## Problem 1: Speak to Me

Below is a table of various data structures and operations over those data structures. Fill in the blanks in the table with the average-case time complexity of each of these combinations. If a time complexity has caveats—*e.g.*, the runtime is amortized or the structure requires balancing—state so in your answer. Entries that are filled with a black box (■) indicate invalid entries and should not be filled in.

| | `add` | `get` | `remove` | `contains` |
|---|---|---|---|---|
| Array List | (to end) | (at index $i$) | (element $v$) | |
| (Singly-) Linked List | (to end) | (at index $i$) | (element $v$) | |
| Binary Search Tree | | ■ | | |
| *Hash Map* | (*i.e.*, `put(k, v)`) | | | (*i.e.*, `hasKey()`) |
| *Heap* | | (*i.e.*, `peek()`) | (*i.e.*, `poll()`) | ■ |

For each of the following statements, determine whether it is *true* (T) or *false* (F).

(1) If an object needs to hide a particular field from other objects (of different type), then that field should be marked `private`. (T) (F)

(2) Non-static methods can access the static members declared inside its containing class. (T) (F)

(3) An array is an appropriate structure when we wish to store a fixed number of elements of different types. (T) (F)

(4) A linked list is a variable-size, sequential data structure. (T) (F)

(5) Subtype polymorphism is accomplished through generics in Java. (T) (F)

(6) The best-case runtime of quicksort is $\mathcal{O}(n \log n)$. (T) (F)

(7) The worst-case time complexity of merge sort is $\mathcal{O}(n^2)$. (T) (F)

(8) Insertion sort is appropriate when not all of the data to sort is immediately available. (T) (F)

(9) The `map` function of a stream allows us to selective drop elements from the stream based on some predicate function. (T) (F)

(10) A tree encodes hierarchical relationships between its elements. (T) (F)

(11) We can use a tree-like structure to efficiently store an ordered set of elements. (T) (F)

(12) Given a hash function $h$ and values $v_1$ and $v_2$, if $v_1 = v_2$ then $h(v_1) = h(v_2)$. (T) (F)

(13) We use interface or class extension to realize *has-a* relationships in Java. (T) (F)

(14) We prefer class inheritance over interface implementation because inheritance mechanically is simpler than interfaces. (T) (F)

(15) The heart of object-oriented programming—the defining feature not easily replicable in other language paradigms—is dynamic dispatch. (T) (F)

## Problem 2: Breathe

Draw the step-by-step evolution of a binary search tree after each of the given insert and removal operations. Assume that removal chooses the next largest element in the in-order traversal of the tree as the value to rotate upwards.

(a) `Tree<Integer> t = new Tree<>();`

(b) `t.insert(1);`

(c) `t.insert(2); t.insert(3); t.insert(4);`

(d) `t.remove(1);`

Draw the step-by-step evolution of a priority queue after each of the given `add` and `poll` operations. The priority queue is a *min* priority queue—that is, `poll` returns the minimum element in the queue. You do not need to distinguish between equivalent elements in your priority queue.

(e) `PriorityQueue<Integer> pq = new PriorityQueue<>(); pq.add(10);`

(f) `pq.insert(8); pq.insert(6); pq.insert(3);`

(g) `pq.insert(7);`

(h) `pq.poll();`

Draw the step-by-step evolution of two hash tables after each of the given put operations. The keys of the hash table are objects of type C. The table to the right describes the hash values of these objects. Throughout, *Make sure to write both the key and value in the table rather than just the key.*

| | |
|---|---|
| c1 | 0 |
| c2 | 3 |
| c3 | 4 |
| c4 | 5 |
| c5 | 1 |

(i) The first hash table is implemented with a *linear probing* strategy with an initial backing array of size 3. When this table is full, the table proceeds by first (1) doubling the backing array size and (2) rehashing the current elements of the table from left-to-right.

```
/* i   */  Map<C, Character> m = new HashMap<>();
           m.put(c1, 'a');

/* ii  */  m.put(c2, 'b');

/* iii */  m.put(c3, 'c');

/* iv  */  m.put(c4, 'd');

/* v   */  m.put(c5, 'e');
```

(b) The second hash table is implemented with a *separate chaining* strategy with an initial backing array of size 3. When the load factor (the number of entries divided by the size of the backing array) is 0.75 or greater, the table grows by (1) doubling the backing array size and (2) rehashing the current elements of the table from left-to-right.

| | |
|---|---|
| c1 | 0 |
| c2 | 3 |
| c3 | 4 |
| c4 | 5 |
| c5 | 1 |

```
/* i   */  Map<C, Character> m = new HashMap<>();
           m.put(c1, 'a');

/* ii  */  m.put(c2, 'b');

/* iii */  m.put(c3, 'c');

/* iv  */  m.put(c4, 'd');

/* v   */  m.put(c5, 'e');
```

## Problem 3: On the Run

Consider the following class hierarchy:

```
public class A {
    public void f1() { System.out.println("A.f1"); }
}
public class B extends A {
    public void f2() { System.out.println("B.f2"); }
}
public class C extends B {
    public void f1() { System.out.println("C.f1"); }
}
public class D extends B {
    public void f3() { System.out.println("D.f3"); }
}
```

For each of the following variable initialization statements, state (a) if it type checks and (b) if it does type check, what is the static and dynamic types of the variable.

(a) `B b = new B();` _____

(b) `D d = new C();` _____

(c) `A a = new D();` _____

(d) `A a = new C();` _____

For each of the combinations of variable initialization statements and method invocations, determine if (a) the method invocation type checks and (b) if so, the output of the method call.

| | f1() | f2() | f3() |
|---|---|---|---|
| `B b = new C();` | | | |
| `A a = new D();` | | | |

## Problem 4: Time

Consider the following implementation of an association list mapping integers to booleans in Java:

```
public class Pair {                                    /* Write your changes below */
    public int fst;
    public boolean snd;
    public Pair(int fst, boolean snd) {
        this.fst = fst;
        this.snd = snd;
    }
}

public class AssociationList {
    public List<Integer, Boolean> list;

    public AssociationList() {
        list = new List<>();
    }

    public boolean get(int key) {
        for (Pair p : list) {
            if (p.fst.equals(key)) {
                return p.snd;
            }
        }
        return null;
    }

    public void set(int key, boolean value) {
        Pair old = null;
        for (Pair p : list) {
            if (p.fst.equals(key)) { old = p; break; }
        }
        if (old != null) { list.remove(old); }
        list.add(new Pair(key, value));
    }
}
```

Fix this implementation so that it can hold any pair of key and value types by using generics.

## Problem 5: The Great Gig in the Sky

For each of the following methods:

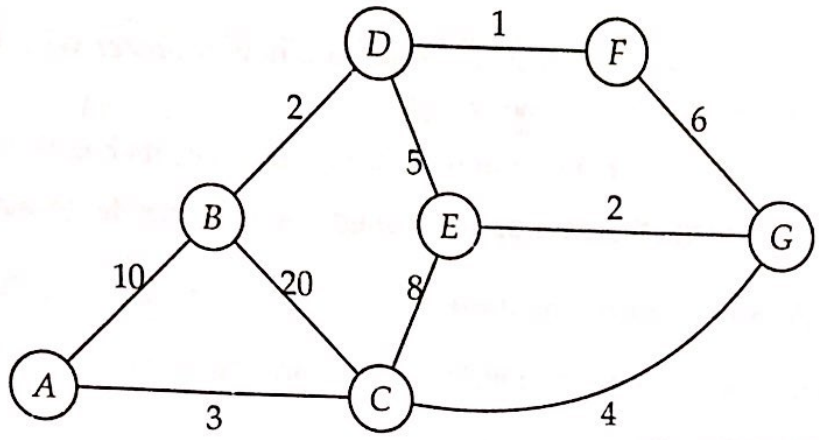
1. Give a mathematical function or recurrence that models the time complexity of the method. State explicitly what operations your function tracks as well as what the input to the function represents. If your model is a recurrence relation, solve that relation for an explicit mathematical function.
2. Give a tight upper-bound for your function using Big-O notation. You can simply state the upper-bound rather than formally proving it correct.

```
// pre: arr1, arr2 != null
  public static boolean[] f1(int[] arr1, int[] arr2, int k) {
    if (arr1.length != arr2.length) { return null; }
    boolean[] ret = new boolean[arr1.length];
    for (int i = 0; i < arr1.length; i++) {
        for (int j = i+1; j < arr2.length; j++) {
            if (arr[j] - arr[i] == k) { ret[i] = true; }
        }
    }
    return ret;
}
```

```
// pre: arr != null
// assume f2 is called initially with lo = 0, hi = arr.length
public static boolean f2(int v, int[] arr, int lo, int hi) {
    if (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        int sum = 0;
        for (int i = 0; i < arr.length; i++) { sum += arr[i]; }
        if (sum < 0) {
            return sum - f2(v, arr, lo, mid);
        } else {
            return sum + f2(v, arr, mid + 1, hi);
        }
    } else {
        return 0;
    }
}
```

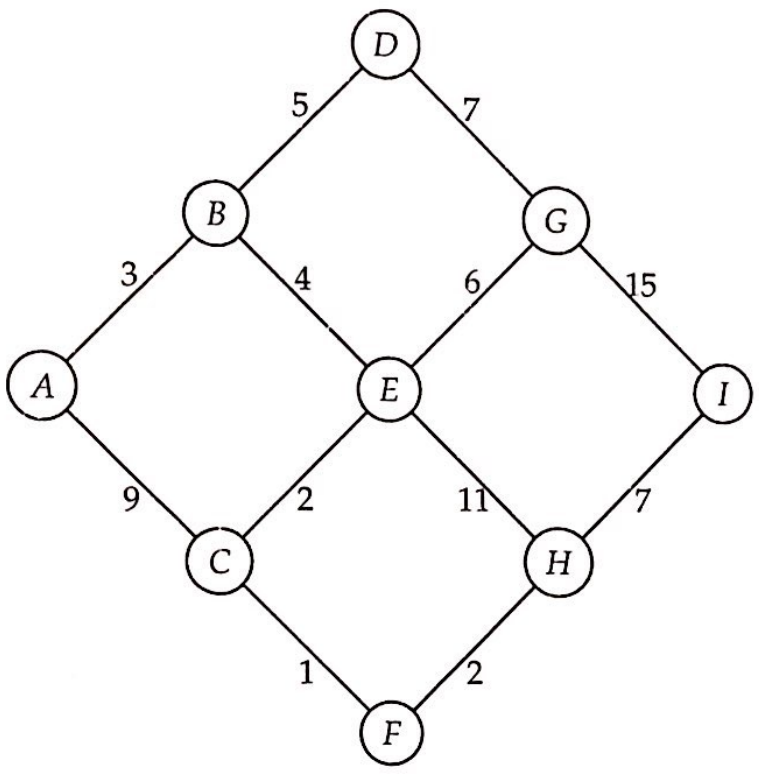
## Problem 6: Money

(a) Consider the following weighted, undirected graph. Clearly highlight in the graph a *minimum spanning tree* (MST) for the graph **and** give the total weight of the MST that you identify. You may use any method to compute the MST of this graph.




Minimum spanning tree weight: _____

(b) Consider the following weighted, undirected graph below. Use Dijkstra's algorithm to compute the shortest paths from node *A* to every other node in the graph. You should report in the space provided both the weight of the shortest path from *A* to each node as well as the path itself.




| Path to | Length | Path |
|---|---|---|
| *A* | | |
| *B* | | |
| *C* | | |
| *D* | | |
| *E* | | |
| *F* | | |
| *G* | | |
| *H* | | |
| *I* | | |

## Problem 7: Us and Them

Write a class `Monster` that represents a monster in a Nethack-style role-playing video game. A monster has a name, a position in the game world (a pair of integers $(x,y)$), and health and attack statistics represented as integers. Your class should support the following constructor and operations:

- `Monster(name, x, y, health, attack)`: creates a new monster with the given attributes located at position $(x,y)$ in the game world.
- `boolean isAlive()`: returns true if the monster is alive, *i.e.*, its health is positive.
- `int getDamage()`: returns the amount of damage the monster deals: attack $\times 2 - 1$.

In addition, you should also override the three standard methods from the `Object` class:

- `String toString()` which returns a string of the form `"name[(x, y), health, attack]"`.
- `boolean equals(Object o)`.
- `int hashCode()`.

Your work will be graded on correctness as well as design. In particular, you should only expose members that the interface defined above says you should expose.

## Problem 8: Any Colour You Like

Write a class, `InfiniteIterator<T>`, that walks backwards and forwards through an *array list* and iterates through it infinitely. That is, when the iterator reaches the end of the list and moves forward, it wraps around to the front. When the iterator is at the front of the list and moves backwards, it wraps around to the back.

Your `InfiniteIterator` class should have the following constructor and operations:

- `InfiniteIterator(ArrayList<T> l)`: constructs a new `InfiniteIterator` over the given `ArrayList<T>`.
- `boolean hasNext()`: returns `true` iff the `InfiniteIterator` still possesses elements to iterate over. (*Hint*: when might this be false?)
- `T next()`: returns the element the iterator is pointing at and moves the iterator forwards, wrapping to the front of the list if it walks off the end.
- `T prev()`: returns the element the iterator is pointing at and moves the iterator backwards, wrapping to the end of the list if it walks off the front.

## Problem 9: Brain Damage

In class, we discussed the *higher-order functions*, `map`, `filter`, and `fold` over our list abstract data type. It turns out that we can implement these functions over any container we please! In this problem, we'll implement these higher order functions over a *binary search tree*.

Recall that in Java 8, we work with anonymous functions (lambda expressions) as follows:

- The type of a lambda is dictated by a number of *functional interfaces* defined in the `java.util.function`. For our purposes, we only care about two: `Function<T, R>`—the type of functions that take a `T` and produce an `R` and `BiFunction<T, U, R>`—the type of functions that take a `T` and a `U` and produce an `R`.
- To invoke an anonymous function, we call the method specified by its functional interface. For the Java standard library types, this method is called `apply`, *e.g.*, `f.apply(5)` calls the lambda stored in `f` with the value 5.
- The syntax of a lambda itself is `(<parameters>) -> <expression body>`, *e.g.*, `(int x) -> x + 1` is a lambda that returns its argument, an integer, plus one.

For these methods, assume a standard definition of a binary search tree as presented in class:

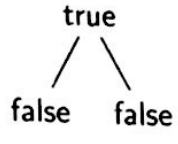
```
public class Node<T extends Comparable<T>> {
    public T value;
    public Node<T> left;
    public Node<T> right;
    public Node(T value, Node<T> left, Node<T> right) {
        this.value = value;
        this.left  = left;
        this.right = right;
    }
}

public class BinarySearchTree<T extends Comparable<T>> {
    private Node<T> root;
    // Your methods will go here...
}
```

(a) Write an implementation of the static *map* method over the `BinarySearchTree` class. Recall that map takes a transformation function over elements and produces a new tree whose elements are the result of applying this function to the elements of the old tree.

```
public static <T, U> BinarySearchTree<U> treeMap(Function<T, U> f,
                                                 BinarySearchTree<T> t);
```

For example, we may map the following input tree:


To the following output tree:

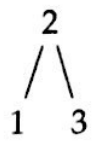



By passing a transformation function that returns true if the input is even. (*Hint:* You will need a helper function that performs recursive insertion over the nodes of a tree. This helper will then be called by your `treeMap` function with the root.)

(b) Complete the definition of the static *fold* method over the `BinarySearchTree` class with signature:

```
public static <T, R> R treeFold(BiFunction<R, T, R> f, R initial
                                BinarySearchTree<T> t) {
```

`fold` takes as input an initial value and a binary transformation function. This function takes the accumulated value so far as well as the current element of the tree and produces a new updated value. For example, if we fold the addition function over the following tree with an initial value of 0:



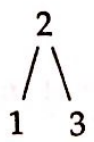

Then the result of the fold is $0 + 2 + 1 + 3 = 6$.

Assume that `fold` applies its argument function to the tree using an *in-order* traversal.

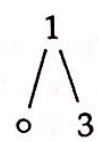
(c) Complete the definition of the static *filter* method over the `BinarySearchTree` class:

```
public static <T> BinarySearchTree<T> treeFilter(Function<T, Boolean> f,
                                                 BinarySearchTree<T> t);
```

`filter` takes as input an initial value and a function from elements of the tree to booleans. Filter then keeps all the elements of the tree where the function (or *predicate*) returns true for that element. For example, we may map the following input tree:




To the following output tree:




If our filter returned true for any integer that was odd.

To implement `filter`, assume the existence of a method `public boolean remove(T t)` in the `BinarySearchTree` class that removes the first occurrence of `t` from the tree in an in-order traversal, preserving the order of the binary search tree. The method returns the root of the updated tree after removal.

(*Hint:* to implement this function, first create a copy of the tree to filter using `map`. Then traverse the old tree, removing nodes from the new tree using the assumed `remove` method.)

## Problem 10: Eclipse

Imagine that you are writing a text editor program. The key piece of data in such a program is the *text buffer*. Because text is linear in nature, our first inclination is to store the text in an array, *e.g.*, an array list of characters:

```
['h', 'e', 'l', 'l', 'o', '\n', 'w', 'o', 'r', 'l', 'd']
                          ^
```

(Recall that character literals in Java are written with single quotes and `'\n'` is the character literal for a newline. Note that newlines are not treated specially—they are simply additional characters in the buffer).

The caret above represents the position of the cursor in the buffer. The cursor is an essential part of a text buffer because we can only edit text (add or remove characters) at the cursor.

(a) There is a big performance problem with this choice of representation, however, when we try to insert text into the buffer. What is it? Demonstrate this problem on the sample array list of characters give above (*Hint:* consider what array list operation we need to perform to insert text, and what happens if the text buffer is very large).

(b) How can we perform insertion at the cursor more efficiently? Describe an alternative way of representing text so that insertion is $\mathcal{O}(1)$ (potentially amortized over a fraction of the text). Demonstrate your scheme on the example list of characters above. (*Hint:* Under what circumstances can we add onto a list in $\mathcal{O}(1)$ time? With this in mind, how can we split up the text to take advantage of this fact when inserting at the cursor?)

(c) Design a data structure that allows us to perform insertion into the text at the cursor more efficiently than the naive scheme described above. Implement this in a class called `TextBuffer`. Your `TextBuffer` class should contain the following constructor and methods:

- `TextBuffer()`: creates a new, empty text buffer.
- `void move(n)`: moves the cursor $n$ characters forward in the buffer (an integer); a negative $n$ causes the cursor to move backwards
- `void insert(ch)`: inserts the given character at the cursor and advances the cursor (operates in O(1) amortized time)

The following methods of the ArrayList class will be helpful in your implementation.

- `void add(v)`: adds $v$ to the end of the array list.
- `T remove(index)`: removes the element at the given index from the array list.