UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2018

Final Examination Solutions

READ THIS PAGE FIRST. *Please do not discuss this exam with people who haven't taken it.* Your exam should contain 10 problems on 18 pages. Officially, it is worth 46 points (out of a total of 200).

This is an open-book test. You have two hours and fifty minutes to complete it. You may consult any books, notes, or other non-responsive objects available to you. You may use any program text supplied in lectures, problem sets, or solutions. Please write your answers in the spaces provided in the test. Make sure to put your name, login, and TA in the space provided below. Put your login and initials *clearly* on each page of this test and on any additional sheets of paper you use for your answers.

Be warned: my tests are known to cause panic. Fortunately, this reputation is entirely unjustified. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious. Should you feel an attack of anxiety coming on, feel free to jump up and run around the outside of the building once or twice.

Your name: _____ Login: _____

Your SID: _____ Discussion TA: _____

Login of person to your Left: _____ Right: _____

Please sign:

I pledge my honor that during this examination, I have neither given nor received assistance.

Signature: _____

1. [3 points] Fill in the blanks in the following to fulfill the comments.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

class Args {

    /** A Java pattern that matches a skeletal version of argument
     *  lists in Python, such as
     *       (A)
     *       (AA,A)
     *       (AA,A=AAA,AAA=A)
     *  The arguments themselves and the keywords that appear
     *  to the left of "=" must be non-empty sequences of A's. Any
     *  keyword arguments (with =) must follow the positional
     *  (non-keyword) arguments, and there must be at least one
     *  positional argument.  There will be no whitespace in the string.
     */
    static final String ARGS =

      "\\(A+(,A+)*((,A+=A+)*)\\)                                              ";

    /** Group number KEYWORDS in the regular expression ARGS captures
     *  the keyword arguments and the preceding comma.  See comments
     *  on main for an example. */

    static final int KEYWORDS = 2                        ;

    /** Prints the trailing keyword arguments in INPUTS[0],
     *  if INPUTS[0] is a valid argument list.  So
     *       java Args "(A,AA,AAA=A,AA=AAAA)"
     *  prints the string ",AAA=A,AA=AAAA" and
     *       java Args "(AAA,A)"
     *  prints a blank line. */
    public static void main(String... inputs) {
        Matcher matcher = Pattern.compile(ARGS).matcher(inputs[0]);
        if (matcher.matches()) {
            System.out.println(matcher.group(KEYWORDS));
        }
    }

}
```

2. [3 points] These questions concern the bit operators

```
&  |  ^  ~  << >> >>>
```

with the `int` operands `x` and `y`. Fill in the blanks so that the indicated equalities are true.

a.
```
x < 0 == ( 0 != x & 0x80000000                                    );
      == ( 1 == x >>> 31 ); // Another solution
```

b. Assume that `x` and `y` are integers of the form $0z0z0z0z0z0z\ldots0z$ (in binary) where each z can be 0 or 1. That is, each odd-numbered bit (counting from 0 at the right) is 0. For example, 0, 1, 5, 16, and 20 qualify, but not 2, 9, 36, or any negative number. Fill in the blank to make the equation true, using **only** integer literals, bit operators, the variables `x` and `y`, and parentheses:

```
x + y == (x ^ y) | ((x & y) << 1)                                 ;
```

3. [7 points] Fill in the appropriate bubbles or provide short answers to the following.

a. In a hash set that uses external chaining, one can replace the external chains with balanced binary search trees. Suppose we double the capacity of our set as needed so that the load factor does not exceed 3. What is the resulting worst-case cost of finding a key (as a function of N, the number of keys in the set)? Assume the hashing function takes constant time.

◯ $\Theta(1)$ ● $\Theta(\lg N)$ ◯ $\Theta(N)$ ◯ $\Theta(N \lg N)$ ◯ $\Theta(N^2)$ ◯ $\Theta(N^3)$ ◯ $\Theta(2^N)$

b. Under the same conditions as part (a), what is the worst-case time for adding a key to the set?

◯ $\Theta(1)$ ◯ $\Theta(\lg N)$ ◯ $\Theta(N)$ ● $\Theta(N \lg N)$ ◯ $\Theta(N^2)$ ◯ $\Theta(N^3)$ ◯ $\Theta(2^N)$

c. Again, under the same assumptions as in part (a), what is the worst-case time for adding N keys to an initially empty set?

◯ $\Theta(1)$ ◯ $\Theta(\lg N)$ ◯ $\Theta(N)$ ● $\Theta(N \lg N)$ ◯ $\Theta(N^2)$ ◯ $\Theta(N^3)$ ◯ $\Theta(2^N)$

d. The representation of part (a) is seldom used; one usually uses simple linked lists for each bucket, and this choice is generally faster. Why?

For decent hash functions, the buckets will remain small (near the load factor), so simple sequential search will be quite fast, since it avoids the overheads of constructing and manipulating the tree.

e. One operation in Dijkstra's algorithm is to examine a neighbor of a vertex in the graph and possibly update its estimated distance to the source. As a function of V, the number of vertices in the graph, how often does this operation occur in the worst case?

◯ $\Theta(1)$ ◯ $\Theta(\lg V)$ ◯ $\Theta(V)$ ◯ $\Theta(V \lg V)$ ● $\Theta(V^2)$ ◯ $\Theta(V^3)$ ◯ $\Theta(2^V)$

f. What is the answer to part (e) if the graph is required to be undirected and acyclic?

◯ $\Theta(1)$ ◯ $\Theta(\lg V)$ ● $\Theta(V)$ ◯ $\Theta(V \lg V)$ ◯ $\Theta(V^2)$ ◯ $\Theta(V^3)$ ◯ $\Theta(2^V)$

g. If a trie contains N keys whose combined length is B characters, how long does it take in the worst case to see if a string S of length M is a *prefix* of some string in the trie (that is, if some string in the trie begins with S)?

◯ $\Theta(\lg N)$ ◯ $\Theta(\lg B)$ ◯ $\Theta(\lg M)$ ◯ $\Theta(N)$ ◯ $\Theta(B)$ ● $\Theta(M)$

4. [6 points] In the following questions, notations such as $A \subseteq B$ mean that every function in the set of functions A is also in the set of functions B. Likewise, $A = B$ means that A and B are the same set of functions. Fill in the appropriate bubbles.

a. True or false: $O(f(N)) \subseteq \Theta(f(N))$.

◯ True ● False

b. True or false: $\Omega(f(N)) \subseteq O(f(N))$.

◯ True ● False

c. True or false: $\Theta(2 \cdot 3^N + 1000 \cdot 2^N \lg N) = \Theta(3^N)$.

● True ◯ False

d. What is the worst-case running time of `f(N)` as a function of `N`? Assume that `h` is a constant-time function.

```
int f(int N) {
    if (N <= 4)    return N;
    else if (h(N)) return 1 + f(N - 1);
    else           return 4 + f(N - 2);
}
```

◯ $\Theta(1)$ ◯ $\Theta(\lg N)$ ● $\Theta(N)$ ◯ $\Theta(N \lg N)$ ◯ $\Theta(N^2)$ ◯ $\Theta(N^3)$ ◯ $\Theta(2^N)$

e. What is the worst-case running time of `merge(A,B)` as a function of N, the sum of the lengths of `A` and `B`?

```
public static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    int i, j;  i = j = 0;
    while (i < A.length || j < B.length) {
        while (i < A.length && (j >= B.length || A[i] <= B[j])) {
            C[i + j] = A[i];
            i += 1;
        }
        while (j < B.length && (i >= A.length || B[j] < A[i])) {
            C[i + j] = B[j];
            j += 1;
        }
    }
    return C;
}
```

◯ $\Theta(1)$ ◯ $\Theta(\lg N)$ ● $\Theta(N)$ ◯ $\Theta(N \lg N)$ ◯ $\Theta(N^2)$ ◯ $\Theta(N^3)$ ◯ $\Theta(2^N)$

f. What is the worst-case running time of the call `new Seq().llcs(A, B)`, as a function of N, where N is the maximum of the lengths of **int** arrays `A` and `B`?

```
class Seq {
    private int[][] _val;
    private int[] _A, _B;
    public int llcs(int[] A, int[] B) {
        _val = new int[A.length + 1][B.length + 1];
        _A = A; _B = B;
        return llcs(A.length, B.length);
    }
    private int llcs(int lenA, int lenB) {
        if (lenA == 0 || lenB == 0) {
            return 0;
        } else if (_val[lenA][lenB] == 0) {
            if (_A[lenA - 1] == _B[lenB - 1]) {
               _val[lenA][lenB] = 2 + llcs(lenA - 1, lenB - 1);
            } else {
                _val[lenA][lenB] = 1 + max(llcs(lenA - 1, lenB),
                                           llcs(lenA, lenB - 1));
            }
        }
        return _val[lenA][lenB] - 1;
    }
}
```

○ $\Theta(1)$ ○ $\Theta(\lg N)$ ○ $\Theta(N)$ ○ $\Theta(N \lg N)$ ● $\Theta(N^2)$ ○ $\Theta(N^3)$ ○ $\Theta(2^N)$

5. [1 point] The following collection of measurements of fundamental physical constants has been proposed to define a certain set of quantities. What set is it?

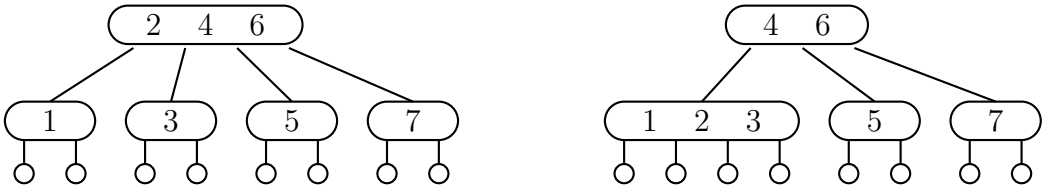
$$\begin{aligned}
\Delta v(^{133}\mathrm{Cs})_{\mathrm{hfs}} &= 9{,}192{,}631{,}770 \ \ \mathrm{sec}^{-1} \\
c &= 299{,}792{,}458 \ \ \mathrm{m\ sec}^{-1} \\
h &= 6.62607015 \times 10^{-34}\ \mathrm{kg\ m}^2\ \mathrm{sec}^{-1} \\
e &= 1.602176634 \times 10^{-19}\ \mathrm{A\ sec} \\
k &= 1.380649 \times 10^{-23}\ \mathrm{kg\ m}^2\ \mathrm{K}^{-1}\ \mathrm{sec}^{-2} \\
N_A &= 6.02214076 \times 10^{23}\ \mathrm{mol}^{-1} \\
K_{\mathrm{cd}} &= 683\ \mathrm{cd\ sr\ sec}^3\ \mathrm{kg}^{-1}\ \mathrm{m}^{-2}
\end{aligned}$$

They implicitly provide the new definitions of the SI base units (second, meter, kilogram, ampere, kelvin, mole, and candela) by reference to fundamental physical constants (hyperfine structure transition frequency of cesium 133, speed of light, Planck constant, elementary charge, Boltzmann constant, Avagadro constant, and luminous efficacy of monochromatic radiation of frequency 540×10^{12} Hz.)

6. [6 points]

a. Give two distinct (2, 4) trees containing the keys $\{1, 2, 3, 4, 5, 6, 7\}$.

Answer:



b. We'll say that a red-black tree with exactly one key in it:

has two black nodes along each path from the root to the (empty) leaves (each path contains the root and an empty node, usually represented as null). Consider a red-black tree in which there are exactly three black nodes along each path from the roots to the (empty) leaves. What is the maximum number of keys it can contain? Assume that we consider all general red-black trees that correspond to (2, 4) trees.

○ 3 ○ 4 ○ 6 ○ 7 ○ 8 ○ 14 ● 15 ○ 16 ○ 30 ○ 31 ○ 32

c. Again consider a red-black tree in which there are exactly three black nodes along each path from the roots to the (empty) leaves. What is the maximum number of keys it can contain if we restrict ourselves to red-black trees that correspond to (2, 3) trees?

○ 3 ○ 4 ○ 6 ○ 7 ● 8 ○ 14 ○ 15 ○ 16 ○ 30 ○ 31 ○ 32

7. [5 points] Below you will find some intermediate steps in performing various sorting algorithms on the same input list. The steps do not necessarily represent consecutive steps in the algorithm (that is, many steps are missing), but they are in the correct sequence. For each of them, select the algorithm (by filling in the appropriate bubble) from among the following choices: insertion sort, selection sort, mergesort, quicksort (first element of sequence as pivot), heapsort, LSD radix and MSD radix sort.

In all these cases, the final step of the algorithm will be this:

```
900, 1741, 3134, 3689, 4344, 4766, 5610, 7128, 8191, 8542, 8800, 9176
```

but it may not be shown.

Input List:

```
5610, 8542, 8191, 1741, 8800, 4344,  900, 4766, 7128, 3134, 9176, 3689
```

a. ◯ Ins. ◯ Sel. ◯ Mer. ◯ Qui. ● Heap. ◯ LSD. ◯ MSD.

```
5610, 8542, 8191, 1741, 8800, 4344,  900, 4766, 7128, 3134, 9176, 3689
9176, 8800, 8191, 7128, 8542, 4344,  900, 1741, 4766, 3134, 5610, 3689
3689, 8800, 8191, 7128, 8542, 4344,  900, 1741, 4766, 3134, 5610, 9176
8800, 8542, 8191, 7128, 5610, 4344,  900, 1741, 4766, 3134, 3689, 9176
3689, 8542, 8191, 7128, 5610, 4344,  900, 1741, 4766, 3134, 8800, 9176
4344, 3689, 3134, 1741,  900, 4766, 5610, 7128, 8191, 8542, 8800, 9176
 900, 3689, 3134, 1741, 4344, 4766, 5610, 7128, 8191, 8542, 8800, 9176
```

b. ● Ins. ◯ Sel. ◯ Mer. ◯ Qui. ◯ Heap. ◯ LSD. ◯ MSD.

```
5610, 8542, 8191, 1741, 8800, 4344,  900, 4766, 7128, 3134, 9176, 3689
5610, 8191, 8542, 1741, 8800, 4344,  900, 4766, 7128, 3134, 9176, 3689
1741, 5610, 8191, 8542, 8800, 4344,  900, 4766, 7128, 3134, 9176, 3689
 900, 1741, 4344, 4766, 5610, 8191, 8542, 8800, 7128, 3134, 9176, 3689
 900, 1741, 3134, 4344, 4766, 5610, 7128, 8191, 8542, 8800, 9176, 3689
```

c. ◯ Ins. ◯ Sel. ● Mer. ◯ Qui. ◯ Heap. ◯ LSD. ◯ MSD.

```
5610, 8542, 8191, 1741, 8800, 4344,  900, 4766, 7128, 3134, 9176, 3689
5610, 8191, 8542, 1741, 4344, 8800,  900, 4766, 7128, 3134, 9176, 3689
1741, 4344, 5610, 8191, 8542, 8800,  900, 4766, 7128, 3134, 9176, 3689
1741, 4344, 5610, 8191, 8542, 8800,  900, 3134, 3689, 4766, 7128, 9176
 900, 1741, 3134, 3689, 4344, 4766, 5610, 7128, 8191, 8542, 8800, 9176
```

Continued on next page.

d. ◯ Ins. ◯ Sel. ◯ Mer. ● Qui. ◯ Heap. ◯ LSD. ◯ MSD.

```
5610, 8542, 8191, 1741, 8800, 4344,  900, 4766, 7128, 3134, 9176, 3689
3689, 1741, 4344,  900, 4766, 3134, 5610, 8800, 7128, 8191, 9176, 8542
3689, 1741, 4344,  900, 4766, 3134, 5610, 7128, 8191, 8542, 8800, 9176
3134, 1741,  900, 3689, 4766, 4344, 5610, 7128, 8191, 8542, 8800, 9176
```

e. ◯ Ins. ◯ Sel. ◯ Mer. ◯ Qui. ◯ Heap. ● LSD. ◯ MSD.

```
5610, 8542, 8191, 1741, 8800, 4344,  900, 4766, 7128, 3134, 9176, 3689
5610, 8800,  900, 8191, 1741, 8542, 4344, 3134, 4766, 9176, 7128, 3689
8800,  900, 5610, 7128, 3134, 1741, 8542, 4344, 4766, 9176, 3689, 8191
```

8. [4 points] We saw quite a variety of implementations of graphs and traversals in project 3. These questions concern some of these. For each, fill in the appropriate bubble. We assume that during the traversal, when a successor vertex, `v`, of a vertex, `u`, is given a new weight, `w`, we use the following code before `v` is added back into the fringe.

```
_fringe.remove(v);
setWeight(v, w);
setPredecessor(v, u);
```

[Reminder: `P1 ? V1 : P2 ? V2 : V3` has the value `V1` if `P1`, otherwise `V2` if `P2`, and otherwise, `V3`.] Assume the algorithm is given a connected graph with E edges.

(The comment "`// *CHANGED*`" indicates lines that have changed from the previous part.)

Important: In each question that follows, fill in only the *first* bubble that applies.

a.
```
/* Representation of fringe: */
    new TreeSet<Integer>(new VertexComparator());
/* Comparator */
    class VertexComparator implements Comparator<Integer> {
        public int compare(Integer v0, Integer v1) {
            double d0 = getWeight(v0) + estimatedDistance(v0),
                   d1 = getWeight(v1) + estimatedDistance(v1);
            return d0 > d1 ? 1 : d0 < d1 ? -1 : v0 - v1;
        }
    }
```

This representation

- ◯ Will not work, because items added to the fringe will disappear.
- ◯ Will not work, because the fringe will be ordered incorrectly.
- ● Will work properly and allow the shortest-path algorithm to run in worst-case $O(E \lg E)$ time.
- ◯ Will work properly, but will not allow the shortest-path algorithm to run in worst-case $O(E \lg E)$ time.

b.
```
/* Representation of fringe: */
    new PriorityQueue<Integer>(new VertexComparator());  // *CHANGED*
/* Comparator */
    class VertexComparator implements Comparator<Integer> {
        public int compare(Integer v0, Integer v1) {
            double d0 = getWeight(v0) + estimatedDistance(v0),
                   d1 = getWeight(v1) + estimatedDistance(v1);
            return d0 > d1 ? 1 : d0 < d1 ? -1 : v0 - v1;
        }
    }
```

Assume that the class `PriorityQueue` uses a heap, as we have done in class. This representation

- ◯ Will not work, because items added to the fringe will disappear.
- ◯ Will not work, because the fringe will be ordered incorrectly.
- ◯ Will work properly and allow the shortest-path algorithm to run in worst-case $O(E \lg E)$ time.
- ● Will work properly, but will not allow the shortest-path algorithm to run in worst-case $O(E \lg E)$ time.

c.
```
/* Representation of fringe: */
    new TreeSet<Integer>(new VertexComparator());   // *CHANGED*
/* Comparator */
    class VertexComparator implements Comparator<Integer> {
        public int compare(Integer v0, Integer v1) {
            double d0 = getWeight(v0) + estimatedDistance(v0),
                   d1 = getWeight(v1) + estimatedDistance(v1);
            return d0 > d1 ? 1 : d0 < d1 ? -1 : 0;  // *CHANGED*
        }
    }
```

This representation

- ● Will not work, because items added to the fringe will disappear.
- ◯ Will not work, because the fringe will be ordered incorrectly.
- ◯ Will work properly and allow the shortest-path algorithm to run in worst-case $O(E \lg E)$ time.
- ◯ Will work properly, but will not allow the shortest-path algorithm to run in worst-case $O(E \lg E)$ time.

d.
```
/* Representation of fringe: */
    new TreeSet<Integer>(new VertexComparator());
/* Comparator */
    class VertexComparator implements Comparator<Integer> {
        public int compare(Integer v0, Integer v1) {
            double d0 = getWeight(v0) + estimatedDistance(v0),
                   d1 = getWeight(v1) + estimatedDistance(v1);
            return d0 > d1 ? 1 : -1;    // *CHANGED*
        }
    }
```

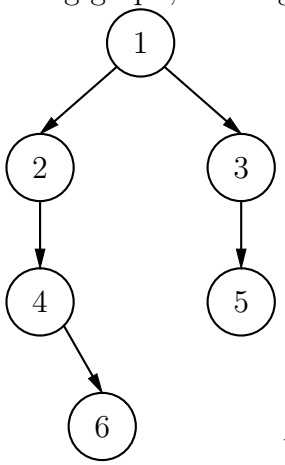
This representation

- ◯ Will not work, because items added to the fringe will disappear.
- ● Will not work, because the fringe will be ordered incorrectly.
- ◯ Will work properly and allow the shortest-path algorithm to run in worst-case $O(E \lg E)$ time.
- ◯ Will work properly, but will not allow the shortest-path algorithm to run in worst-case $O(E \lg E)$ time.

9. [2 points] We saw variations on the following main loop in the algorithm for doing depth-first search with pre- and post-visiting (the code is organized differently from the project). Here, `_fringe` is initialized to a stack containing the starting vertex and `_toBePostVisited` and `_marked` are sets.

```
while (!_fringe.isEmpty()) {
    int u = _fringe.pop();
    if (_toBePostVisited.contains(u)) {
        postVisit(u);
    } else if (!_marked.contains(u))
        _marked.add(u);
        visit(u);
        if (shouldPostVisit(u)) {
            _toBePostVisited.add(u);
            _fringe.add(u);
        }
        for (int v : _G.successors(u)) {
            if (!_marked.contains(v)) {
                _fringe.add(v);
            }
        }
    }
}
```

Unfortunately, this implementation is incorrect. Consider a depth-first traversal of the following graph, starting from vertex 1.



What one edge can we add to this graph so that (for some orderings of `_G.successors(u)`) the algorithm visits or postvisits nodes in the wrong order? There may be more than one answer; pick all that apply.

☐ 6 → 6 ☐ 2 → 1 ■ 2 → 3 ☐ 5 → 6 ☐ None of these.

10. [10 points] The abstract type `SearchTree<T>` on page 16 is intended to represent a general kind of search tree. If, for example, `IntTree` extends `SearchTree<Integer>`, then the intent is that a client can write

```
// S is a sentinel node representing an empty IntTree.
IntTree S = new IntTree();
S.add(5);
S.add(1);
S.add(6);
S.add(5);
System.out.println(S.contains(9) + " " + S.contains(6) + " " + S.contains(5));
```

and the program should print "false true true".

As you can see, the `SearchTree` class has been written so that the representation of its children is left up to its subtypes. Furthermore, unlike an ordinary binary search tree, its nodes may have any number of children. The sentinel root node will always have at least one child and a null label.

a. Fill in the definition of `contains` (on page 17) to fulfill its comment.

b. Fill in the definition of `add` (on page 17) to fulfill its comment.

c. Define the subtype `IntTree` (on page 18) to be a subtype of `SearchTree<Integer>`—that is, an ordinary binary search tree.

It is not necessary to check for erroneous inputs. In particular, where a documentation comment says that a value is non-null, or in some range, it is not necessary to check for that condition.

```
/** A general search-tree class containing values of type T. */
public abstract class SearchTree<T> {
    /** A SearchTree node with empty children and label LAB. */
    protected SearchTree(T lab) {
        _label = lab;
    }

    public T label() {
        return _label;
    }

    /** Returns a new SearchTree containing the single non-null label LAB */
    abstract protected SearchTree<T> create(T lab);

    /** Returns my K'th child, numbering from 0.  Error if there are fewer
     *  than K + 1 children. */
    abstract protected SearchTree<T> child(int k);

    /** Sets child(K) to C. Error if node has fewer than K+1 children. */
    abstract protected void setChild(int k, SearchTree<T> c);

    /** Returns an index, k, such that V must either be in my k'th child or
     *  is not present. Must always return 0 if label() is null.
     *  Otherwise undefined if V equals label() (that is, if the value
     *  V is contained in this node). */
    abstract protected int select(T v);

    /** True iff this tree contains the non-null value V. */
    public boolean contains(T v) {
        // To be implemented in part (a).
    }

    /** Insert the non-null value V into this tree (if necessary) so that
     *  contains(V) is true afterwards.  Does nothing if V is present. */
    public void add(T v) {
        // To be implemented in part (b).
    }

    final private T _label;
}
```

```
// Part a.
/** True iff this tree contains the non-null value V. */
public boolean contains(T v) {
    if (v.equals(_label)) {
        return true;
    } else {
        int k = select(v);
        return child(k) != null && child(k).contains(v);
    }
}

// Part b.
/** Insert the non-null value V into this tree (if necessary) so that
 *  contains(V) is true afterwards.  Does nothing if V is present. */
public void add(T v) {
    if (!v.equals(_label)) {
        int k = select(v);
        if (child(k) == null) {
            setChild(k, create(v));
        } else {
            child(k).add(v);
        }
    }
}
```

```
// Part c.
public class IntTree extends SearchTree<Integer> {

    public IntTree() {
        super(null);
    }

    public IntTree(int v) {
        super(v);
    }

    @Override
    protected SearchTree<Integer> create(Integer v) {
        return new IntTree(v);
    }

    @Override
    protected SearchTree<Integer> child(int k) {
        return _child[k];
    }

    @Override
    protected void setChild(int k, SearchTree<Integer> c) {
        _child[k] = c;
    }

    @Override
    protected int select(Integer v) {
        if (label() == null || v < label()) {
            return 0;
        } else {
            return 1;
        }
    }

    private SearchTree<Integer>[] _child = new IntTree[2];
}
```