UC Berkeley – Computer Science
CS61B: Data Structures

Midterm #2, Spring 2016

This test has 10 questions worth a total of 60 points. The exam is closed book, except that you are allowed to use two pages (both front and back, for 4 total sides) as a written cheat sheet. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Write the statement out below, and sign once you're done with the exam. **Write the statement out below in the blank provided and sign. You may do this before the exam begins.**

"I have neither given nor received any assistance in the taking of this exam."

____

____

Signature: _______________________

	Points		Points
0	0.5	5	6
1	9	6	3.5
2	4	7	6
3	4	8	8
4	0	9	9
		10	10

Name:
SID:
Three-letter Login ID:
Login of Person to Left:
Login of Person to Right:
Exam Room:
Primary TA (if any):

Total	60

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.
- Not all information provided in a problem may be useful.
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs during the exam, we'll announce a fix. Unless we specifically give you the option, the correct answer is not 'does not compile.'
- The last problem is the "hard" one.

Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹__________________☺].
After exam: [☹__________________☺].

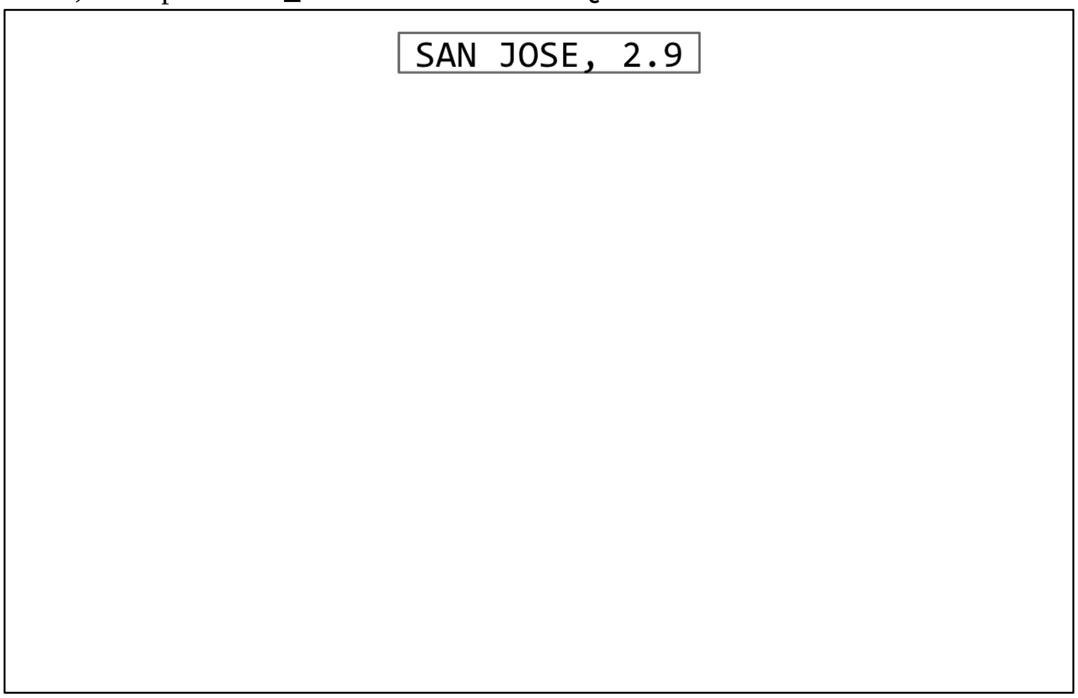
0. So It Begins II (0.5 points). Write your name and ID on the front page. Circle the exam room. Write the IDs of your neighbors. Write the given statement. Sign when you're done with the exam. Write your login in the corner of every page.

1. **BST and Hash Table Essentials (9 Points).**

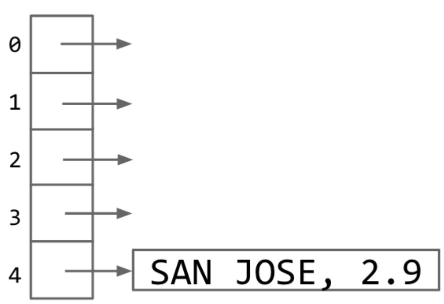
a) Suppose we're building a map that represents the rental cost in dollars per square foot of various locations. Starting from an initially empty BSTMap, if we call `put("SAN JOSE", 2.9)`, we'd get the tree shown in the box containing one node, with height equal to zero.

Draw the BSTMap after all of the following `put` operations have completed, in the order given. Assume that the tree is ordered based on alphabetical order. Draw your answer in the box to the right. This tree is not self-balancing. For reference, the alphabet is `_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789`.

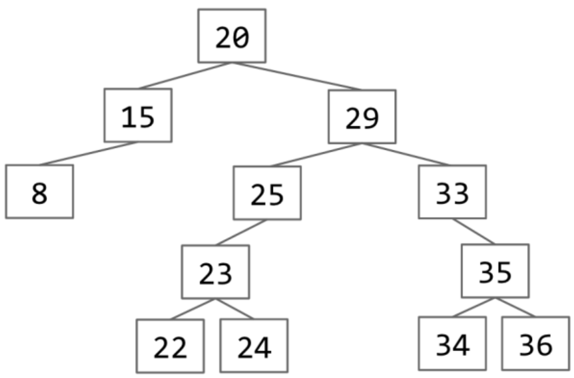
```
put("LA", 2.0)
put("NYC", 2.1)
put("SD", 6.3)
put("BOSTON", 2.3)
put("UNIT 3", 18.0)
put("SF", 2.7)
put("SD", 1.9)
```



b) Suppose we repeat the exercise from part a, but with a hash map. Assume the `hashCode` of our strings is equal to the number of the first letter of the string. For example, the `hashCode("SF") = 19`. For reference, B → 2, L→ 12, N → 14, S → 19, U → 21. Assume we have 5 buckets, and assume no resizing happens. Use the method we used in class (chaining).



c) Suppose we use a BST to represent a `TreeSet`. Suppose we call `remove(29)` on the `TreeSet` below. Draw a valid BST that results. You must use the deletion procedure from class (also known as Hibbard deletion). At most two references should change. Draw your tree in the space to the right.



d) Suppose we try to use a `HashMap` on a data type where the key's `hashCode` always returns 2000000000, and the value's `hashCode` always returns -5. Will the `HashMap's containsKey` and `get` methods always return the expected result (don't worry about runtime)? Assume that `equals` is properly implemented. State yes or no, and **briefly** explain your answer in the space below.

____

e) Suppose we try to create `HashSet<Glelk>` on the `Glelk` datatype described below. Will all operations on the `HashSet<Glelk>` behave as we expect? State yes or no, and **briefly** explain your answer in the space below.

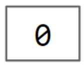
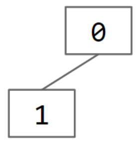
____

```
public class Glelk {
      private int x;
      private int y;

     /** Normally an equals method should check that o is actually a Glelk,
         as in HW3. However we have omitted this check for brevity. This
         will not affect the answer to part e.*/
      public boolean equals(Object o) {
            Glelk other = (Glelk) o;
            return (this.x == other.x) && (this.y == other.y);
      }
      public int hashCode() {
            return super.hashCode() / 2;
      }
}
```

2. WeightedQuickUnionUF (4 Points).

a) Draw a valid `WeightedQuickUnionUF` tree with worst case height, given sizes of N=1, N=2, N=4, N= 6, and N=8 in the boxes below, where N is the number of items in the `WeightedQuickUnionUF`. The first two are done for you. Recall that the height of a tree is the length of the longest path from the root to any leaf, so the height of the tree for N=2 is 1.

 N=1	 N=2	N=4
N=6	N=8	(draw anything)

b) Give the best case height and worst case height of a `WeightedQuickUnionUF` tree in Θ notation in terms of N, the number of items in the `WeightedQuickUnionUF`.

Best: _____________

Worst: _____________

3. Exceptions (4 Points).

Consider the code below, with print statements in **bold**. Recall that x / 2 rounds down to the nearest integer.

```
public static void checkIfZero(int x) throws Exception {
    if (x == 0) {
        throw new Exception("x was zero!");
    }
    System.out.println(x);
}

public static int mystery(int x) {
    int counter = 0;
    try {
        while (true) {
            x = x / 2;
            checkIfZero(x);
            counter += 1;
            System.out.println("counter is " + counter);
        }
    } catch (Exception e) {
        return counter;
    }
}

public static void main(String[] args) {
    System.out.println("mystery of 1 is " + mystery(1));
    System.out.println("mystery of 6 is " + mystery(6));
}
```

What will be the output when main is run? You may not need all lines.

____
____
____
____
____
____
____
____

4. (0 points). This religion, founded by J.R. “Bob” Dobbs, first made its public appearance in the 1979 pamphlet "The World Ends Tomorrow and You May Die".

5. Runtime in Context (6 Points).

a) Suppose we read a text file containing a list of city names and their cost of living, using the following code. Here `BSTMap` is the same as the implementation you created for lab8 with no special balancing features. Assume `isEmpty`, `readString`, and `readDouble` run in constant time. Assume that all `Strings` are of constant length. Assume throughout the problem that the input files are properly formatted and that no errors occur during execution. Assume all city names are unique.

```
public static Map<String, Double> readData(In in) {
   Map<String, Double> m = new BSTMap<String, Double>();
   while (!in.isEmpty()) {
      m.put(in.readString(), in.readDouble());
   }
   return m;
}
```

If there are `N` such cities in the file, what will be the runtime needed to complete execution of the `readData` function? Give your answer in Θ notation, for the best and worst case.

Best case: __Θ____________
Worst case: __Θ____________

b) Suppose that instead of a `BSTMap`, we use a **`HashMap`** like the one you implemented in `lab9` or `java.util.HashMap`. Give the best and worst case runtimes to complete execution of the `readData` method. The `String` class's `hashCode` method takes takes Θ(1) time (since our Strings are of constant length).

Best case: __Θ____________
Worst case: __Θ____________

c) Finally, suppose that instead of a `BSTMap`, we instead use a **`2-3-TreeMap`**. Give the best and worst case runtimes to complete execution of the `readData` method.

Best case: __Θ____________
Worst case: __Θ____________

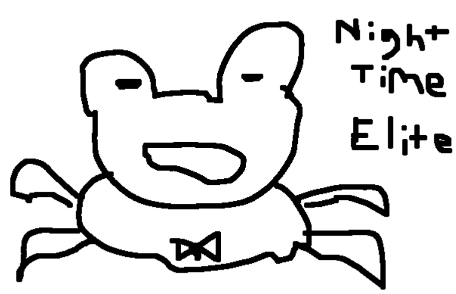
6. **Empirical Analysis (3.5 Points)**.

a) Suppose we write a program that takes one argument as input N. Suppose we use the Stopwatch class to measure the total running time R(N) of our program for various values of N, collecting the following data. **Approximate** the empirical run time in **tilde notation** as a function of N. Reminder from Asymptotics III: assume the formula is of the form ~ aN^b , and use only the largest data points. It is OK to round your exponent. It is OK to leave any constant factors in terms of a fraction. Do not leave your answer in terms of logarithms.

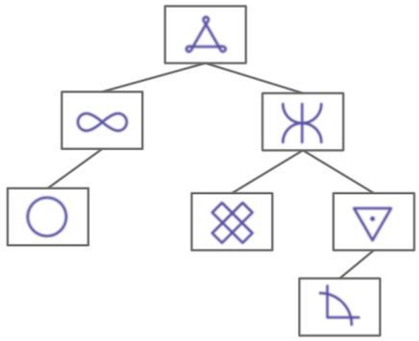
N	R(N)
62	17000
125	39000
250	75000
500	500000
1000	4000000

R(N) ~ _______________

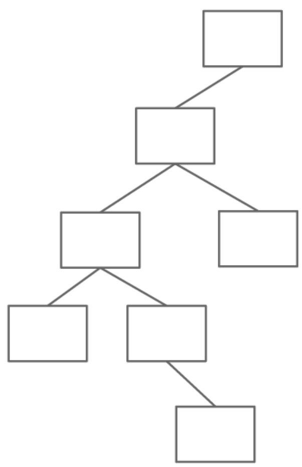
Designated Chillout Zone. Have a good time!!



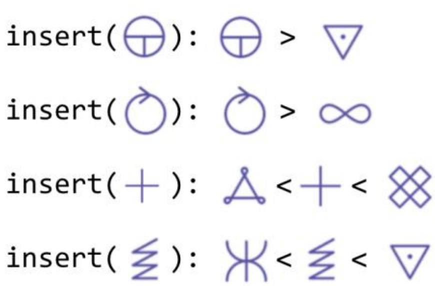
7. Leeway (6 Points). Consider the binary search tree below. Each symbol represents an object stored in the BST, e.g. ∞ might represent the string "josh", and  might represent the string "snowman".



a) Based on the ordering given by the tree above, fill in the tree below with valid symbols. Symbols must be unique. You may only use the 7 printed symbols (do not include any symbols from part b).



b) For each of the insertion operations below, use the information given to "insert" the element into the **TOP TREE WITH PRINTED SYMBOLS, <u>NOT THE TREE WITH YOUR HANDWRITTEN SYMBOLS</u>** by drawing the object (and any needed links) onto the tree. You can assume the objects are inserted in the order shown below. You should not change anything about the original tree; you should only add links and nodes for the new objects. If there is not enough information to determine where the object should be inserted into the tree, circle "not enough information". If there is enough information, circle "drawn in the tree above" and **draw in the tree AT THE TOP OF THE PAGE.**



Drawn In Tree Above   Not Enough Information

Drawn In Tree Above   Not Enough Information

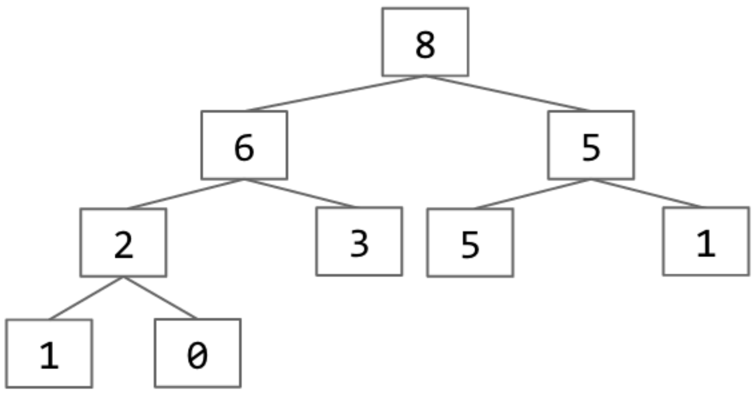
Drawn In Tree Above   Not Enough Information

Drawn In Tree Above   Not Enough Information

8. Balanced Trees (8 Points)

a) Suppose we have the max heap below, with array representation as shown. Show the heap after the maximum is deleted, using the procedure described in class. **Give your answer as an array.**

---	8	6	5	2	3	5	1	1	0			

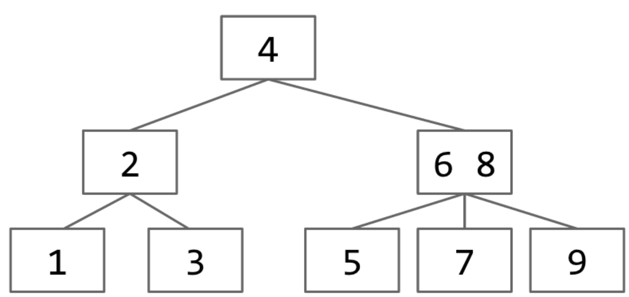


Your answer:

---												

b) Consider the 2-3 tree below. What order should we insert these numbers so that we get the tree shown? There may be multiple correct answers.

Answer:



c) Tumelo Barttrain suggests creating a special version of a heap where each item has 4 children to improve performance. Would such a heap have better, worse, or the same asymptotic performance in Θ notation as compared to a normal binary heap? Briefly explain your reasoning.

9. That Asymptotics Problem You Knew Was Coming (9 Points).

For each of the pieces of code below, give the runtime in $\Theta(\cdot)$ notation as a **function of N**. Your answer should be simple, with no unnecessary leading constants or unnecessary summations.

____
```
public static void p1(int N) {
     for (int i = 0; i < N; i += 1) {
          for (int j = 1; j < N; j = j + 2) {
               System.out.println("hi!");
          }
     }
}
```

____
```
public static void p2(int N) {
     for (int i = 0; i < N; i += 1) {
          for (int j = 1; j < N; j = j * 2) {
               System.out.println("hi!");
          }
     }
}
```

____
```
public static void p3(int N) {
     if (N <= 1) return;
     p3(N / 2);
     p3(N / 2);
}
```

____
```
public static void p4(int N) {
     int m = (int) ((15 + Math.round(3.2 / 2)) *
       (Math.floor(10 / 5.5) / 2.5) * Math.pow(2, 5));
     for (int i = 0; i < m; i++) {
          System.out.println("hi");
     }
}
```

____
```
public static void p5(int N) {
     for (int i = 1; i <= N * N; i *= 2) {
          for (int j = 0; j < i; j++) {
               System.out.println("moo");
          }
     }
}
```

10. Yum Yum Agar (10 Points). The Agar class is defined as follows:

```
public class Agar {
    public int size; /* assume this is always even and positive */
    public Agar(int s) { size = s; }
    public boolean equals(Object o) {
        Agar other = (Agar) o;
        return this.size == other.size;
    }
    public int hashCode() { /* Not shown, but assume it's consistent with
      equals, always positive, and does not change unless size changes. */
    }
}
```

Usually, `Agars` are simple, inert creatures, and coexist peacefully with one another... but not always. You are going to store these `Agars` in a `HashSet`. `Agars` are inserted using a special `insertAgar` method: If you try to insert an `Agar` into a hash bucket, and there is already an `Agar` with exactly half its size in the same bucket, then the new `Agar` will eat the smaller `Agar`, absorb its size (new size will be the original size plus the eaten `Agar`'s size), and then attempt insertion (now bigger) into the `HashSet` again. It is possible for an `Agar` to eat multiple other `Agars` before finally being inserted into the `HashSet`. `Agars` do not eat `Agars` in other buckets. **Next page has an example and clarifications.**

Your ultimate goal for this problem is to write the code for `insertAgar` to successfully emulate these rules. Don't worry about `null` input cases. Here is a description for each of your input arguments:
- `x`, the `Agar` that you are inserting. You have access to its `public` fields and methods.
- `set`, the `HashSet` that will contain all `Agars` so far (if they haven't been gobbled up). It should be noted all instance variables of `HashSet` are private.
- `M`, the number of buckets in the `HashSet`. You may assume `M` does not change during the method call.

a) Write a helper method that returns `true` if a given `Agar x` is going to eat a smaller `Agar`. Assume `HashSet` uses the same process for converting `hashCodes` to bucket numbers that we used in class. Note that the `hashCode` is always positive, so there's no need for `abs` or `& 0x7FFFFFFF`[1]. **For full credit, your solution should be as asymptotically fast as possible (we won't say exactly how fast).**

```
static boolean shouldEatSomething(Agar x, HashSet<Agar> set, int M) {
    ________________________________________________
    ________________________________________________
    ________________________________________________
    ________________________________________________
    ________________________________________________
    ________________________________________________
    ________________________________________________
    ________________________________________________
    ________________________________________________
    ________________________________________________     }
```

[1] Though we didn't discuss in HW3: the reason we did & 0x7FFFFFFF was to handle negative hash codes.

b) Now finally write `insertAgar`. You may use `shouldEatSomething` even if you did not implement it successfully. The simplest solution will use `shouldEatSomething`, but it is possible to implement `insertAgar` without using it.

```
static void insertAgar(Agar x, HashSet<Agar> set, int M) {
    ______if set.contains(x) { return; }________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
}
```

Example:

- Suppose we have a `HashSet` called `set` with 1000 buckets, and there is an `Agar` of size 20 in bucket 0, an `Agar` of size 30 in bucket 1, and an `Agar` of size 180 in bucket 5.
- Suppose we create `Agar chris = new Agar(40)`, then call `insertAgar(chris, set, 1000)`. If (and only if) `chris` tries to go into bucket 0, `chris` will eat the `Agar` of size 20 (destroying it), increase in size to 60, and then attempt insertion again, potentially with a different hash code. If (and only if) `chris` (now of size 60) happens to try to go into bucket 1 this time, `chris` will eat the `Agar` there of size 30, increase in size to 90, and then attempt insertion yet again. If `chris` tried to go into bucket 5, he'd get inserted into that bucket, joining the existing `Agar` of size 180. `chris` is not eaten, since the size 180 `Agar` was not just inserted.

Some clarifications and notes:

- Assume that the size never exceeds the maximum integer value in Java: 2147483647.
- Assume that the size of an existing `Agar` is never changed by any code other than yours.
- Any call to `insertAgar` should increase the number of `Agars` by at most one, but could actually decrease the number of `Agars`, if some `Agars` are eaten.
- If we call `insert(x, set, M)` and `x` exists in the `HashSet`, we abort the operation, even if `shouldEatSomething(x, set, M)` is true.
- public HashSet methods include `add`, `addAll`, `clear`, `contains`, `containsAll`, `equals`, `getClass`, `hashCode`, `isEmpty`, `iterator`, `remove`, `removeAll`, `size`, `toArray`, `toString`.