return;
    int parentIdx = idx/2;
    int depth = (int) (Math.log(idx) / Math.log(2));

    if (isEvenLevel(idx)) {
        if (item.compareTo(heap[parentIdx]) > 0)
        { // if item (even) is larger than parent (odd)
            swap(idx, parentIdx); // swap them
            percolateUpEven(parentIdx); // percolate the item up through odd levels
        } else { // otherwise
            percolateUpOdd(idx); // percolate up through even levels
        }
    } else { // if item is at odd level
        if (item.compareTo(heap[parentIdx]) < 0)
        { // if item (odd) is smaller than parent
            swap(idx, parentIdx); // swap them
            percolateUpEven(parentIdx); // percolate item up through even levels
        } else { // otherwise
            percolateUpOdd(idx); // percolate up through odd levels
        }
    }
}
```

## Problem 6

**7.1** Sort the sequence $3, 1, 4, 1, 5, 9, 2, 6, 5$ using insertion sort.

```
Original: [3, 1, 4, 1, 5, 9, 2, 6, 5]
Swap 1: [1, 3 | 4, 1, 5, 9, 2, 6, 5]
Swap 2: [1, 3, 4 | 1, 5, 9, 2, 6, 5]
Swap 3: [1, 1, 3, 4 | 5, 9, 2, 6, 5]
Swap 4: [1, 1, 3, 4, 5 | 9, 2, 6, 5]
Swap 5: [1, 1, 3, 4, 5, 9 | 2, 6, 5]
Swap 6: [1, 1, 2, 3, 4, 5, 9 | 6, 5]
Swap 7: [1, 1, 2, 3, 4, 5, 6, 9 | 5]
Swap 8: [1, 1, 2, 3, 4, 5, 5, 6, 9]
```