STOP!

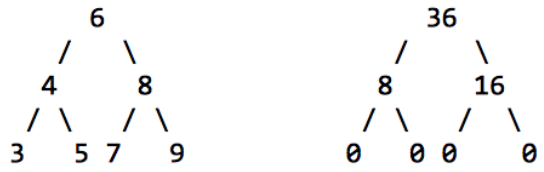
Before you proceed:

Don’t be tempted to check the solutions if you feel like you understand the solution to the problem.

Make sure you fully attempt a problem before checking its solution. This way, you can simulate an exam setting and see how you’d do if a particular problem were on the exam!

c. Suppose that we want to write a method `sumDescendants`, which replaces the value of each node in a tree with the sum of all of its descendants' values (not including itself), and then returns the sum of its original value (before being changed) plus all of its descendants' values.

For example, given the tree on the left, `sumDescendants` on node 6 would return 42 and change the tree to look like the one on the right (since 36 + 6 = 42).



Fill in the `sumDescendants` method. You may not need all lines. Do not use more lines.

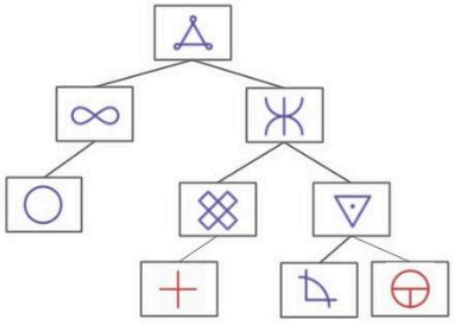
```
public class TreeNode {
    public TreeNode left, right;
    public int value;

    public TreeNode(int n) {
        value = n;
    }

    /* Replaces value with sum of all of its descendants' values. */
    public int sumDescendants() {
        if (left == null && right == null) {
            int oldVal = value;
            value = 0;
            return oldVal;
        }
        else {
       int leftSum = 0;int rightSum = 0;
       if (left != null) {
           leftSum = left.sumDescendants();
       }
       if (right != null) {
           rightSum = right.sumDescendants();
       }
       int oldVal = value;
       value = leftSum + rightSum;
            return oldVal + value;
        }
    }
}
```

3 Every *K*th Element (Fall 2014 MT1 Q5)

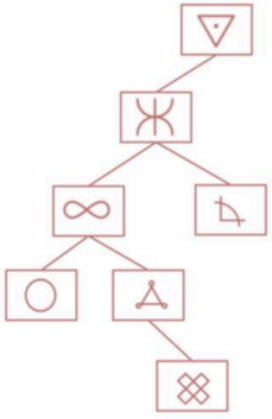
Fill in the `next()` method in the following class. Do not modify anything outside of `next`.

```
import java.util.Iterator;
import java.util.NoSuchElementException;
/** Iterates over every Kth element of the IntList given to the constructor.
*   For example, if L is an IntList containing elements
*   [0, 1, 2, 3, 4, 5, 6, 7] with K = 2, then
*       for (Iterator<Integer> p = new KthIntList(L, 2); p.hasNext(); ) {
*           System.out.println(p.next());
*       }
*   would print get 0, 2, 4, 6. */
public class KthIntList implements Iterator <Integer> {
    public int k;
    private IntList curList;
    private boolean hasNext;

    public KthIntList(IntList I, int k) {
        this.k = k;
        this.curList = I;
        this.hasNext = true;
    }

    /** Returns true iff there is a next Kth element. Do not modify. */
    public boolean hasNext() {
        return this.hasNext;
    }

    /** Returns the next Kth element of the IntList given in the constructor.
     *  Returns the 0th element first. Throws a NoSuchElementException if
     *  there are no Integers available to return. */
    public Integer next() {
        if (curList == null) {
            throw new NoSuchElementException();
        }
        Integer toReturn = curList.head;
        for (int i = 0; i < k && curList != null; i++) {
            curList = curList.tail;
        }
        hasNext = (curList != null);
        return toReturn;
    }
}
```

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, NOT THE TREE WITH YOUR HANDWRITTEN SYMBOLS** 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.**

insert(⊜): ⊜ > ▽ — (Drawn In Tree Above) — Not Enough Information

insert(◯): ◯ > ∞ — Drawn In Tree Above — (Not Enough Information)

insert(+): ⅄ < + < ⊠ — (Drawn In Tree Above) — Not Enough Information

insert(≶): ⅄ < ≶ < ▽ — Drawn In Tree Above — (Not Enough Information)

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.

$\Theta(N^2)$

```
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!");
        }
    }
}
```

The outer for loop makes N iterations. In each of these, the inner for loop makes N/2 iterations, so we have running time $N*(N/2) = \Theta(N^2)$

$\Theta(N \log N)$

```
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!");
        }
    }
}
```

The outer for loop makes N iterations. In each of these, the inner for loop makes $\log_2 N$ iterations, so we have running time $N*(\log_2 N) = \Theta(N \log N)$

$\Theta(N)$

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

Think of the recursive calls as nodes in a binary tree. This makes a perfectly balanced tree with height $\log_2 N$. The work done in each node is independent of N (constant), so the total runtime is proportional to the number of nodes in the tree. This gives us the sum $1 + 2 + 4 + 8 + \ldots + 2^{\log N} = \Theta(N)$.

$\Theta(1)$

```
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");
      }
}
```

None of the computation depends on N so p4 runs in constant time!

$\Theta(N^2)$

```
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");
            }
      }
}
```

This time the inner for loop is different for differing values of i, so we cannot simply multiply the iterations of the two for loops. The inner for loop runs i iterations at each i, so to get the overall runtime we sum over the sequence of values of i. Since the outer for loop does `i *= 2` each iteration, we sum over powers of 2, up until the value N^2. This looks like $1 + 2 + 4 + 8 + \ldots + N^2 = \Theta(N^2)$.

Note: This is a similar sum to p3. A sum of the powers of 2 is in big theta of the largest term, which was N in p3 but N^2 in p5.

5. Code Analysis (2.5 points).

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

$\Theta(n)$
```
public static void f1(int n) {
    for (int i = 0; i < 2*n; i += 1) {
        System.out.println("hello");
    }
}
```

$\Theta(n \log n)$
```
public static void f2(int n) {
    if (n == 0) { return; }
    f2(n/2);
    f1(n);
    f2(n/2);
}
```

Note: The problem above is the same pattern as mergesort.

$\Theta(n \log n)$
```
public static void f3(int n) {
    if (n == 0) { return; }
    f3(n/3);
    f1(n);
    f3(n/3);
    f1(n);
    f3(n/3);
}
```

Note: The problem above is the same pattern as mergesort, but now with 3 subproblems,all of size N/3.

$\Theta(2^n)$
```
public static void f4(int n) {
    if (n == 0) { return; }
    f4(n-1);
    f1(17);
    f4(n-1);
}
```

$\Theta(n^m)$
```
public static void f5(int n, int m) {
    if (m <= 0) {
        return;
    } else {
        for (int i = 0; i < n; i += 1) {
            f5(n, m-1);
        }
    }
}
```

2 Hashtable Runtimes (Fall 2016 MT2: Q3)

Consider a hash table that uses external chaining and also keeps track of the number of keys that it contains. It stores each key at most once; adding a key a second time has no effect. It takes the steps necessary to ensure that the number of keys is always less than or equal to twice the number of buckets (i.e., that the load factor is ≤ 2). Assume that its hash function and comparison of keys take constant time. All bounds should be a function of N, the number of elements in the table.

1. Give $\Theta()$ bounds on the worst-case times of adding an element to the table when the load factor is 1 and when it is exactly 2 before the addition.

   ```
   Bound for load factor 1: Θ(N)
   Bound for load factor 2: Θ(N)
   ```

2. Assume that the hashing function is so good that it always evenly distributes keys among buckets. What now are the bounds on the worst-case time of adding an element?

   ```
   Bound for load factor 1: Θ(1)
   Bound for load factor 2: Θ(N)
   ```

3. Making no assumption about the goodness of the hashing function, suppose that instead of using linked lists for the buckets, we use some kind of binary search tree that somehow keeps itself "bushy." What bound can you place on the worst-case time for testing to see if an item is in the table?

   ```
   Bound: Θ(lg N)
   ```

4. Using the same representation as in part (c), but with a very good hash function, as in part (b), what bound can you place on the worst-case time for testing to see if an item is in the table?

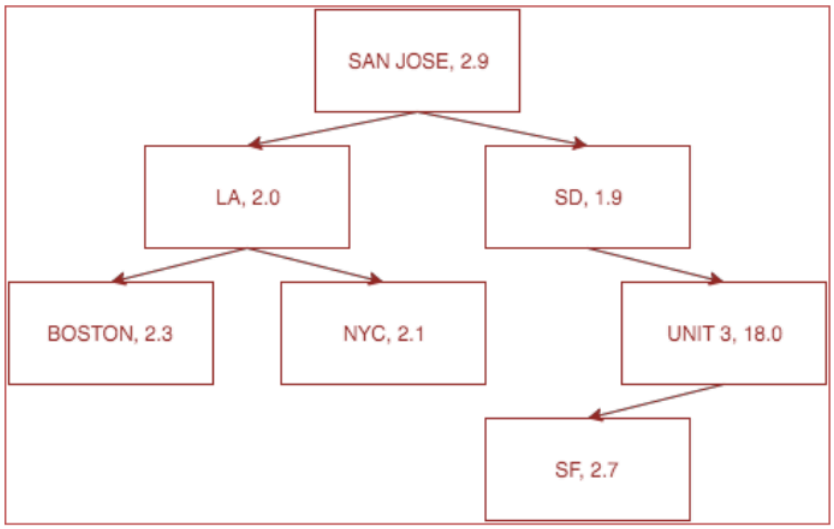
   ```
   Bound: Θ(1)
   ```

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

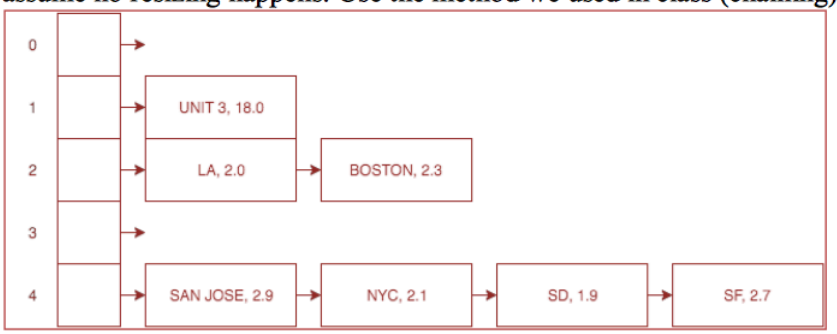
a) Suppose we're building a map that represents the rental cost in dollars per square foot of various locations. Starting from an initially empty BSTMap, if we call `put("SAN JOSE", 2.9)`, we'd get the tree shown in the box containing one node, with height equal to zero.

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`.

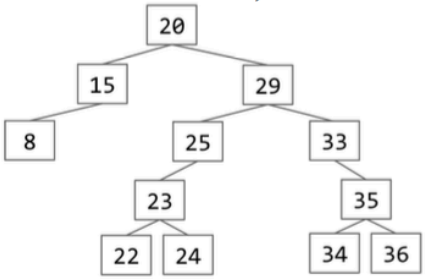
```
put("LA", 2.0)
put("NYC", 2.1)
put("SD", 6.3)
put("BOSTON", 2.3)
put("UNIT 3", 18.0)
put("SF", 2.7)
put("SD", 1.9)
```

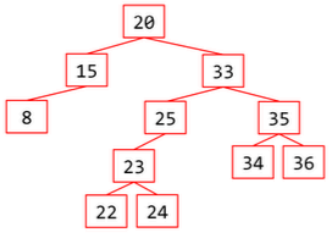


b) Suppose we repeat the exercise from part a, but with a hash map. Assume the `hashCode` of our strings is equal to the number of the first letter of the string. For example, the `hashCode("SF")` = 19. For reference, B → 2, L→ 12, N → 14, S → 19, U → 21. Assume we have 5 buckets, and assume no resizing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

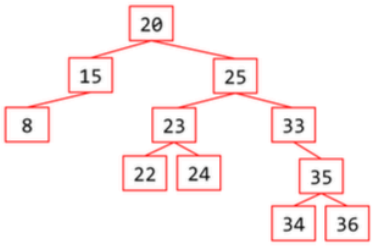


c) Suppose we use a BST to represent a `TreeSet`. Suppose we call `remove(29)` on the `TreeSet` below. Draw a valid BST that results. You must use the deletion procedure from class (also known as Hibbard deletion). At most two references should change. Draw your tree in the space to the right.





or



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.

Yes. Though our key's poor `hashCode` method will cause all keys to be thrown into one bucket, the `HashMap` will still be able to function correctly by iterating over the keys in that bucket and checking if each is `.equals` to the query key. This is guaranteed to work because `equals` is properly implemented.

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.

No. `Glelk`'s `hashCode` method calls the `Object` class' `hashCode` method, which simply returns the object's memory address. This means that two keys considered equal by `Glelk`'s equals method may hash to different buckets, causing keys to "disappear" from the `HashSet`.

```
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;
      }
}
```