

CMSC420 Project - Spring 2005

Version 1.0

The BIG 420 Project*
Part 1 will be due on February TBD at 10pm
Submission instructions will be forthcoming.

Last Modified January 31, 2005

1 General Project Information

A substantial portion of your grade in this course will be determined by your performance on several programming assignments (collectively the project). The time commitment required on your part varies from student to student, but expect to spend a good bit of time planning as well as coding and debugging each part of the projects.

Despite the quantity of work required to successfully implement these projects, most students, when finished with CMSC420, agree that the experience made them better programmers. All of your projects this semester will be implemented in the Java programming language. It is possible to go from having almost no Java knowledge to adding 'professional-level Java developer' to your resume in a single semester by a single course. Not only will you learn to exploit the Java API data structures you'll be implementing, we'll be providing some tips, tricks, and general advice about how to write not only good Java code but how to develop good object-oriented design in general. So, don't let the programming component of the course scare you off; it will be worth it, we promise.

The project will consist of several major parts (enumerated by dates on which you need to turn things in). If the semester goes as planned, the project will be comprised of four parts. This number may be reduced or increased as Dr. Hugue sees fit. Excluding mitigating circumstances, count on having four major parts.

Each part will typically involve two major components: coding a functional data structure and actually putting it to use. For example, in the past we have had students implement a Fibonacci heap, and use it as the priority queue component of Dijkstra's algorithm. The first project will require that you write what has come to be known as a command parser, which will allow us to pass input to your program as a series of commands (e.g., add (x,y) to your B+ tree). Each project will introduce a new set of commands you will need to handle, so updating your command parser will be another component of each of your projects.

We will test your projects in a variety of ways. We will test your command parser with a series of inputs and do a standard diff with your results (similar to the way your code has been tested in 214). Because we are dealing primarily with data structures in this course, the asymptotic complexity of your algorithms is of paramount interest to us when testing your code, and we will test this by passing very large inputs to your code and timing it, so unlike your previous courses, the efficiency of your code will matter here. Because we will be using Java, we will not be providing header files (with which we can test each method independently with a driver as done in 214). Instead, we will be asking that many of your data structures inherit from common Java classes (the actual mechanism we will use is to have you implement interfaces). In this way, we can test individual methods for correctness.

*Participation in this project may prove HAZARDOUS to your health. Unfortunately, failure to participate early and often will definitely adversely affect your GPA. Take my advice. Start now, because you're already behind. If you don't believe me, ask anyone who took this CMSC 420 with Dr. Hugue.

Ideally, we will provide you with the official primary input/output pairs before the project is due. We will also attempt to provide the code which will test your individual methods as described above. Our policy in this course is that there is no reason that every student shouldn't be able to get an A on the project, and we will make every attempt to give you, the students, as much of our testing material as possible before the project is due so you'll pretty much know what your project grade is before you even submit.

Projects will be graded on a point system, whereby points will be awarded for each test successfully passed. As you will soon see when you begin implementing your first data structure, we will only ask that you implement a fraction of the functionality described by the Java API for the interfaces we are going to ask you to implement. However, we believe strongly in awarding students who go above and beyond the requirements of a project and therefore our testing will be far more comprehensive than necessary. In other words, there will probably be many, many extra credit tests that we run on your projects. Don't get too excited extra credit will help boost your project grade but it won't necessarily net you more than a 100 students who did remarkably well on the project. This will help Dr. Hugue write you a sparkling letter of recommendation for graduate school or employment. A side effect of having a lot of extra credit tests is that the possible total project score will probably be an order of magnitude greater than you'll need to get an A on the project. So there may be 200 possible points on a given project but 80 of them will net you the A. (Dr. Hugue's exams are typically the same way). Another advantage of this approach is that you students will have varying levels of programming and educational experience, and so the extra credit will be enough to challenge the higher-level students without frustrating everyone else.

Unlike many other courses at this university, we in CMSC420 believe that you should be free to talk at length about the project with each other. The algorithms you'll be writing to implement data structures are not secrets; in most cases you will have pseudocode or even actual code available to you through many resources. The newsgroup (csd.cmsc420) is mandatory reading in this course and is the perfect place to ask questions and get answers about the project, Java, the data structures, the meaning of life, etc. Dr. Hugue and the TAs regularly read the newsgroup and we try to answer all outstanding questions that haven't been correctly addressed by other students (or at all). Do note, however, that even though our policy is more lenient than other professors in their courses, we do take academic honesty quite seriously and students have received XF's in this course for blatantly copying other students' code. It's one thing to develop an algorithm with another student it's another to copy and paste your buddy's code. We will still be doing the standard code comparisons not only between your code but also the code of students of semesters past (yes, we keep it). So don't try any funny stuff and you'll be fine. The general rule of thumb: when in doubt, cite your resource via in-code comments and notify the professor and ask permission.

The last important matter of business is to mention that all of your input and output for your command parser will be in XML (eXtensible Markup Language). If you've never heard of XML, ask your friend Google about it and you'll be in no short supply of information. XML offers a couple of major advantages to a standard text-based command parser; the first is that it is largely more relevant to a data structures course since the structure of XML is a general tree. The second is that XML is popping up all over the IT industry, and chances are that you will be dealing with XML at your place of employment. It's becoming the new standard for information exchange, so it's a good thing to be learning. Another great thing about XML is that, in the past, students were required to error-check the commands. Dr. Hugue is a dependability expert and she believes strongly in writing dependable code; a malformed text command should not crash your program. XML is easily validated (both syntactically and structurally), so you can use pre-existing and readily available tools to confirm that both the input and output XML is error-free.

Java, at this point, contains only interfaces for XML processing, and some of those interfaces are acknowledged to be incomplete. So Java doesn't come stocked with classes to process Java documents. But here's the good news: Evan Machusak, a former TA and author of a large portion of the project specifications, has written a (mostly) full implementation of Java's DOM interfaces (the interfaces in the `org.w3c.dom` package) which you're free to use (though aren't required to). This includes an XML parser and an XML validator, which, when used will simplify the amount of work you'll need to do to get an XML-based command parser up and running. Of course, if you want to do it the hard way, please, be our guest.

1.1 Disclaimer

No programmer is perfect. No guarantee is made that bugs do not exist in the code that is provided. However, most of the code has been tested extensively; so, check your code as well as any code that is supplied. Reporting and, perhaps, correcting bugs in supplied code is just one way to improve the quality of the course and earn class participation points.

2 General notes on Java

We assume merely a minimal understanding of the Java language; so, don't fret that your less-than-expert knowledge of Java will preclude your passing the course, much less excelling in it. We will provide detailed explanations of the Java concepts and constructs to be used throughout the semester, and you should be extremely comfortable in Java by semester's end. The syntax of Java is very similar to C++, but there are a few important distinctions to be aware of before jumping into data structure development with both feet.

2.1 IDEs

We **strongly** encourage you to use an IDE to develop your project code. Although you could develop this project using only emacs and a JAVA compiler and virtual machine, it is in your best interest to use an integrated development environment (IDE). An IDE allows you to write, compile, test, debug, and run your program without having to go to the command line (or a shell in emacs). A good IDE is one that helps you find compilation errors and allows you to debug your program by stepping through it line-by-line while displaying a print out of all local variables.

Many JAVA IDEs are available. Try out a few and find out that works for you. Some potential IDEs include but are not limited to:

- Eclipse [<http://www.eclipse.org/>]
- JCreator [<http://www.jcreator.com>]
- Dr. Java [<http://drjava.sourceforge.net/>]
- jbuilder [<http://www.borland.com/jbuilder/>] (free but registration required)
- NetBeans [<http://www.sun.com/software/sundev/jde/index.html>]
- SunOne [www.sun.com/sunone/] (Community Edition is a free download from Sun)

Do a Google search to find the URLs to download these IDEs or look for them on the WAM machines (I have no idea which of these are installed on the UMD networks, other than Eclipse and Dr. Java). If you find another IDE which you like, post it to the newsgroup to earn class participation points and allow others to share in your wisdom at the same time.

You are permitted to use any JAVA drawing facility with which you are comfortable. However, It is this package that will be most readily supported by the TA's should any problems arise. The package 'Canvas.java' provides a simple class which allows drawing of circles , squares, lines, captions, and other simple primitives in a java JFrame. While this isn't being used in part 1, it will show up in the not too distant future so you may want to take a peek at it.

2.2 Pass by reference, but not really

Every semester a new group of students gets caught up by the same thing in Java. They start out hearing "Java is always pass by reference" and they do silly looking things like the following:

```
void foo(String t) { t = new String("World"); }

String s = new String("Hello"); foo(s);

System.out.print(s); //prints "Hello". Why didn't it change?
```

In true pass by reference C++ this would have worked. But what is happening is not really pass by reference, it is pass by value, except what is being passed is a pointer. If you were to transfer the above to C++ it would look like:

```
void foo(String *t) { t = new String("World"); }

String *s = new String("Hello"); foo(s);

cout<<*s<<endl; //prints "Hello". Hopefully obvious why
```

You can see in the second example that t is only a local copy of s. If you alter the value t is pointing at then s will see the change. However, if you point t at something else s will never know. In this example there is actually no way for foo to change s, since java Strings are immutable after creation. An error less obvious than the above is:

```
void foo(String t) { t = t+"World"; }
```

This looks like concatenation, not reallocation, but that '+' operator actually allocates a new String. The above is actually just a shortcut in Java for:

```
void foo(String t) { StringBuffer temp = new StringBuffer();
    temp.append(t); temp.append("World"); t = temp.toString(); }
```

It's important to realize what's going on in the background! Of course in the above example, foo still doesn't change t, but what you could do instead is:

```
void foo(StringBuffer t) { t.append("World"); }
```

This time, since t always points to the same location, the original value really is modified. In java, "pass by reference" as C++ programmers tend to think of it always requires some kind of wrapper. In the last example, StringBuffer is a wrapper for a dynamically sized character tabular. There is a quick and dirty hack to get a similar effect without building an entire class wrapper, pass a 1 element tabular instead:

```
void foo(String[] t) { t[0] = new String("World"); }

String s[]=new String[1]; s[0] = new String("Hello"); foo(s); //s[0]=
"World"
```

This works for a similar reason. t points to the same tabular in memory that s does. When an element of the tabular is updated by t, s will see the change as well. This ends my FYI on pass by reference, try not to get caught up by this common error :)

2.3 A few notes about Java's object oriented design

In C++, inheritance and polymorphism could be considered practically an afterthought by comparison to languages like Java and C

#

which were designed as object oriented from the ground up. In Java, you never write programs, you write classes (objects). It's not possible to write a function that is not encapsulated as part of a class in other words, all functions in Java are more appropriately methods. Writing Java code that isn't written with careful attention to its design as an object is a crime against the language; no more should you write C style code in Java than you should write Lisp style code in C. It's sufficient to remember that when you're writing Java code, you're writing OO code. Don't forget that. Following is a brief outline of a few of Java's OO features that differ from C++.

In C++, to inherit from any other class, the syntax looks like this:

```
class Foo : Parent1, Parent2 { ... };
```

In Java, the colon is subsumed by the keyword **extends**. Also note that every class-level and package-level declaration requires an access modifier, so an entire Java class must be marked as being either public, protected, or private. So in Java, the syntax for inheritance looks like this:

```
public class Foo extends Parent1 { ... }
```

Java does not allow multiple inheritance in this manner. Every class (except Object) has exactly one parent. If a class is not defined to explicitly extend some other class, by default it extends Object. Every class you write in Java will extend Object. Clearly Object is the only class in the Java language that does not have a parent class.

However, Java does allow multiple inheritance, but that inheritance is restricted. Java introduces the concept of interfaces. An interface could be defined as being a class which may contain only public abstract methods and static fields of any kind. (In Java, "abstract" means the same thing as "pure virtual" in C++ speak; "static" means the same thing as in C++). A class is free to inherit from as many interfaces as it wants, however a different key word is used when describing inheritance from an interface:

```
class Foo extends Bar implements Interface1, Interface2 { ... }  
class Foo implements Interface1 { ... }
```

Note that in Java all methods (except static methods) are virtual. This has two consequences: the first is that you are not required to qualify your methods as "virtual" and the second is that you cannot deliberately make a method non-virtual. Static methods are not virtual but they also behave differently and have added restrictions (for instance, you cannot refer to the class's fields inside a static method since the magic "this" pointer does not exist in static methods), so don't use static methods expressly to prevent a method from being treated as virtual. (In case you are rusty on your OO slang, a virtual method in C++ is one whereby if you assign a child class to a parent class pointer and invoke an inherited method on that pointer, the child's method will be executed. A more eloquent description, which uses even more OO jargon, would be to say that a virtual method is invoked based on a variable's actual type irrespective of its declared type).

In Java, methods can be explicitly marked as being "pure virtual" (in C++, a pure virtual method is a virtual method that doesn't have a body). However, recall that in Java the term for "pure virtual" is instead "abstract". If you wish to declare a method "abstract", do so like this:

```
public abstract void foo();
```

If a class contains a single abstract method, the class itself must also be marked abstract. Abstract classes are not allowed to be instantiated (i.e., their constructors are not allowed to be invoked). Abstract classes are still allowed to have constructors since their non-abstract child classes might wish to invoke the parent constructor from within their own constructors. To mark a class abstract:

```
public abstract class Foo { ... }
```

Although the presence of a single abstract method in the body of the class would suggest that the entire class is abstract and therefore marking it explicitly is basically unnecessary, no compliant compiler will allow you to compile a class that contains an abstract method but is not itself declared abstract.

2.4 Comparable, Comparators, and how Java gets by without overloaded operators

In C++, you had the opportunity to overload operators for whichever types you wanted. In 214 you may have implemented a templated binary search tree (BST) that worked with the caveat that the provided type had to have at least its 'less than' operator overloaded. This is fine and it works, but there's no elegant way for you to enforce this invariant in code – the BST would simply break if you passed in a class that didn't have its operator properly defined. Furthermore, suppose your application changes and you want to sort your data elements in a different order. With the stock C++ approach you'd need to modify the data

element's less than operator to reflect the change. But what happens if your application changes again and you want to have both of the orderings (forward and reverse) available simultaneously? Have two versions of your data element, with different operators? That probably sounds like a terrible solution because it is. Overloaded operators are one feature that many programmers (even those who would call themselves OO competent) claim to miss. But overloaded operators actually cause more problems than they solve, and Java's solution is actually much more elegant and far more modular.

It is true that not all objects can be compared to one another in any meaningful way; some data is simply not sortable. Some data has many different orderings. Some data has one obvious and universally meaningful ordering. Numbers, for instance, typically have a well understood ordering. Strings, too, have an accepted lexicographical ordering that programmers have come to expect. In Java, objects that have one commonly accepted method for comparing themselves against each other are designated to implement the Comparable interface.

The Comparable interface contains one method with the following signature:

```
public int compareTo(Object obj);
```

Note the return type. The API specifies that the int value returned must obey three rules: if the parameter is less than this object, return a number strictly less than zero ($x < 0$). If the parameter is equal to this object, return exactly 0. If the parameter is greater than this object, return a number strictly greater than zero ($x > 0$). Objects that implement this interface can be compared like this:

```
public class Foo implements Comparable { ... }
...
Foo f1 = new Foo();
Foo f2 = new Foo();
int r = f1.compareTo(f2);
if (r < 0) System.out.println("foo1 < foo2");
else if (r == 0) System.out.println("foo1 == foo2");
else System.out.println("foo1 > foo2");
```

Note that in Java, the == operator is unilaterally used for pointer comparison. The == operator will only return true if the two variables refer to *exactly* the same object in memory (i.e., they point to the same location). The most commonly used class that implements Comparable is String. Remember that interfaces are types. The following code segments are legal:

```
Comparable c = new String();
if (c instanceof Comparable) System.out.println("tautology");
int r = ((Comparable)c).compareTo("hello");
```

Be warned, however, that the object versions of the primitives (Integer, Float, Double, Long, Short, Byte, Boolean) do in fact implement the Comparable interface, but they are not Comparable to each other – this has to do with precision issues. You can compare a Float against a Float, but if you try to compare a Float against an Integer, a ClassCastException will be thrown.

Objects that implement the Comparable interface can be inserted into sorted data structures in the API such as TreeMap.

This is fine, but suppose you are given some class for which you lack the ability or the privileges to modify its source to make it implement the Comparable interface, or you are in the situation whereby you wish to impose an alternate ordering to preexisting objects. The Comparable interface does not solve the BST problem suggested previously – you would need to modify the body of String's compareTo() method to sort strings in reverse alphabetical order. In this case, you can't modify String or TreeMap, so how could you put Strings into a TreeMap and have them sorted in reverse order? The answer is by introducing a third party object that will do the comparison for you.

A Comparator is an independent class which contains one method which takes two parameters – these two parameters are the two objects being compared. Where the compareTo() method in Comparable always used the invocation target as one of the two objects being compared (i.e., the object on the left of the .), a comparator takes both sides of the comparison as parameters. In Java, comparators are defined by the Comparator interface, which contains a compare method with this signature:

```
public int compare(Object o1, Object o2);
```

A typical Comparator class will look like this:

```
public class MyComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        // code to compare o1 against o2 goes here
    }
}
```

Observe that to be as generic as possible, the parameters that the compare method takes are both Objects, but typically a comparator is designated for a specific type. Usually the first thing the compare() method will do is cast the parameters into the target types. For instance, let's write a comparator for Strings that imposes reverse ordering:

```
public class StringReverseComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        int r = s1.compareTo(s2);
        if (r < 0) return Math.abs(r);
        else if (r == 0) return 0;
        else return -r;
    }
}
```

It's easy to see that this code can be reduced: all we're doing is swapping the sign of the result of the compareTo() method for the two Strings. A better version would be:

```
public class StringReverseComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return -s1.compareTo(s2);
    }
}
```

However, to be even more generic, we can observe that this code would work for any two objects that implement Comparable. In fact, at a minimum, it would work for any two objects as long as the first parameter implements Comparable. (By work, I mean execute without throwing an exception). Granted, o1 and o2 should probably not just be Comparable but also be the same type. We can check that programmatically, but for brevity's sake let's trust the user (cardinal rule in software engineering: never trust the user). A still better version:

```
public class ReverseComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        return -((Comparable)o1).compareTo(o2);
    }
}
```

We can still make another improvement. Here, we are reversing the ordering imposed by the compareTo() method only for objects that implement Comparable. But remember the entire reason for inventing Comparators in the first place – not all objects are comparable! We might want to reverse an ordering imposed by another comparator. Thus, what we really want to do is compose our ReverseComparator out of another comparator. Consider this implementation:

```

public class ReverseComparator implements Comparator {
    private Comparator comp;
    public ReverseComparator() { this(null); }
    public ReverseComparator(Comparator comp) { this.comp = comp; }
    public int compare(Object o1, Object o2) {
        if (comp != null) return -comp.compare(o1,o2);
        else return -((Comparable)o1).compareTo(o2);
    }
}

```

Now we have a comparator that can handle both situations – if you create a `ReverseComparator` based off some other `Comparator`, the result will be the reversal of that comparator’s ordering. Otherwise, if you pass in nothing (or `null`), it will assume that the objects you pass it implement `Comparable`. (If they don’t, a `ClassCastException` will be thrown).

Now that you see how a `Comparator` is implemented, consider how a `TreeMap` could use one. Taking a look at the docs for `TreeMap`, you will notice that `TreeMap` has a constructor which takes a `Comparator` as a parameter. By passing in an instance of our `ReverseComparator`, `TreeMap` will use that comparator whenever it has to make a comparison between two objects. It will use this comparator to figure out where in the tree each object belongs. `TreeMap` will sort a given object `x` as first in its ordering if, for all other objects `y` in the map, `comp.compare(x,y) < 0` where `comp` is the comparator passed in to the `TreeMap` at creation time. Likewise, when searching the `TreeMap` for a given object `x`, the `TreeMap` will report a successful search if it finds some other object `y` in the map such that `comp.compare(x,y) == 0`. You can make `TreeMap` (or any other sorted structure available in Java) do amazing and varied things by altering the comparator you pass into it. For instance, you could easily create a “black hole” object by passing a `TreeMap` a comparator that never returns 0. You could add objects all day long and it would sort them properly – iterating over the objects in the `Map` would produce the proper ordering. But any object you tried to search for would be reported as not existing in the tree because the comparator is incapable of returning 0. The bulk of your first project will be tricking `TreeMap` into behaving like plenty of other things by playing tricks with the `Comparators` you pass it.

3 Part 1: Adjacency List, Priority Queue, and some XML

For the first part of your project, you will implement an adjacency list and a priority queue. You will also need to write an interpreter that will be able to handle basic XML commands. Finally, you will need to implement working versions of two graph-based algorithms: the first to construct one, and the second a well-known favorite, Dijkstra’s algorithm. Don’t panic – it sounds like a lot of work, but it’s actually a lot less than it sounds. This is because your adjacency list and priority queue can both be done merely by playing games with comparators, thereby convincing a good old `TreeMap` or `TreeSet` to act like it’s something else altogether.

3.1 Adjacency List

An adjacency list is a graph representation stored as a list of lists – the first list contains and is indexed by each of the connected vertices in the graph. Each vertex `v` in this list points to another list containing all other vertices connected to `v` by some edge. A list indexed by something other than a number (its offset from the array’s virtual origin in memory) is called many things in different languages: the generic term is “associative array”. In Perl this is called a “hash”. In Java, this is called a `Map`. A `Map` is any structure that associates one object with another. So for your adjacency list, you want a `Map` which associates a named vertex with a list of its neighbors. `Map` is a Java interface which contains methods that accomplish this goal. The folks at Sun have also implemented a guaranteed logarithmic `Map` called `TreeMap` (located in the `java.util` package).

A `Map` is a collection of entries. An entry is a key-value pair. Values are retrieved by searching the map for its corresponding key. The map is sorted by its keys, so for a `TreeMap` which sorts the entries for

logarithmic-time retrieval, the keys must either be Comparable or a Comparator must be provided. For an adjacency list, your keys would be the names of the vertices in the graph and the values would be their corresponding neighbor lists. A map is described as "mapping x to y" where x and y are types; since our vertices will have names, we will represent them with strings. The neighbor lists can really be any collection (a list or a set; edges in our graphs will be unique based on their endpoints, so we will never have more than one edge connecting two vertices x and y). So our adjacency list is really just a map which maps Strings to Lists.

Our goal with our adjacency list is that we want to know if an edge (x,y) exists as efficiently as possible. With TreeMap, which is guaranteed to behave logarithmically, given a key k, we can get the value mapped to by k in logarithmic time. However, our adjacency list maps strings to lists. We can get the list of neighbors for a given vertex in logarithmic time, but unless we can also then search that list for the other endpoint in logarithmic time, our overall complexity will still be linear. Remember how I mentioned that our lists are really sets due to uniqueness of edges? Java has another great class for us called TreeSet which is, as you've probably guessed, a guaranteed logarithmic set. So if your adjacency list is built using a TreeMap which maps Strings to TreeSets, the overall complexity of locating an edge in your adjacency list will be $O(\log(v) + \log(e))$. Not bad!

So it turns out you don't really need to write any new structures to implement a logarithmic adjacency list – you just need to use the existing Java structures to your advantage. Your AdjacencyList class really only needs to contain a TreeMap which maps Strings to TreeSets and write a methods that interact with that TreeMap. (Note that as of Java 1.4, there is no formal way, as in C++ templates, to specify that you are creating a TreeMap that specifically maps Strings to TreeSets – that invariant is enforced only by what you put in the map. Java 1.5 addresses this issue). Your adjacency list has no formal requirements about how it is implemented – we will interact with it only via your command parser, so feel free to name the methods anything you want. You are not even required to write an AdjacencyList class if you don't feel the need to write one. The real requirement in place is that you must be able to acquire some collection object containing all vertices adjacent to a given vertex in logarithmic time or better. As long as this requirement is met, you have complete liberty in implementing this however you please.

3.2 Priority Queue

In order to successfully implement Dijkstra's algorithm, you'll need a priority queue. A priority queue is any structure where the smallest object (or largest, depending on how you look at it) can be removed from the queue consistently in (generally) constant time (in other words, if you remove the smallest item but then have to run some logarithmic algorithm in between to put the next smallest object on top of the queue, you are not fulfilling this requirement). Inserts and removals other than the smallest item should be as quick as possible. The best known priority queue is the Fibonacci Heap which offers constant time removal of the smallest item and amortized constant time for the other operations. However, constant time for removal of the smallest item and logarithmic time for other operations is sufficiently fast for a good implementation of Dijkstra's algorithm, although it bloats its complexity a bit. Your goal is to implement such a priority queue.

You can probably already guess that any sorted structure that offers logarithmic adds and removals will suffice. Java doesn't have any sorted lists. Its two sorted structures, TreeMap and TreeSet, have one crippling limitation: they don't allow duplicates. This is a problem, since a priority queue is generally allowed duplicate entries (and our priority queue must accept them as well). However, we know that TreeMap and TreeSet will only report that an object is contained within them if their comparators return 0 when comparing the search item with other items in the collection. Also, we know that for a priority queue, the only item we ever care about is the first (the smallest). So suppose you attached a comparator to a TreeSet that never returned 0. The TreeSet would never be able to detect that you were adding an object that already exists in the set because the comparator never tells the TreeSet that two objects are the same. The result is that you will be able to add duplicates until the cows come home. A nasty side effect, however, is that you won't be able to search for objects in the TreeSet – the TreeSet will always say that the object isn't found.

So what, you're not searching the priority queue, so why do you care? Unfortunately, TreeMap and TreeSet won't remove an object it can't find. You can't tell a TreeMap to remove key k when it can't find

key *k* inside it. This is a small problem. The only thing you really want to do is remove the first item, right? Good news, there's code to handle that:

```
TreeSet s = new TreeSet(new ComparatorThatNeverReturnsZero()); ...
// remove the first item:
Iterator i = s.iterator();
Object o = i.next(); // o is now set to the first item in the set
i.remove();
// the remove method for an iterator causes the last item returned by a call to next() to be removed from
```

A little awkward, but if you pass a `TreeSet` a comparator that inhibits its ability to find objects inside it, you can't remove anything by name, so this is the only show in town.

Now sure, playing a trick with a comparator isn't the whole story. Your priority queue is basically going to be used to store edges. There are a couple of ways you can use this trick. The first is to create an `Edge` class that contains a data field representing its distance. Your comparator that never returns zero will compare these `Edge` objects, except that if the distance fields of two edges are the same, you return either 1 or -1 instead of 0. (If you choose -1, new items added to the set will come before old ones, vice versa if you return 1). You can do this with a `TreeMap` as well (where you map an edge's distance to the edge object itself). If you choose to do it this way, you'll need to use the object versions of the primitives as keys in the map (`TreeMap`'s methods expect `Object`, and `float` is not a subclass of `Object` because it is not a class at all – you'll have to use `Float` instead). Unfortunately `Floats` are not comparable so when you write your comparator that never returns 0, you'll also have to write the extra code to manually compare the `Float`'s values (you can use the `.floatValue()` method and the regular comparison operators for that). Also, to remove the first entry in a map, the code is only slightly different:

```
TreeMap m = new TreeMap(new ComparatorThatNeverReturnsZero()); ...
// remove the first item: Iterator i = m.entrySet().iterator();
Map.Entry me = (Map.Entry)i.next();
Object key = me.getKey();
Object value = me.getValue();
// the remove method for an iterator causes the last item returned by a call to next() to be removed from
i.remove();
```

`Map` does not have an `iterator()` method. You'll need to treat the map as a set (the `entrySet()` method returns a set containing all of the entries in the map). The resulting set contains objects which implement an interface inside `Map`, called `Entry` (thusly qualified as `Map.Entry`). The key would be the `Float` representing the distance and the value would be the edge itself. As you can already tell, the first method is easier.

3.3 A command interpreter for simple XML

If you haven't heard of XML yet, now is a good time to read up. XML stands for extensible markup language and is quickly becoming the standard for textual data representation. If you haven't used XML yet at work or for another class, you will probably see it soon. XML is basically just a tag-based hierarchical organization of data (i.e., a data structure). If you want to look at an XML document through a data structures lens, an XML document is basically a general tree whose nodes are either named tags or blocks of text. A great site with great tutorials that cover the basics of XML is available at: <http://www.w3schools.com>.

Google will also tell you everything you want to know about XML – it's so widespread at this point that there are probably hundreds of tutorials already written, so I'm not going to reinvent the wheel.

For your project, your input and output will be in XML format. XML is convenient because it is designed to be validated (in other words, checked for correctness). Every XML document can optionally include a reference to its DTD (document type definition) which is (usually) an external document that contains the rules for what elements an XML document can contain. In other words, a DTD defines the rules for the XML structure. Thus, by validating an XML document against a DTD, you can quickly determine whether or not the XML contains certain kinds of errors. If you happen to have handy a solid XML parser and a solid DTD validator, you can eliminate the majority of possible input/output errors. Fortunately for you I have written both of those things for you.

At this point I am going to assume you are familiar with XML. In order to test your project, you are going to provide a Java program (i.e., a class with a `main()` method) that will read a series of XML commands from an input stream, process those commands, and generate an XML document containing the results. For simplicity's sake, the input XML will be simple enough to process without using an XML tools and will in fact differ little from the command structure used in semesters past. The XML will simply be a sequence of empty elements whose names indicate the command to issue and whose attributes will provide the extra information, for example:

```
<createDot name="A" x="5" y="25" color="red" />
<createDot name="B" x="10" y="10" color="blue" /> ...
<deleteDot name="A" />
```

And so forth. If you wanted, you could write your own command parser that treats these tags as merely strings with which you can do as you please, as students were required to do in previous semesters. However, because this is XML, it follows all of XML's rules and is free to do whatever it wants when there isn't an XML rule about it. For example, in XML attributes are not required to appear in any order. So for us, the following two commands are identical:

```
<createDot name="X" x="0" y="0" color="yellow" />
<createDot x="0" name="X" color="yellow" y="0" />
```

This will complicate your parsing slightly.

A few things to keep in mind – first, XML is case-sensitive when it comes to element and attribute names, so "createDot" is NOT the same as "cReAtEdOt". Another thing to observe is that there is no way for an XML's DTD to specify what is contained within the quotes as the value of an attribute. So we cannot, for example, automatically enforce the restriction that the `x` and `y` attributes for the `createDot` command must adhere to the regular expression:

```
(-[1-9][0-9]*)
```

so you will have to refer to the spec for the regular expressions associated with certain attributes. Some attributes, like `color`, have only a few legal values. Attributes whose values can be enumerated in a list of legal values can be enforced by a DTD, so you won't have to worry about those. However, in general:

Any attribute whose value is obviously a number (such as `x` and `y` in `createDot`) will typically follow the aforementioned regular expression `(-[1-9][0-9]*)`. In other words, integers. A programmatic description is that you should be able to obtain the integer value of numerical attributes by obtaining their values as strings and passing them through `Integer.parseInt()`. If that method throws an exception when attempting to parse a numerical value, you can assume that attribute's value is invalid.

Any attribute whose value is defined to be a string (such as `name` in `createDot`) will be restricted only to alpha characters for the sake of keeping it simple. This makes them mutually exclusive with numerical attributes, so the regular expression for string attributes will be:

```
([_a-zA-Z]+)
```

(i.e., any combination of upper and lower case letters and the underscore of length greater than or equal to 1).

3.3.1 Using the provided XML processing code to get a working parser

I've provided an XML processing package which implements the DOM (document-object model) API. The DOM is most relevant to data structures since it organizes XML objects into a general tree. XML objects are lots of things, but the three things we care most about are documents, elements, and attributes. In the DOM, every XML object implements the `org.w3c.dom.Node` interface. Nodes have a lot of useful methods, but the ones you care most about are `getNodeName()`, `getNodeValue()`, and `getAttributes()`. Suppose you had a `Node` object representing one of our command elements. You could determine which command it was with the following code:

```

Node command = ...;
if (command.getNodeName().equals("createDot"))
// createDot command

```

To get the values of the attributes, you could use this code:

```
String name = command.getAttributes().getNamedItem("name").getNodeValue();
```

The next logical question is how do we get an input XML file into a collection of Node objects? In the xml package I have provided there is a class called XMLParser which contains a few overloaded static methods called parseDocument() which can take a File or an InputStream as parameters. The result is that, as long as the XML does not contain syntax errors, this method will return an org.w3c.dom.Document object that models the XML document. For our collection of commands, the XML file will be similar to this:

```

<?xml version="1.0" ?>
<commands>
<createDot ... />
...
</commands>

```

The first line is a processing instruction which you don't have to worry about (if you are writing a parser by hand, you can ignore any tag beginning with "?"). Because XML is a tree, it must have exactly one root, so we have to nest the list of commands as child elements of the single root element. Given an org.w3c.dom.Document object, you can acquire the XML's root element by using the getDocumentElement() method. To get a list containing all of the child nodes of this element, use the method getChildNodes() defined in Node. The return type of this method is NodeList, which is basically a typesafe subset of the java.util.List interface. To iterate through the items in this list, the code would look like this:

```

Document d = XMLParser.parseDocument("in.xml");
NodeList nl = d.getDocumentElement().getChildNodes();
for(int i = 0; i < nl.getLength(); ++i) {
Node command = nl.item(i); // process the command here
}

```

The last piece of usefulness is the ability to validate the element against the document's DTD to make sure, for example, that the command is one of the valid commands for this part of the project, that all of the required attributes are present in this element, and so forth. I've written a class called XMLDocumentValidator which contains a method to validate a single node. The method's signature is public boolean valid(Node, XMLDocumentType). Note that the method is not static, so you'll need to have an instance of XMLDocumentValidator hanging around to invoke this method. You have the Node, so where does the XMLDocumentType object come from? So glad you asked!

The syntax for binding a DTD to an XML document looks like this:

```
<!DOCTYPE commands SYSTEM "http://www.cs.umd.edu/~emach/Part1.DTD">
```

This line would appear in the XML file before the first element:

```

<?xml version="1.0" ?>
<!DOCTYPE commands SYSTEM "http://www.cs.umd.edu/~emach/Part1.DTD">
<commands>
<createDot ... />
...
</commands>

```

Note a few things. First, the second token in the DTD line is the same name as the root element of the document. A single DTD file could define multiple document hierarchies; specifying which hierarchy this XML document adheres to is important to disambiguate the DTD. The third token, SYSTEM, is like a reserved word that indicates from where the DTD will come (don't worry about it). The fourth token, in

quotes, indicates where the DTD is located. This is either a path to a local file, or in this case a URL. The provided XMLParser class automatically recognizes both of these, though if you are specifying a URL, it must begin with "http://" or it will be treated like a file). If the XML file passed to the XMLParser contains a DTD reference, the DTD will automatically be acquired from its source (either from a file, or downloaded from the web if it's an http url), automatically parsed, and automatically attached to the resulting org.w3c.dom.Document object.

As a result, if you are using XMLParser to generate a Document object, you will already have the DTD available to you if the XML file has one defined. If not, I highly suggest simply ignoring the DTD definition and hand-coding the error checking. It's probably more trouble than it's worth to write a DTD validator based on strings (as opposed to Nodes). Alternately, you could create the DTD object yourself by using the DTDParse class, acquire the XMLDocumentType object from that parser, and create Node objects on the fly based on strings to validate with the XMLDocumentValidator, although I think that's probably more trouble than it's worth.

If you are working with an org.w3c.dom.Document object, you can get the associated DTD with this code:

```
Document d = XMLParser.parseDocument(System.in);
XMLDocumentType dtd = (XMLDocumentType)d.getDoctype();
```

The cast is important – the getDoctype() method returns an object only of type DocumentType (an interface). XMLDocumentType implements that interface, but in order to do pretty much anything with a DTD, I needed to have access to the actual type since, if you look at the DocumentType interface, it provides far, far too few methods to do anything useful. Thus, my XMLDocumentValidator requires the actual type. But don't worry, the cast will succeed because the only concrete implementation of DocumentType that exists in the xml package is XMLDocumentType.

Once you have the DTD, you can validate an element against the DTD with the following code:

```
Document d = XMLParser.parseDocument("in.xml");
XMLDocumentType dtd = (XMLDocumentType)d.getDoctype();
XMLDocumentValidator validator = new XMLDocumentValidator();
NodeList nl = d.getDocumentElement().getChildNodes();
for(int i = 0; i < nl.getLength(); ++i) {
Node command = nl.item(i);
if (validator.valid(command,dtd) == false) // report an error
else {
// process the command here
}
}
```

However, this approach, though accurate, does not provide all the info we want. It would be nice if we could narrow down exactly what kind of error causes the element to be invalid. XMLDocumentValidator is written to provide this functionality by way of throwing exceptions, but its default behavior is to simply return false if the element is not valid. You can change that behavior by passing in a flag in the constructor of XMLDocumentValidator, like so:

```
XMLDocumentValidator validator = new XMLDocumentValidator(XMLDocumentValidator.FAIL_THROW_EXCEPTION);
```

Now, instead of just returning false, an exception will be thrown when an element is in valid. If uncaught, this exception will halt your program. So, you'll have to catch the exception, and do something with it. The code would look like this:

```
Document d = XMLParser.parseDocument("in.xml"); XMLDocumentType
dtd = (XMLDocumentType)d.getDoctype(); XMLDocumentValidator
validator = ne XMLDocumentValidator(XMLDocumentValidator.FAIL_THROW_EXCEPTION);
NodeList nl = d.getDocumentElement().getChildNodes();
for(int i = 0; i < nl.getLength(); ++i) {
```

```

Node command = nl.item(i);
try {
    validator.valid(command,dtd);
    // process the command here
}
catch(InvalidXMLException e) {
    if (e == InvalidXMLException.UNKNOWN_ELEMENT) // report bad command error
    else if (e == InvalidXMLException.UNKNOWN_ATTRIBUTE) // report bad attribute error
    else if (e == InvalidXMLException.MISSING_REQUIRED_ATTRIBUTE) // report missing attribute error
}
}

```

Observe the use of the == sign. The InvalidXMLException class is not abstract, but its constructors are private. The result of this is that the class can be instantiated, but only from within the class itself. This is a typical implementation of the Singleton design pattern, although there are several InvalidXMLExceptions instantiated instead of just a single one. XMLDocumentValidator will throw one of these pre-existing exceptions instead of instantiating a new exception. Thus, the same object will be thrown multiple times. I can and should use reference comparison for this application because I do want to see that this exact object was thrown. If you want to see the whole list of available exceptions that could be thrown, have a look at the javadocs for the xml package. The three I covered are really the only ones we care about.

There is one deficiency in this implementation: the validator will not provide specific info in its exceptions. In other words, you won't know exactly which required attribute is omitted, just that at least one has been omitted. This could be corrected by making these exceptions mutable, but then the trustworthiness of the details of the exception would be in question since there is nothing forcing the user to set the details before throwing the exception. Because of this issue, we won't ask you to be any more specific about the reason the command is invalid except to say that the command either does not exist, it contains an attribute that is not defined, or it is missing an attribute that is required.

You will, however, need to check the values of the attributes. Remember that validating an element only assures that all the attributes exist, not that their values are appropriate. We will probably only run one test input that contains invalid XML elements to see that you are checking for those events – the majority of our dependability testing will come from providing illegal values for attributes and assuring that you compensate for them. The values of the attributes should match the regular expressions that are appropriate to their type (either alphabetic string or numerical value).

The last bit of testing you'll need to do is contextual or semantic checking – for instance, attempting to create two dots with the same name should result in the second command issuing an error. The full list of error conditions will be listed separately since as the semester progresses new error conditions will be introduced as new commands are introduced as well. This last type of checking will involve interfacing with your dictionaries.

3.3.2 Outputting XML

For the first project, again for the sake of brevity your output will be a series of elements in response to commands, each of which is a reaction to an issued command.

If a command is successfully executed, the result should be a `<success>` tag, which will look roughly like this:

```
<success command="createDot" message="Dot successfully created at (x,y)" />
```

Successfully executed commands may generate more than one line of input. For example, in one mode of operation your command parser will automatically create edges between points as they are added. In that event, the createDot command will generate multiple success entries, for instance: `<success command="createDot" message="Dot successfully created at (x,y)" />` `</i>` `<success command="createDot" message="Edge (A,B) successfully created" />` `</i>`

Also, commands which require that a structure be printed to XML will typically produce tags that are not empty `<success>` or `<error>` tags. For example, a `<printAdjacencyList>` command might produce the following XML:

```

<success command="printAdjacencyList">
<adjacencyList>
<edge start="A" end="B" />
</adjacencyList>
</success>

```

The testing of your messages won't be as stringent as the fact that you reported a success in the first place. More precise information about what the messages for each successful command should be will be posted on the Part 1 Command Specification page.

If a command fails, the result should be an `<error>` tag. Errors are like successes:

```

<error command="createDot" message="Dot already exists at (x,y)" />

```

It is sufficient to merely spit these out to `System.out` (the standard output stream) as they are discovered when processing the commands. However, remember that the result must be a well-formed XML document and a well-formed XML document must have a single root element. The root element for output will be `<results>`. The final XML, at a minimum, should look like this:

```

<results>
<success ... />
<success ... />
<success ... />
<error ... />
<success ... />
...
<success ... />
</results>

```

(You may want also to stick an `<?xml version="1.0" ?>` tag in the front, and possibly a DTD reference, if you want. Neither is required, but both are nice). So in other words, you don't need to worry about building a DOM object, since we'll expect you to just print it to the standard out anyway. We will test your output XML by feeding it into our XMLParser, validating it against our DTD, and then reprinting the DOM object we create, and diffing the results. After we standardize your output by building it into a DOM object and rewriting it to a stream, our XML and your XML should look exactly the same.

3.4 Data dictionaries and notes about Dots and Edges

We have alluded to the fact that you are going to have to maintain a dictionary (a collection) of all the dots we create using the `createDot` command. Generally speaking, you are going to want to be able to access dots by their names in logarithmic time. You can use a `TreeMap` for this, where the keys are the dots' names and the values are the dot objects themselves. This will allow you to get the dots' (x,y) coordinates based on their names very quickly. You also need to be able to find dots based on their (x,y) coordinates, since aside from two dots bearing the same name being illegal, two dots cannot occupy the same (x,y) coordinate. You'll need a way to sort points in 2D space in such a way that they can be stored in a `TreeMap`. Remember reading about comparators?

You can write a comparator that will sort (x,y) coordinates in a useful way: sort on the x coordinate first; if two dots have the same x coordinate, compare their y coordinates to result in their final ordering. A sample ordering:

```

(0,0)
(0,1)
(0,1000)
(1,0)
(1,5)
(100,0)
(100,100)

```

Keep in mind: dots will always have integer coordinates.

So for every dot created, you'll need to add it to two dictionaries: the first maps strings to dots (name to dot object) and the second would map dot objects to strings (names). However, if you write your Dot class such that the dot's name is one of its fields, then using maps doesn't seem to make any sense because the keys contain all the info you need (but keep reading). You could instead use sets. The first set, which sorts by name, uses a `DotNameComparator` which compares two dots just based on the default string comparison between their two names, and the second which uses a `DotCoordinateComparator` which compares two dots based on the (x,y) ordering discussed above. However, there is one minor problem with using sets: once you put something into a set, you can't easily get it out again based on some other data type. In other words, suppose I pass you a string `s`, who I claim is the name of a dot in your set. If you modified your comparator to allow strings to be passed to the `compare()` method, the best you could do is tell me that the dot with that name either exists or doesn't exist – you can't get the actual dot object back out of your set in better than linear time. So at least for the names, you definitely want to use a Map despite the fact that a Set could sort them for you. The coordinate dictionary will probably only be used to check containment, so in that case a set is probably sufficient.

Here I go again alluding to this mysterious Dot object. Yeah, the major data type we'll deal with throughout this project is a colored, named point in 2D space which we will be calling a Dot. Yes, you will have to write some class to store this information. A subjugate data type which you will probably also need to implement is a line segment in 2D space, which we're calling an Edge. You'll have to do geometric computations involving edges and dots (specifically distance). Boo. Fortunately, Java has already done that for you. If you look at the `java.awt.geom` package, you'll see two useful classes: `Point2D` and `Line2D`. Both of these are abstract classes, but each contains two inner classes that are concrete implementations of their enclosing classes: `Point2D.Float`, `Point2D.Double`, `Line2D.Float`, and `Line2D.Double`. As you've guessed, the `.Float` and `.Double` refer to the precision in which their coordinates are stored. There's no `Point2D.Integer`, but you can just use the `Float` version. We'll never pass you a point whose coordinates have more than 23 significant digits (thus subjecting you to a precision error due to the limitations of `Float`).

The best way to implement a Dot is to have your class extend `Point2D.Float`, and add data members (strings) for name and color. Note that `Point2D` defines two public fields, `x` and `y`, which store the coordinate data. Note: do NOT redefine an `x` and `y` in your Dot class if you extend `Point2D.Float`. Most Java compilers will allow you to create fields with the same names as fields in the parent class, but good editors like Eclipse will warn you about it. The problem w/ith redefining your own fields is that the `Point2D` class has already implemented all the geometric computations you need to worry about, but those methods use the `x` and `y` as defined in the `Line2D.Float` class. If you redefine `x` and `y` in Dot and fail to set the `x` and `y` in the parent class (by saying `super.x =` and `super.y =`) your geometric computations will never work because all your points will be treated like (0.0f,0.0f). The same goes for `Line2D` and its `x1,y1,x2,y2` fields. (You may find it useful to create two Dot fields in your Edge class – pointers to the two endpoint objects – so you can figure out the edge's name based on its two endpoints). Remember that all fields (except for primitives) in Java are references (i.e. pointers).

You should avoid the urge to make your Dots or Edges implement the `Comparable` interface since I've already described two obvious ways to sort them and there are probably more. It's better to force your users to provide a `Comparator` than run the risk of them expecting one ordering and discovering another. `Comparators` are quick and easy to write and prevents any possible confusion on how they'll turn out when sorted.

You won't need a data dictionary to store your edges since that's what your adjacency list is for. And I agree with you – an edge object is not really necessary if you implement your adjacency list entirely based on endpoint names. However, you will need edges in later projects when we start shoving them into spatial data structures. Might as well do it now.

3.5 XML command specifications

Herein lies the XML specifications (i.e. DTDs) for the input files you will be provided and for the output files you will be expected to generate. Check these pages regularly as they contain the information both most relevant and most likely to change. Each phase of the project will introduce a new set of commands

your parsers will be expected to handle.

3.5.1 Input

Input will be provided via input redirection. In other words, input files will be passed to your program via the standard input stream (System.in).

The basic structure of the input XML will be a document whose root tag is `<commands>`. `<commands>` will contain, as its children, any series of the other commands. All commands (for part 1 at least) will be single empty elements (recall that an empty element is one that has no children) which contain zero or more attributes. The DTD and sample inputs will probably be sufficient in describing the input format. Of most interest is the success and error conditions for each command and the corresponding messages each should provide. Here's the list:

- **commands** – this is the root element of the XML tree for the input files. It contains attributes that will affect the behavior of your command interpreter. For part 1, the only attribute we have defined is **autoGraph** which has two legal values, **true** and **false**. When set to true, your command interpreter should automatically generate the edges according to the edge creation rules discussed in the algorithms of interest section of the part 1 spec. When set to false, your command interpreter should **not** generate these edges. It is very important that you not only fail to add the edges into your adjacency list but also that you don't both creating them at all when autoGraph is set to false. The reason for this attribute in the first place is that the graph creation algorithm is linear (it requires that you scan all other existing dots each time a new dot is created). When we set autoGraph to false, we are doing so in order to speed test your implementation of Dijkstra's algorithm. The only way we can test the fact that you implemented a logarithmic priority queue is by speed testing your Dijkstra's algorithm on *very* large inputs. Thus, if you are going through the effort of creating the edges (spending linear time) and then simply choosing to add them or not to add them to the adjacency list after you're already created the edges, you are defeating the purpose of this attribute (and the extra work it entails) and guaranteeing yourself a timeout on our Dijkstra's speed tests.
- **createDot** – creates a dot (a vertex) with the specified name, coordinates, radius, and color (the last two attributes will be used in later project parts). A dot can be successfully created if its name is unique (i.e., there isn't another dot already who has the same name) and its coordinates are also unique (i.e., there isn't another dot already who lies at the same (x,y) coordinate). Names are **case-sensitive**. The message for successfully creating a dot is:

```
Dot <name> successfully created at (<x>,<y>)
```

where `<name>`, `<x>`, and `<y>` are the values of the name, x, and y attributes of the `<createDot>` XML element, respectively.

An error is reported for a name collision or a coordinate collision, in that order. In other words, if a dot already exists with both the exact name and coordinates as a pre-existing dot, you should report a name collision error. The error messages for these two events are:

```
Dot <name> already exists  
Dot at (<x>,<y>) already exists
```

If the autoGraph attribute in the `<commands>` tag has a value of "true", this command may also report 0, 1, or 2 more successes or failures. In attempting to automatically create the edges to link this vertex into the graph, those creation events may either succeed or fail (or not occur at all; for the first vertex created, no edges will be created; for the second vertex created, only one edge will be created). When auto-graph creation is enabled, a createDot command should automatically issue the appropriate commands according to the rules of the createEdge command, reporting successes or errors as if a createEdge command had been manually issued.

- **deleteDot** – removes a vertex with the specified name. The criteria for success here is simply that the dot exists. The success message is:

Dot <name> successfully deleted

Observe that this dot may be an endpoint of zero (0) or more edges, created either dynamically or manually. When deleting a dot, you must delete all of the edges that contain it, as though the `deleteEdge` command had been issued for each of these edges. Thus, this command may produce many other `jsuccessi` tags.

An error is reported if the dot with the given name does not exist. The error message is:

Dot <name> does not exist

- **createEdge** – creates an edge between the two dots (vertices) named by the **start** and **end** attributes. Success is reported when both the start and end dots exist in the graph, and an edge between them does not already exist. The message for success is:

Edge (<start>,<end>) successfully created

Note: for this project, we will not attempt to create any edge from a vertex to itself.

Errors should be reported in the following order: start point does not exist, end point does not exist, edge already exists. Only one error element should be produced in the output XML. For an undirected graph, if (A,B) has been created, it is an error to also attempt to create (B,A). The error messages for these three events are as follows:

Start point (<start>) does not exist

End point (<end>) does not exist

Edge (<start>,<end>) already exists

where `jsstarti` and `jsendi` are the values of the **start** and **end** attributes, respectively.

- **deleteEdge** – deletes an edge between the two dots (vertices) named by the **start** and **end** attributes. Success is reported when both the start and end dots exist in the graph, and an edge between them exists. The message for success is:

Edge (<start>,<end>) successfully deleted

Errors should be reported in the following order: start point does not exist, end point does not exist, edge does not exist. Only one error element should be produced in the output XML. The error messages for these three events are as follows:

Start point (<start>) does not exist

End point (<end>) does not exist

Edge (<start>,<end>) does not exist

where `jsstarti` and `jsendi` are the values of the **start** and **end** attributes, respectively.

- **clearAll** – resets all of the structures, clearing them. This has the effect of removing every dot and every edge. This command cannot fail, so it should unilaterally produce a `jsuccessi` element in the output XML. The success message is:

Structures successfully cleared

- **listDots** – prints all dots (vertices) currently present in the graph. This command is only successful if there is at least 1 (1 or more) vertices in the graph. On success, a non-empty `jsuccessi` tag will be produced in the output XML. This `jsuccessi` tag will have no **message** attribute. The resulting `jsuccessi` tag will have one child element, `jsdotListi`. A `jsdotListi` has one or more child elements of type `jsdoti`. A sample `jsuccessi` for this command:

```

<success command="listDots">
<dotList>
<dot name="A" x="420" y="2004" color="blue" radius="15"/>
</dotList>
</success>

```

The order in which the attributes for the `<dot>` tags are listed is unimportant. However, the dot tags themselves must be listed in ascending order either by name or by coordinate, as per the **sortBy** attribute in the **listDots** command, whose two legal values are **name** and **coordinate**. The ordering by name is asciibetical according to the `java.lang.String.comapreTo()` method, and the ordering by coordinate is discussed in the part 1 spec. To reiterate, coordinate ordering is done by comparing x values first; for dots with the same x value, one dot is less than another dot if its y value is less than the other.

An error is reported when the dot list would be empty (i.e., there are no dots). In this event, an `<error>` tag should be produced with the following message:

```
No dots exist to list
```

- **printAdjacencyList** – lists all edges currently present in the adjacency list. A non-empty `<success>` tag with no **message** attribute will be produced on success. The criteria for success is that at least 1 edge exists in the adjacency list. The `<success>` tag will have one child element, `<adjacencyList>`, which will have one or more `<edge>` child elements. The `<edge>` element has two attributes, **start** and **end** which refer to the start and end dots of the edge by name. The edges should be printed in asciibetical order of the names according to `java.lang.String.compareTo()`, with the rule that edges should be compared first by the names of their start points. For two edges with the same starting dot, an edge is less than another edge if its endpoint is asciibetically less than the other edge.

Note: for the sake of simplicity and generality, although your graph is undirected, you should print both the edge (A,B) and the edge (B,A) when printing your adjacency list.

An error is reported if the adjacency list is empty (i.e., there are no edges to list). This event will produce an `<error>` tag with the following message:

```
The adjacency list is empty
```

- **shortestPath** – prints the shortest path between the dots named by the **start** and **end** attributes, respectively. Success is reported if such a path exists. (Generally we will only test this on connected graphs; we may test on disconnected graphs for extra credit, however). The `<success>` element in response to this command is empty and has no **message** attribute. It has one child element, `<path>`, which itself has a number (at least one) `<edge>` elements as children. The `<path>` element has two attributes which must be set: **length** which reports the length of the total path rounded to 3 decimal places, and **hops** which indicates the number of unique **edges** travelled to get from the start to the end. A sample of a `<success>` tag for `<shortestPath>`:

```

<success command="shortestPath">
<path length="10.000" hops="3">
<edge start="A" end="B" />
<edge start="B" end="C" />
<edge start="C" end="D" />
</path>
</success>

```

Note that when implementing Dijkstra's, when two vertices have the same priority in the priority queue, you should dequeue the vertex whose name comes first asciibetically. Thus, in the event that

two or more shortest paths exist, the path that is reported is the path that travels to the vertices with lesser asciibetical names.

An error is reported in the following events: start point does not exist, end point does not exist, and no path exists between start and end. Errors should be reported in that order, but only one error should be produced. The error messages for these events are as follows, respectively:

```
Start point (<start>) does not exist
End point (<end>) does not exist
No path from <start> to <end> exists
```

Input DTD

```
<!ELEMENT commands (createDot|createEdge|deleteDot|deleteEdge|clearAll|listDots|shortestPath|printAdjac
<!ATTLIST commands
autoGraph (true|false) #REQUIRED
>

<!ELEMENT createDot EMPTY>
<!ATTLIST createDot
name CDATA #REQUIRED
x CDATA #REQUIRED
y CDATA #REQUIRED
radius CDATA #REQUIRED
color (red|green|blue|yellow|purple|orange|black) #REQUIRED
>

<!ELEMENT createEdge EMPTY>
<!ATTLIST createEdge
start CDATA #REQUIRED
end CDATA #REQUIRED
>

<!ELEMENT deleteDot EMPTY>
<!ATTLIST deleteDot
name CDATA #REQUIRED
>

<!ELEMENT deleteEdge EMPTY>
<!ATTLIST deleteEdge
start CDATA #REQUIRED
end CDATA #REQUIRED
>

<!ELEMENT clearAll EMPTY>

<!ELEMENT listDots EMPTY>
<!ATTLIST listDots
sortBy (name|coordinates) #REQUIRED
>

<!ELEMENT shortestPath EMPTY>
<!ATTLIST shortestPath
start CDATA #REQUIRED
end CDATA #REQUIRED
```

>

```
<!ELEMENT printAdjacencyList EMPTY>
```

3.5.2 Part 1 Output and DTD

The basic structure of output will be:

```
<results>
<success ... />
<error ... />
<success ... />
...
<success ... />
<success ... />
</results>
```

For information regarding the output of individual commands, see the input specification. Do note: except for one input set per project part, we will provide you syntactically error-free XML, which means that we will only test your error checking on one set of test inputs. However, we will check the syntactic validity of *every* output file you provide. So, make sure that your output XML can be validated against the DTD we provide for each part, or you are at risk for losing substantial credit.

```
<!ELEMENT results (success|error)*>
```

```
<!ELEMENT success (adjacencyList|dotList|path)?>
<!ATTLIST success
command CDATA #REQUIRED
message CDATA #IMPLIED
>
```

```
<!ELEMENT error EMPTY>
<!ATTLIST error
command CDATA #REQUIRED
message CDATA #REQUIRED
>
```

```
<!ELEMENT adjacencyList (edge+)>
<!ELEMENT dotList (dot+)>
<!ELEMENT path (edge+)>
<!ATTLIST path
length CDATA #REQUIRED
hops CDATA #REQUIRED
>
```

```
<!ELEMENT dot EMPTY>
<!ATTLIST dot
name CDATA #REQUIRED
x CDATA #REQUIRED
y CDATA #REQUIRED
radius CDATA #REQUIRED
color (red|green|blue|yellow|purple|orange|black) #REQUIRED
>
```

```
<!ELEMENT edge EMPTY>
```

```
<!ATTLIST edge
start CDATA #REQUIRED
end CDATA #REQUIRED
>
```

3.6 Algorithms of interest

The two major algorithms you need to implement that will actually interact with the structures we're asking you to implement for this project are as follows:

3.6.1 Edge and graph construction

There are two ways to create edges: the first is to manually specify the endpoints, and the second is to automatically generate them.

The first method will be used to allow us to (more) easily generate specific test cases, and more importantly, to create a series of edges quickly. We will want to stress test your implementation of Dijkstra's algorithm to ensure you are not using a linear time priority queue. In order to do that we'll need to keep the graph construction on par with Dijkstra's (generally logarithmic with respect to V). Thus, there exists a command, `createEdge`, which has two attributes, `start` and `end`. However, issuing this command without disabling your automatic graph creation defeats the purpose of fast edge creation to begin with, since your automatic graph creation will be slow. Thus, the `<commands>` tag (the root of all the command elements) will contain an attribute called `autoGraph`, whose value will be either "true" or "false". When the value is false, you should only create edges via the `createEdge` command.

However, when true, you will need to dynamically create edges each time a new vertex is added to the graph (via the `createDot`) command. For each dot added, you must search your data dictionary to find the dot that is geometrically closest in space to the new dot being added and the dot that is geometrically farthest away. You will create an edge between the new dot and the new dot's closest neighbor and also an edge between the new dot and the new dot's farthest neighbor. Nothing to it, right? Don't worry about the fact that there isn't really a good algorithm to do this. You're going to have to brute force your dictionary for each new dot created, so your running time is in the worst case $O(n^2)$. In the future, when you've implemented a better spatial data structure, you'll be able to do this with improved asymptotic complexity bounds, so don't worry about it for now.

3.6.2 Dijkstra's algorithm

This one just keeps coming back to haunt you, doesn't it? As the mastermind of this generation of projects once said, "you should not be awarded a diploma unless you can adequately explain how Dijkstra's works." Fortunately, I haven't graduated yet. Just Google it. A good resource for Dijkstra's algorithm for Java is here:

<http://renaud.waldura.com/doc/java/dijkstra/>

Some students may have correctly implemented an efficient shortest/fastest path algorithm in the past. Others have implemented it, but have not made it efficient. Therefore, in order to support shortest path calculations on large data sets within a reasonable amount of time, you will be required to run your shortest path algorithm in $O(E \log V)$ time. In order to achieve this, you may need to restructure parts of your adjacency list. Note that, in general, insertion/deletion from this structure is $O(\log(n) + \log(m))$, where n is the number of nodes in the graph and m is the degree of the graph (the max number of edges incident to a single vertex). This represents a binary search to find the correct row of the list to find the starting vertex, followed by a binary search to check for existence of the ending vertex.

Recall that in general, graphs are traversed in **depth first order** by expanding a start node, PUSHing all the kids onto a stack, and choosing the next node to expand by POPing from the stack. Similarly, traversing a graph in **breadth first order** is performed by inserting the children of a node into a FIFO queue, and choosing the next node by dequeuing the first element. Furthermore, **greedy** algorithms attempt

to produce a global minimum (or maximum) value by choosing the locally optimal value at every step, and not all greedy algorithms actually converge.

Fortunately, Dijkstra's algorithm is a **convergent greedy algorithm** which finds minimum length (cost, or weight as appropriate) paths from a given start vertex to all possible destination vertices for graphs with positive edge weights. Since the algorithm converges, if one is only interested in the minimum cost to a single destination, it may be possible to stop the algorithm early. However, it will be possible to pass the stress test without making this type of an optimization. Note that Dijkstra's algorithm visits or expands vertices (our loci) in priority order, where the priority for our project is the weight.

To implement an efficient Dijkstra's algorithm you will need a priority queue, which the JAVA API sadly does not provide. This can be implemented through clever use of a TreeSet/Map and Comparators (the problem with a plain set is that you can't have two elements with the same priority, so you need a way to always break ties) to implement a binary heap (the mere *heap* of CMSC 351). However, since we had the Fibonacci heap in Part 1, everyone has access to at least one working F-heap program. So, while you can receive partial credit for a binary heap, full credit for the stress test will require an F-heap as implemented for Part 1.

You'll use the graph to implement shortest path using Dijkstra's algorithm. Using your handy Fibonacci Heap, the run time of this algorithm is (amortized) $O(V \lg(V) + E)$ (where E is the number of edges and V is the number of vertices). It would be nice if you understand where this bound comes from. We've invited Kevin Conroy, 2004 Dorfman Research Prize winner, to explain:

Allow me to sketch the algorithm to explain the running time (I'm not trying to teach the algorithm here- see your book/class/newsgroup/google). Every iteration of this algorithm you are guaranteed to find the correct shortest path to exactly one node- so we know right away there will be V iterations. At each state your graph is split in two sections- the 'solved' section, for which you know the correct distances, and the rest, which have some distance values associated with them which may or may not be accurate- this set is stored in some kind of priority queue. An iteration begins by selecting the node (call it 'N') from this queue with the best distance value, adding it to the 'solved' set, and 'relaxing' all it's edges. Relaxing is when for each node adjacent to N we see if it is faster to get to that node through N than it's current best known path. We update distances and back-pointers appropriately (so we know what the shortest path actually is when we finish), and that ends the round. Note that if a node's distance value is changed, its position in the priority queue has to be fixed up somehow. (this is where the magical Fibonacci heap would come into play, it's got an advantage in this 'fix up' step). One way to do this is just to have multiple copies of the same node in the queue and ignore them when they come back up (this is the approach I will use in the explanation), or else to remove the old value before reinserting it. Either works.

Now, how long does all this take. There are V rounds, and in every round we have to pull something out of the front of the priority queue using the F-heap is amortized $\Theta(1)$. Rather than try and deal with how many elements are added to and removed from the queue in any single round, it is easier to think about how many such operations can occur in the life of the algorithm. Every single edge in the graph has exactly one opportunity to add a vertex to the queue (during a relax operation), so there are $O(E)$ possible insertions. If we allow duplicates, the size of the queue can grow to $O(E)$, so we may treat each That gives (amortized) $\Theta(E)$ running time for all queue operations during the life of the algorithm. Together that gives a running time of $O(V \lg E + E)$, which is the required running time of your search.

Please be careful with your implementation details. In particular, *do not* use a linear time priority queue. This nailed a lot of people last semester. We will test your project on **very** large inputs.

4 General Policies

4.1 Submission Instructions

To make your submission file, make a directory and copy all required files into it. Change to that directory and type:

```
tar -cvf part#.tar *
gzip part#.tar
```

To submit type(submit will usually be working at least a few days before the due date);:

```
~mh420001/Bin/submit # part#.tar.gz
```

In all cases '#' represents the number of the project part you are submitting(1,2,3 or 4). The filename is not really important; It is important that the file is in .tar.gz format and that your Main.java and other required files are not in a subfolder of the tar file. Subfolders are allowed, such as the cmisc420 package folder.

You must include the following with every submission: All necessary source files (*.java etc.) to compile your program. A file called README, all upper case, which contains your name, login id, and any information you would like to add.

If you leave out the README your project will fail to submit!

There may also be other per project required files that will be checked by the submit program.

If you find the need to provide a makefile, include it in your submission. We will compile your code by running make as normal. If you don't need a makefile, then we should be able to compile and run your project using the following commands:

```
javac Main.java -classpath xml.jar
java Main -classpath xml.jar
```

If you fail to provide a makefile and these two commands fail for any reason, your project will be considered as the dreaded "DNC" (did not compile).

If you provided a makefile and need to provide command line arguments to the java command other than the default, note this in your README file

No promises are made that I will read your READMEs, but they are useful when problems come up with a project.

Please don't include .class files in your submission.

I will grade every submission that is saved (including applicable bonuses and penalties) and you will get the highest grade among them

If there are any errors in my IO you are still responsible for them- the spec is what is in charge. So if you match all my current IO and submit early and then someone points out that I printed the wrong error message for some function, you have to fix your project and resubmit ;) You should be coding to match the command specification, not my sample IO.

4.2 Grading

There will be a few parts to grading your projects Your projects will be graded running them on a number of test files for which I have already generated correct(we hope) output. Your output will have all punctuation, blank lines, and non-newline whitespace stripped before diffing similarly cleaned files.

Some of your data structures may be included with the TAs own testing code to test their efficiency and correctness.

Some text output cannot always be graded by simply diffing because there is no guarantee that we will have the same output. In these cases your project's output will be pre-processed. In the case of the B+ tree, for instance, this program will verify that each node has the correct number of keys, that they are correctly ordered, and that all the correct data is at the leaves(and any other rules I may have left out).

Thanks to the miracle of automation you should expect your projects to be run on very very large inputs.

Typically, each test file will be worth 10 points, and you will be eligible for either 10 or 0 points depending on whether you pass or fail that test. There is no partial credit for an individual test. Only rarely will points be awarded for projects that fail a test because of 'small' errors after initial grading; this will be at the discretion of the ta. The tests will try to test mutually exclusive components of your projects independently. However, if you don't have a dictionary which at least correctly stores all points, and some 'get lost', you may still end up failing other tests since they all require a working dictionary. This does not reflect double jeopardy on the part of the test data, since it is always possible to use some other structure (such as the JAVA tree map) for the data dictionary, and receive points for portions of the tests that merely reference the data in the dictionary.

4.3 Standard Disclaimer: Right to Fail (twice for emphasis!)

As with most programming courses, the instructor reserves the right to fail any student who does not make a good faith effort to complete the project.

If you have problems with completing any given part of the project please talk to Dr. Hugue immediately- do not put it off! While some may enjoy failing students, Dr. Hugue does not; so please be kind and do the project, or ask for advice immediately if you find yourself unable to submit the first two parts of the project in a timely manor. A submission that gets only 20 or 30 points is considerably better for you than no submission at all.

4.4 Integrity Policy

Your work is expected to be your own or to be labeled with its source, whether book or human or web page. Discussion of all parts of the project is permitted and encouraged, including diagrams and flow charts. However, pseudocode writing together is discouraged because it's too close to writing the code together for anyone to be able to tell the difference.

Since the projects are interrelated, and double jeopardy is not our goal, we have a very liberal code use and reuse policy.

- In general, any resources that are accessed in producing your code should be documented within the code and in a README file that should be included in each submission of your project.
- First and foremost, use of code produced by anyone who is taking or has ever taken CMSC 420 from Dr. Hugue requires email from provider and user to be sent to Dr. Hugue. That means that any student who wants to share portions of an earlier part of the project with anyone must inform Dr. Hugue and receive approval for code sharing prior to releasing or receiving said code.
- Second, since we recognize that the ability to modify code written by others is an essential skill for a computer scientist, and that no student should be forced to share code, we will make working versions of critical portions of the project available to all students once grading of each part is completed, or even before, when possible.
- Dr. Hugue is the sole arbiter of code use and reuse, and reserves the right to fail any student who does not make a good faith effort on the project. Violators of the policies stated herein will be referred to the Honor Council.

Remember, it is better to ask and feel silly, than not to ask and receive a complimentary F or XF.

4.4.1 Code Sharing Policy

During the semester we may provide you with working solutions to complete portions of the project. It is legal to look at these solutions, adopt pieces of them, and replace any part of your project with anything from them so long as you indicate that you ACCESSED this code in your README.

Furthermore, any portion of your code that contains any portion of the distributed work should contain identifying information in the comments. That is, note which distributed solution your code is based in the

file where it was used. It is a good idea to wrap shared code with comments such as "Start shared code from source XYZ" and "End shared code from source XYZ." You may also use comments such as "Parts of this function/file were based on code from source XYZ." You cannot err by including this information too often.

Failure to properly document use of distributed code in your project could result in a violation of the honor code. Note which distribution solution(s) your code is based on.