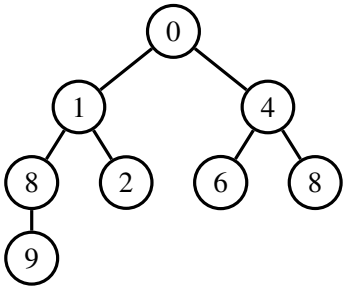
CS 61B Spring 2020 | Small Group Tutoring
Section 8: Hashing and Heaps | Worksheet 11

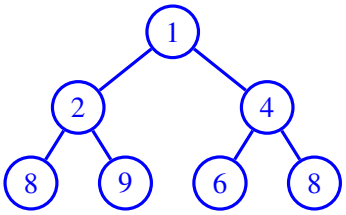
1 Basic Operations

1. **Min Heap**

   (a) Draw the Min Heap that results if we delete the smallest item from the heap.

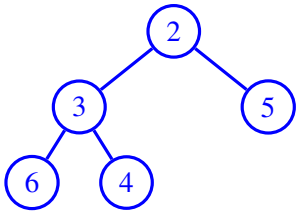


Solution:



   (b) Draw the Min Heap that results if we insert the elements 6, 5, 4, 3, 2 into an empty heap.

Solution:



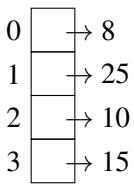
   (c) Given an array, heapify it such that after heapification it represents a Max Heap.

```
int[] a = {5, 12, 64, 1, 37, 90, 91, 97}
```

   After heapifying the array to represent a Max Heap, the array's elements will be ordered as follows: $\{97, 37, 91, 12, 5, 90, 64, 1\}$.

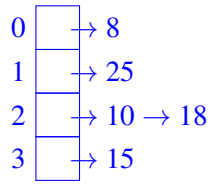
2. **External Chaining**

   Consider the following External Chaining Hash Set below, which resizes when the load factor reaches 1.5. Assume that were using the default hashCode for integers, which simply returns the integer itself.



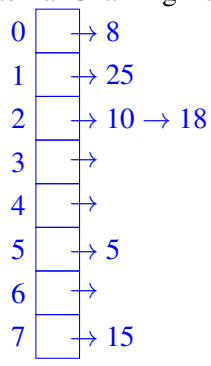
(a) Draw the External Chaining Hash Set that results if we insert 18.

Solution:



(b) Draw the External Chaining Hash Set that results if we insert 5 after the insertion done in part (a).

Solution:



2 Invalid Hashing

Which of the hashCodes are invalid? Assume we are trying to hash the following class:

```
import java.util.Random;
class Point {
private int x;
private int y;
private static count = 0;
public Point(int x, int y) {
    this.x = x;
    this.y = y;
    count += 1;
} }
```

(a) public void hashCode() { System.out.print(this.x + this.y); }

Solution: Invalid. The return type of the hashCode function should be an integer.

(b) public int hashCode() {
Random randomGenerator = new Random();
return randomGenerator.nextInt(Int); }

Solution: Invalid. Using a random generator results in the hashcode function to be not deterministic. There is a possibility that the hashcode for the same Point object could be different across two attempts to hash the object.

(c) public int hashCode() { return this.x + this.y; }

Solution: Valid. This is a good and safe hashcode function because the fields this.x and this.y are private variables. However, if these fields were not private, then any changes made the variables would result in an entirely different hashcode function. For instance, assume that the variables x and y are public. First, hash a Point object and then change its variables (x and y). Now, it will be impossible to find the initial Point object because the hashcode function will not return the same hash value.

(d) public int hashCode() { return count; }

Solution: Invalid. Since count is a static variable (it is shared by all instances of Point), the same object will not return the same hash code if another Point instance is created between calls.

(e) public int hashCode() { return 4; }

Solution: Valid. However, this is a bad hashcode function because all Point objects will be put in to the same bucket.

3 Search Structures Runtime

Assume you have N items. Using Theta notation, find the worst case runtime of each function.

Function	Unordered List	Sorted Array	Bushy Search Tree	"Good" Hash Table	Heap
find	Θ(N)	$\Theta(\log N)$	$\Theta(\log N)$	$\Theta(1)$	$\Theta(N)$
add (Amortized)	$\Theta(1)$	$\Theta(N)$	$\Theta(\log N)$	$\Theta(1)$	$\Theta(\log N)$
find largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(1)$
remove largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(\log N)$

4 Range Query

Write a function that counts the number of elements greater than a given number in a binary search tree. Assume the BST node has methods `label()`, `left()`, and `right()`.

```
static int findGreater(BST<Integer> T, int x) {
    if (T == null) {
        return 0
    }

    if (T.label() > x) {
        return 1 + findGreater(T.right(), x) + findGreater(T.left(), x)
    } else {
        return findGreater(T.right(), x);
    }
}
```

5 Bits (Midterm Review)

Fill in the following function that returns true if the input, an **unsigned** integer, is 1 more or 1 less than than a power of 2. Use bit operations. You may not declare booleans, nor may you assign static numerical values to int variables (i.e. int y = 5).

```
static boolean almostPowerOfTwo(int x) {
    /*  In order for the function to return true, the integer's bit
        representation must be of the form 00...10...01 or 00...11...11;
        two 1s bookending a contiguous bitstring of either 1s or 0s.
    */
    if ((x & 1) == 0) {
        // This is neither 1 more or 1 less than a power of 2.
        return false;
    }
    // Chop off the first bit.
    x = x >>> 1;
    int less = x & 1;
    if (less == 1) {
        /*  The power of 2 + 1 case is no longer possible (except for 3,
            which is both 1 greater than and 1 less than a power of 2).
        /*
        while (x > 0) {
            /*  x should be of the format 00...11...11;
                if x > 0 and the current bit of x is 0, then this is false.
            */
            if ((x & 1) == 0) {
                return false;
            }
            x = x >>> 1;
        }
    } else {
        // The power of 2 - 1 case is no longer possible.
        int seenOne;
        while (x > 0) {
            seenOne = (x & 1) | seenOne;
            x = x >>> 1;
            // We should only see one more 1 bit after removing the first one.
            if (seenOne == 1 && ((x & 1) == 1)) {
                return false;
            }
        }
    }
    return true;
}
```

6 Asymptotics (Midterm Review)

```
O(n^3)_ private static void f(int n) {
             Random generator = new Random();
             Hashset<Integer> hash = new Hashset<Integer>();
             for(int i = 0; i < n; i++) {
                if(generator.nextBoolean()){
                   hash.add(n-i);
                   }
             }
             if(hash.contains(5)){
                f(n-1);
             }
         }

_O(n ^ (1/2))_ private static void f1(int n) {
            int a = 0;
            int b = 0;
            while (b < n) {
               a += 1;
               b = b + a;
            }
         }
We know that each iteration of the for loop takes constant time, so
we need to figure out how many iterations of the for loop occur; let this
be k. At every step, a increments by 1 so we are essentially accumulating
in b = 1 + 2 + 3 + ... + k until b > N. Therefore, we want to solve for
1 + 2 + 3 +... + k = N which approximately simplifies to k^2/N, which we
solve to obtain k = (2N)^1/2 =(2)^1/2(N) ^ 1/2. Our runtime is determined
by k which is therefore O(N^1/2).

_O(b^a)_ private static void f2(int a, int b) {
            if(a <= 0) return;
            for(int i = 0; i < b; i+=1){
              f2(a-1, b);
            }
         }
```