CMSC420 Project - Fall 2004
Version 3.0

The BIG 420 Project*
Part 3 will be due on Sunday, November 28th at 11pm
Last Modified December 7, 2004

1 General Project Information

Like any quality course in data structures, a substantial portion of your grade in this course will be determined by programming assignments (collectively the project) and thus, CMSC420 involves a substantial programming project. You will not only study data structures in the classroom, you will be coding functional versions of many of these structures. The time commitment required on your part varies from student to student, but count on spending a lot of time programming. Some students complain about coding every day. But as Dr. Hugue will point out, you’re a computer science major; what do you think you’ll be doing when you graduate and get a job?

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. This sentiment is capitalized by the fact that 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. In addition to teaching intimately the 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

* 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.
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.

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 psuedocode 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 stock with classes to process Java documents. But here’s the good news I’ve 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. I will guarantee that bugs will exist in the code that is provided. I’ve tested all of the code I will be providing pretty extensively, but there’s a good chance you’ll find bugs. Extra credit will be given to students who can both identify a bug and (if the source is available) offer a solution. The amount is at our discretion and will be based on the severity of the bug. If a bug is so egregious that it affects testing (and therefore affects grading), grades will be corrected.

2 General notes on Java

Your exposure to the Java programming language is not expected to be any more than minimal, so don’t fret if you feel your command of the language is less than expert (if you do, we won’t have done our jobs if that is not remedied in the course of a semester, providing you do your jobs). 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)
- 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 I merely noted the popular IDEs that I have heard of). 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.

While you are permitted to use any JAVA drawing facility you are comfortable with, a simple drawing package is available on the class web page. 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 to distant future so you may want to take a peak at it. A drawing package appropriate for the project can be downloaded from the class webpage.

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 and 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’ 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 Java you may have implemented a templatized 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:

```java
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:

```java
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:

```java
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 infact 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 ascibiect 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:

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

A typical Comparator class will look like this:

```java
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:

```java
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:

```java
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:

```java
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:

```java
private class ReverseComparator implements Comparator {
    private final Comparator comp;
    private 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, \(\text{comp.compare}(x,y) \leq 0\) where \(\text{comp}\) 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 \(\text{comp.compare}(x,y) == 0\). 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, Djikstra’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 blots 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:

```java
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
come 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:

```java
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 seminars 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:

```xml
<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 seminars. 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:

```xml
<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:

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

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

```java
String name = command.getElementsByTagName("name").getFirstChild().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
<?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 getChildren() 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:

```java
Document d = XMLParser.parseDocument("in.xml");
NodeList nl = d.getDocumentElement().getElementsByTagName("name");
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 looks like this:

```xml
<!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 DTDParser 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:

```java
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:

```java
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:

```java
XMLDocumentValidator validator = new XMLDocumentValidator(XMLDocumentValidator.FAIL_THROW_EXCEPTION);
```
Now, instead of just returning false, an exception will be thrown when an element is in invalid. 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 = new XMLDocumentValidator(XMLDocumentValidator.FAIL_THROW_EXCEPTON);
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 output. 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)" />
<success command="createDot" message="Edge (A,B) successfully created" />
` 

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:

```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:

```xml
<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:

```xml
<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 code 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 \( \text{name} \), \( \text{x} \), and \( \text{y} \) are the values of the name, x, and y attributes of the \text{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 `<success>` 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 `start` and `end` 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 `start` and `end` 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 &lt;success&gt; 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 &lt;success&gt; tag will be produced in the output XML. This &lt;success&gt; tag will have no message attribute. The resulting &lt;success&gt; tag will have one child element, &lt;dotList&gt; &lt;/dotList&gt; A &lt;dotList&gt; has one or more child elements of type &lt;dot&gt; A sample &lt;success&gt; for this command:

  &lt;success command="listDots"
  &lt;dotList
  &lt;dot name="A" x="420" y="2004" color="blue" radius="15"/&gt;
  &lt;/dotList
  &lt;/success

The order in which the attributes for the &lt;dot&gt; 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 ascibetical 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 &lt;error&gt; 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 &lt;success&gt; 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 &lt;success&gt; tag will have one child element, &lt;adjacencyList&gt; &lt;/adjacencyList&gt; which will have one or more &lt;edge&gt; child elements. The &lt;edge&gt; 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 ascibetical order of the names according to java.lang.String.comapreTo(), 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 ascibetically 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 &lt;error&gt; 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 &lt;success&gt; element in response to this command is empty and has no message attribute. It has one child element, &lt;path&gt; which itself has a number (at least one) &lt;edge&gt; elements as children. The &lt;path&gt; 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 &lt;success&gt; tag for &lt;shortestPath&gt;:

19
<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|printAdjacent)>  
<!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>
3.5.2 Part 1 Output and DTD

The basic structure of output will be:

```xml
<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, will 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.
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 $O(\log^2 V)$ 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) $O(V^*\log(V)+E)$ (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 $O(V\log E+E)$, 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 Part 2: Spatial Maps, SortedMaps, and the dreaded B+ tree

4.1 SortedMap

You’re already quite familiar with the SortedMap interface – you’ve been using it every time you use a TreeMap, which is Java’s stock implementation of SortedMap. SortedMap is a subinterface of the Map interface, defined in the java.util package as part of the Java Collections Framework. In this project, you are going to get some exposure to implementing a Sun-provided Java interface. This will give you some practical experience coding to a standard and will also teach you leaps and bounds about how to solidly implement an object.

The first thing you’ll need to do is have a look at the API documentation for the Map and SortedMap interfaces. Bookmark these pages now, since you’ll be referring to them again and again throughout this project. You might even find it useful to print out copies. The URLs are:

http://java.sun.com/j2se/1.4.2/docs/api/java/util/Map.html
http://java.sun.com/j2se/1.4.2/docs/api/java/util/SortedMap.html

For this part of the project, you will only need to implement a portion of the SortedMap interface, since we are only going to ask that you write the insert algorithms for your B+ tree. Since the work for coding the B+ tree will be split up among two projects, the work for fully implementing the SortedMap interface will also be divided.

For this project, you are going to be responsible for implementing the following methods:

```java
void clear();
Comparator comparator();
boolean containsKey();
boolean containsValue();
Set entrySet();
boolean equals(Object o);
Object get(Object key);
int hashCode();
boolean isEmpty();
Object put(Object key, Object value);
void putAll(Map t);
int size();
```

Many of these methods are one line, and most of the others are very short also. The majority of the work will be in implementing the put() method, which amounts to writing the algorithms for building a B+ tree. The details of your class are mentioned in the B+ tree portion of the spec (e.g., the name of the class and its required constructors).

For each method that you are not required to implement for this part of the project, the method should immediately throw an UnsupportedOperationException.

There is one method that deserves considerable attention and is the most difficult to implement. It’s vastly important that you understand this concept since it is required for most of the other methods (which you don’t have to implement until the next project). This method is entrySet().

As you can see from the prototype of the method, it returns a Set object. When students see this, they immediately assume that the method is supposed to create a new Set, add all of the Map’s entries into it, and return that Set. That is completely wrong. Observe the API description of the entrySet() method:

```
Returns a set view of the mappings contained in this map. Each element in the returned set is a Map.Entry;
```

vice-versa.
Note that changes in the Set are reflected in the Map and vice versa. Some students will read that and
STILL think that they are supposed to create a new Set, copy all of the elements of the Map into it, return
it, but include some way of managing that relationship (for instance, keep a pointer to the Map in the Set
object, and store a list of all of the entry sets created and returned by that method in the map, updating
all as necessary). **This is also completely wrong** (although it's closer to accurate).

How, then, does one implement this relationship in Java without making copies? The Set object that is
returned from entrySet() does not itself contain any data. The Set that is returned is simply an object
whose methods call the methods in the "backing map" (i.e., the map from which it was created). The
simplest way to implement this behavior is by creating an inner class inside your SortedMap
implementation that itself implements the Set interface. For example:

```java
public class BPTree implements SortedMap {
    public Object remove(Object key) { ... }
    public boolean containsKey(Object key) { ... }
    public Set entrySet() { return new EntrySet(); }

    protected class EntrySet implements Set {
        public boolean remove(Object o) {
            Map.Entry me = (Map.Entry)o; // throws a ClassCastException if this fails, as per the API for Set
            boolean b = BPTree.this.containsKey(me.getKey());
            BPTree.this.remove(me.getKey());
            return b;
        }
        public boolean contains(Object o) {
            Map.Entry me = (Map.Entry)o;
            return BPTree.this.containsKey(me.getKey()) && (me.getValue() == null ? BPTree.this.get(me.getKey()) ==
        }
    }
}
```

Observe that the Set object returned by BPTree’s entrySet() method is of type EntrySet, an inner class of
BPTree, whose remove() method simply calls the BPTree’s remove() method, and containment in the
returned set is checked by existence in the BPTree. Thus, if the BPTree changes AFTER the Set is
created, the contains() method of that Set will still return accurate results. **You must implement
entrySet() to this standard.** We will easily discover whether or not your entrySet() is implemented
correctly. It’s very easy to grade:

```java
SortedMap m = new BPTree();
m.put("Auto","Fail");
Set s = m.entrySet();
m.put("F", "---");
s.add(new Map.Entry("No","Hope"));
if (!s.contains(new Map.Entry("F","---"))) System.out.println("You fail!");
if (!m.get("No").equals("Hope")) System.out.println("You fail!");
```

Note that this wouldn’t compile since Map.Entry is an interface, so you can’t instantiate it. Substitute that
with a concrete implementor of Map.Entry. ;) Also, according to the API, the set returned by a map’s
entrySet() method should throw an UnsupportedOperationException when the add() method is called.
But you get the idea.

**All methods in Set must be implemented by the Set object you return via your entrySet() method.** However, there are certain methods, as detailed in the description of the entrySet() method,
which are explicitly not supported by the entry set. Those include add() and addAll(). A good exam
question might be why the authors of the SortedMap interface (our friend Josh Bloch) made this decision.
There's a good reason – think about it!

25
Unlike the structures you implemented in Part 1, we are going to independently test your implementation of SortedMap programatically (i.e., we’re going to link your BPTree into our code and run a SortedMapTester program on it). Your SortedMap implementation should always agree with TreeMap, thus it is extremely important that your equals() method works correctly; it is equally important that the TreeMap equals() method will work with your B+ tree. Look at TreeMap’s source code, see what it does, and make sure it will be able to check your B+ tree for equality, since we will invoke the equals method bidirectionally. And don’t try to cheat and make your equals() method return true all the time, because we can easily test that too, and we will. Your SortedMap will fail all of the tests if equals() does not work and that is not a valid excuse for a regrade. You have been warned.

Your SortedMap will not support null keys. TreeMap is actually bugged in this regard, in that it will not throw a NullPointerException when a null key is added to the map although it will throw one later, unintentionally. If we attempt to pass null to your B+ tree’s put() method, you should immediately throw a NullPointerException.

Also, RTFAPI. The API is very specific about what exceptions need to be thrown when, what needs to be returned when, etc. Your are expected to throw exactly the exceptions the API defines. This applies both to the methods in the Map/SortedMap interface but also the methods in Set for the object returned by entrySet(). I am imposing another requirement that is commonly done in the Java Collections Framework but not clearly defined in the API, and that refers to the Iterator object returned by a call to entrySet(). In addition to the exceptions described by the API for Iterator (specifically NoSuchElementException if next() is invoked when hasNext() returns false), your iterator must throw a ConcurrentModificationException if the map changes after an Iterator has been created. There is a very easy way to do this – just keep a modCount variable that gets incremented whenever a structure-changing method (i.e., put() and remove()) is called. When you create an iterator, save the modCount according to the Map when the iterator was created, and each time next() is called, check the saved modCount against the modCount of the Map. If the two numbers are not the same, it means the map has changed since the Iterator was created, and at that moment the next() method of the iterator should throw a ConcurrentModificationException.

One more thing: let me teach you anonymous inner classes. If you are looking at other’s code (like Josh’s when you look at TreeMap’s source code) you might see this and be confused. An anonymous inner class is a way of shorthanding the creation of an inner class or the implementation of an interface. The syntax looks like this:

```java
public Set entrySet() {

    return new Set() { 
    public boolean add(Object o) { throw new UnsupportedOperationException(); } 
    };
}
```

Notice two things – first, we are sort of ”instantiating” an interface (we could do the same with an abstract class), but instead of a semicolon we open a curly brace and begin defining its methods. The compiler will not let you compile this code unless you implement all of the methods in Set within the braces (for obvious reasons). What actually happens behind the scenes is that the compiler changes that statement into code that looks like this:

```java
public Set entrySet() { return new AnonymousInnerClass1(); }
public class AnonymousInnerClass1 implements Set {
    public boolean add(Object o) { throw new UnsupportedOperationException(); }
}
```

Pretty handy, huh?

As always, questions to the newsgroup. If you’re lucky, you’ll get sourcecode for a full implementation of SortedMap (in addition to TreeMap’s source code) to study.
4.2 Spatial Maps

Before we go any further...

I’m calling it a map really for lack of a better term – it’s really more like a collection than a map in that it does not map keys to values. In the case of some classes of spatial data structures, what’s really going on is mapping geometry to a region (or some other internal entity). So I’m using the term map, but it’s really more like Java’s Collection than Java’s Map. Got it? Good.

So far we’ve had you do some fun stuff with points and lines (dots and edges, as they’ve been dubbed). But the stuff you’ve done isn’t really that fun (or terribly useful). For instance, suppose we asked you to list all of the points that fall within some given range (for instance, a circle). Or perhaps we said, excluding edges for which the point is an endpoint, find us the closest edge to an arbitrary point (in other words, find us the edge whose distance from some point p is minimal, but strictly greater than zero). Now we ask you to do that in (usually) better than linear time. Could you write the algorithm to solve these problems using the layout of your first project (i.e., using the structures you used for storing dots and edges)?

Hopefully the answer is no, or we’re wasting each other’s time.

If you have any shred of faith left in my credibility after the frequency of very minor errors in the first part of the spec, trust me when I say that the answer is, in fact, no.

Thus, in the next two parts of the spec we’re going to play around with structures that can solve one (or both) of those problems. For this part, because we are also assigning you the insane B+ tree (which in this humble spec writer’s opinion is the algorithmically hardest coding he has done in the twelve upper level CS electives he’s taken), we’re going easy on you in this part. Plus, we’re building on the last part. You’re probably insanely bored of TreeMaps and Comparators by now, but now that you’ve had some practice with them in the minors, it’s time to take your mastery of screwing with pre-existing data structures to your advantage. In addition, all you mathphobes will finally get over your fear since part of this project will involve the formulating of some formulas. Let me go on.

The structure we are going to ask you to implement is something Meesh came up with off the top of her head (or somewhere deeper), and thus it is not published anywhere, except here. The "structure" is really nothing more than a composition of preexisting structures, only this time the organization will be a little different. We are going to use a List to partition space into a series of concentric circles. Each of these concentric circles will have a radius equal to ((n + 1) * r), where n is the 0-based circle index and r is the resolution of the structure (i.e., the width of the rings). The area between the circumferences of adjacent circles form rings. These rings are geometric regions, and points will be stored in these regions.

Furthermore, each region will be subdivided into four subregions, cut along the axes – in this case, we are dealing only with two dimensional data, so each region will be cut by the x- and y- axes for a region centered around (0,0).

Suppose we define the resolution of our structure to be 10, and we define the structure to contain 4 rings (radius = 10, 20, 30, 40) respectively), and we center around the origin (0,0) (so the each region’s subregions correspond to the cartesian quadrants). More formally, for a given ring, its subregions are ordered such that subregion 1 contains all points inside the region and whose x value is less than the center point and whose y value is greater than the center point (in other words, the upper left quadrant from the center), subregion 2 is +x +y from center (upper right), subregion 3 is -x -y (lower left), subregion 4 is +x -y (lower right).

Given the point (25,25), this would fall in the 3rd ring (between r = 20 and r = 30), in the second sub region. The point (-15,15) would fall in the 2nd ring (between r = 10 and r = 20) in the first sub region.

**Regions are closed on all of its boundaries.** This means that the point (0,0) falls in all four subregions of the first region. The point (0,10) falls in both region 1 and region 2, in subregions 1 and 2 of each.

Edges can’t be stored as easily in this structure because, given an edge, the regions that the edge would intersect are difficult to calculate without brute forcing all of the regions. So for this spatial structure, we’ll store only points. Thus, we’re not going to ask you to perform a “nearest segment to point” calculation, but we are going to ask you to do a range search.

Range searches can be performed using a variety of geometry (the two most common are rectangular and circular). We will give you a circle and ask you to tell us all of the points that fall within that circle. Under this organization of space, this operation is fairly quick. I won’t give away the algorithm since that’s one of the benefits of coming up with a structure that is not documented – we can ask you to come up with one on
your own. The asymptotic complexity, of course, will still be O(n) regardless of how good your algorithm is, but we should expect better performance on average since your algorithm should reduce the number of points that you'll need to check for containment. Feel free to talk about it on the newsgroup, though.

This is easy to implement, as it turns out. You can implement the rings as merely an array – you will know ahead of time how many rings you'll need, and we've established that we'll fix the number of subregions to 4, so in theory when you create the structure, you can create a two dimensional array. You will need to figure out an accessing formula to map (x, y) to (i, j) where i is the region and j is the subregion that this point would fall in. Note that you will need to also consider either i+1 or i-1 depending on which way you bias your formula in the event that (x, y) lies on the boundary between two regions; likewise for j if x = 0 and/or y = 0. Your two dimensional array will be an array of Sets, where each Set contains all the geometry in that subregion.

In general, we'll setup your spatial map by giving defining an enclosing region in which all of the points will fall. The center of the spatial map will be determined by the center point of the enclosing region. For example, we'll tell you that your spatial map's "size" will be 1024x1024, meaning that the spatial map will occupy a rectangle from (0,0) to (1024,1024). Your spatial map's largest circle will pass through the points (0,0) and (1024,1024). Your map's center point will be (512,512), which means the subregions will be partitioned along the segments ((512,0),(512,1024)) and ((0,512),(1024,512)). We will tell you how many circles you'll need to create by giving you a resolution value, via an XML attribute.

In the provided code for this part of the project, we will give you a geometry package (geom) and an interface called SpatialMap. For this project, you are not required to implement this interface or use the geometry package at all. However, in the next project we may choose to require that you implement it, in which case modifying your code would be as simple as changing a line like

```
SpatialMap sm = new PolarSpatialMap();
```

with:

```
SpatialMap sm = new PMQuadTree();
```

More work now or more work later, take your pick. ;)

I'll hopefully get you some pictures of what this thing looks like for all you visual learners out there. As always, direct questions to the newsgroup. It's hard to describe a structure with only text; we do our best.

:)

### 4.3 B+ tree

The SortedMap you will (partially) implement for this project is the B+ tree. You will be taught the B+ tree in lecture and there tons of useful B and B+ tree references on the web. Your B+ tree must be correct for **any order greater than 3**.

**Update:** Apparently I was unaware of the definition of order which has caused some mass confusion. I guess this is good news since you all now get a defacto extension of indefinite length which, from the look of it, you could all use for not pointing this out earlier. I must admit that I am baffled that nobody pointed out that the definition I have been using for order is not consistent with anything else being taught in class, both here in the spec and last Thursday. Shame on me for screwing it up, but shame on all of you for not calling me on it. I took this class 4 semesters ago so not all of the definitions are fresh in my head.

For the record, and per Darth Hugue, Shaffer, and just about everyone else the order of the tree is the branching factor. This differs from the definition that I was using which is the number of keys in (primarily) the guides. So, if I infected you with misinformation, the number I was using as order is actually order-1. In other words, a B+ tree with order 3 has at most 2 keys in the guide nodes. (My definition of order that I was operating under would have said that a B+ tree of order 3 has at most 3 keys in the guide nodes, meaning 4 links, which is incorrect). So now I'm telling you that your tree must be correct for all orders greater than 3, meaning you will have at least 3 keys per guide when we test your code, since 2 is messy, and that's what the TreeMap is for.

Fortunately this adjustment will not affect your code much. Since the value passed to the constructor as the tree's order affects the key capacity of your guide nodes, simply subtract 1 from the parameter as soon
as it’s passed in and problem solved – in other words, the order parameter will now be 1 greater than you would have expected it to be operating under my incorrect definition of order. This should be a painless fix. 

Also for the record, to make sure we’re on the same page here (no pun intended!) a general B+ tree does not require that the number of keys in the leaves is the same as the number of guide keys per node, but in our project that is how we’re operating. So if your guides can contain 2 keys per node, your leaves will also contain 2 keys per node. By contain I mean when they are full, i.e., a tree of order 3 has guide nodes that can contain at most 2 keys (at least 1) and leaves that can contain at most 2 keys (at least 1). Got it? 

Good. Sorry for the confusion. Next time if the spec disagrees with something Messh says in lecture, get Darth Hugoe on it, because she’s never wrong.

Oh, and for the record, cardinality is the total number of keys stored in the leaf nodes. This correlates exactly to the value returned by the size() method of Map in Java. I apparently incorrectly assumed that this was evident, either self- or google-.

So, to be perfectly clear:

**Order:** the branching factor of the B+ tree, i.e., how many links per guide node. The total number of keys that can exist in a guide is therefore order - 1. A guide node must contain at least $floor(order/2)$ keys.

A leaf node must contain at least $floor(order-1/2)$ keys.

$v$: the number of keys in a leaf node. For this project, $v$ is order - 1.

Thus, suppose the order is odd (e.g. 5), then $floor(order-1/2)$ is $floor(5-1/2)$ is $floor(4/2)$ is $floor(2)$ is 2, so a guide must contain at least 2 keys, thus 3 links. A leaf node must contain at least $floor(order/2)$ keys.

Note that $floor(order/2)$ is equivalent to ceiling($v/2$), $floor(order/2) = floor(5/2) = floor(2.5) = 2$. Since a leaf can contain at most $v$ keys, and $v = order - 1 = 4$, so ceiling($4/2) = 2 = floor(order/2)$.

Now suppose the order is even (e.g. 6), then $floor(order-1/2)$ = $floor(5/2) = floor(2.5) = 2$. This is correct; for even-ordered trees, when splitting a full guide containing 5 keys and 6 links, by adding a 7th link you are now splitting 7 links across two nodes, and for this you only need 5 keys instead of 6. Thus one half of the split contains 3 keys and 4 links and the other contains 2 keys and 3 links. For leaves, $floor(order/2) = floor(6/2) = floor(3) = 3$. Leaves do not suffer from the imbalance problem that guides do, so they can always be at least 50 since our leaves will contain at most $v$ keys and $v = order - 1$, in this example a leaf can contain 5 keys and thus they must contain at least 3 keys ($floor(order/2)$) to be at least 50

To reiterate:

The B+ tree has two varieties of nodes, guide nodes and leaf nodes. Guide nodes must contain $floor(order-1/2)$ keys at all times and leaf nodes must contain $floor(order/2)$ at all times. The root node of the B+ tree is unique in that it has no fill requirement.

Since not all B+ trees are created equal, we won’t be able to simply diff your output. Instead we will need to programatically examine your B+ tree and determine that it is valid. In the provided code for this part, there is a class in the util package called BPTreeValidator that will examine an XML output of a B+ tree and determine whether it is valid. This class hasn’t yet been thoroughly tested so it is provided as-is (we’re actually hoping you test it for us with your broken B+ trees. We are most interested in incidents where a valid B+ tree is reported as invalid).

To help you debug your B+ tree, I will provide you with a utility which will draw a B+ tree given in the XML format specified in the output DTD. Visualizing your B+ tree will make debugging a lot easier. Note that when testing your B+ tree, adding keys sequentially (say, the strings "1" to "100") will not produce a very good test of your structure since the operations performed are predictable and repeated too often.

Your B+ tree class must be called BPTree and it must exist in BPTree.java. Your class must implement the following constructor:

BPTree(Comparator comp, int order);

### 4.4 XML command specifications

Since we tested you on your adjacency list and your shortest path implementations in part 1, we won’t retest those features this time around. As a result, the shortestPath and adjacencyList elements have been removed from the part 2 DTD.

Edges are not in play for this part of the project since your spatial structure is only efficient for points. As a result, the createEdge and deleteEdge commands have also been removed. They will make a comeback in
the next part of the project when we replace this part’s spatial structure with a more robust one that is efficient for lines too.

A side effect of the aforementioned is that the autoGraph attribute is now deprecated. However, most of your parsers are probably written such that if that attribute were missing, they would probably crash out. As a result, we have not removed the autoGraph attribute from the commands element. Instead, we have changed the legal values list for the autoGraph command so that "true" is no longer legal. So autoGraph will always be "false".

The most important commands in this project will be printBPTree and rangeSearch.

Oh, and this time around, I’ve also remembered to actually give you error messages for invalid XML, so make sure you update existing commands with the new error conditions. ;)

### 4.4.1 Input

In the event that there is a problem with the XML that prohibits normal processing of a command, your processor is required to produce an error. The conditions for faulty input are as follows:

- **An undefined element is present in the input** – In this event, this command element is not defined in the DTD but exists in an input file. An error element should be generated in this event. The value of the command attribute is the name of the undefined element. The value of the message element should be as follows:

  Element `<command>` is undefined

  where `command` is the name of the element (the value of the command attribute).

- **A required attribute is missing from a defined command element** – in this event, you are not required to provide the name of the missing attribute, only indicate that at least one required attribute is missing. An error tag should be produced whose message is as follows:

  Missing required attribute

- **An attribute exists in the element that has not been defined** – in this event, you are also not required to provide the name of the illegal attribute, only indicate that at least one attribute exists in the element that has not been defined. An error tag should be produced whose message is as follows:

  Element contains an undefined attribute

- **An attribute’s value is outside the list of it’s defined enumeration** – for attributes like color in createDot, the DTD provides an enumerated list of legal values. If the value of the attribute is outside this list, the DocumentValidator will throw an exception. The error message reported in this event is:

  Attribute value is not a member of defined enumeration of legal values

- **An attribute’s value is semantically illegal** – this occurs when the value of an attribute does not match the corresponding regular expression for that value’s type. This is not enforceable by a DTD (you need to use the vastly more complicated XML Schema, which serves the same purpose as a DTD, for that), so you need to manually enforce this on a per-attribute basis. There are two relevant error messages that you’ll need to produce, in two cases. The first is when an alpha attribute, such as name, contains characters other than lower and upper case A-Z and the underscore character. The other is when a numeric attribute, such as radius, cannot be parsed into a number. The Integer.parseInt method will throw a NumerFormatException in this event; catching the exception indicates that this error has occurred. The error messages for these two events are as follows:

  Illegal string value for attribute `<attribute>`
  Illegal numeric value for attribute `<attribute>`
where `attribute` is the name of the offending attribute. You will need to produce multiple `error` tags if multiple attributes contain illegal values. You should produce these error tags in alphabetical order according to the name of the attribute, thus for a `createDot` command whose 4 attributes (except color) are all incorrect, you would produce the following set of `errors`:

```xml
<error command="createDot" message="Illegal string value for attribute name" />
<error command="createDot" message="Illegal numeric value for attribute radius" />
<error command="createDot" message="Illegal numeric value for attribute x" />
<error command="createDot" message="Illegal numeric value for attribute y" />
```

**Disclaimer:** In the event of a really nasty XML command that might commit every one of these sins, I am not sure which exception the XMLDocument (validator class will throw first. Thus, the precedence of these errors is whatever precedence the XMLDocumentValidator decides. If this becomes a problem, this decision may be overruled on the newsgroup and in future versions, so stay tuned. The goal is to test each of these errors individually so it won't be an issue.

- **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. This element has several new attributes which will affect the setup of your data structures for this part of the project. In this part of the project, we have not removed the `autoGraph` attribute – instead, we have given it only one legal value: `false`. The new attributes are `bpOrder`, whose value is an integer. This number represents the order of the B+ tree you will use for this input. The number must be positive and greater than or equal to 3. If the attribute's value is less than 3, you should produce one `error` tag, whose command attribute’s value is "commands" and whose message is:

```
Illegal order (⟨order⟩) for B+ tree
```

where `order` is the invalid attribute.

The next attributes are `spatialWidth` and `spatialHeight`, both unrestricted integers (we will not provide integers that are too large or small for 32 bit 2’s complement to represent). These two attributes define the rectangular region the spatial data structure can store. Note that the lower left corner of this region is always (0,0), the origin, as we will not be dealing with negative coordinates in this project. Thus, the spatial structure’s bounding volume is the rectangle whose lower left corner is (0,0) and whose width and height are given by these two parameters. You must setup your spatial structure to be centered accordingly. Note that although all of our coordinates will be given as integers, you may need to center your spatial structure around a coordinate that is not an integer, so make sure you plan accordingly.

Observe that these two values will affect the success of the `createDot` command, as it is now the case that dots must fall strictly inside this range (i.e., exclusively – this means dots that fall on the border of the rectangle defined by (0,0) and the values of `width` and `height` are considered illegal).

The last attribute for this part of the project is `resolution`, whose value is an integer. This value will ultimately determine how many rings your spatial structure will contain, as the resolution value determines the width of each ring.

We will probably provide a printSpatialMap command in the near future, in which case the resolution value provided here will influence how your spatial map should be serialized to XML.

- **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 \( \text{name}_e \), \( \text{x}_e \), and \( \text{y}_e \) are the values of the name, x, and y attributes of the \( \text{createDot}_e \) 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 \(<\text{name}> \) already exists**
**Dot at \((<\text{x}>,<\text{y}>)\) already exists**

In this part of the project, dots must also fall strictly (exclusively) inside the boundary defined by the attributes \textit{spatialWidth} and \textit{spatialHeight} given in the \( \text{command}_i \) tag. In the event that at attempt is made to create a dot outside of this range, print the following error:

**Dot \((<\text{x}>,<\text{y}>)\) is out of range**

- **clearAll** – resets all of the structures, clearing them. This has the effect of removing every dot. This command cannot fail, so it should unilaterally produce a \( \text{\success}_e \) 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 \( \text{\success}_e \) tag will be produced in the output XML. This \( \text{\success}_e \) tag will have no \textit{message} attribute. The resulting \( \text{\success}_e \) tag will have one child element, \( \text{\dotList}_e \). A \( \text{\dotList}_e \) has one or more child elements of type \( \text{\dot}_e \). A sample \( \text{\success}_e \) for this command:

  ```xml
  <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 \( \text{\dot}_e \) 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 \textit{sortBy} attribute in the \( \text{listDots} \) command, whose two legal values are \textit{name} and \textit{coordinate}. The ordering by name is ascilbetical 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 \( \text{\error}_e \) tag should be produced with the following message:

  **No dots exist to list**

- **printBPTree** – prints the structure of your B+ tree. Since your B+ tree will be used in place of TreeMaps for your data dictionary (that maps strings to Dots), the B+ tree printed by this command should print the B+ tree containing all of the dots created so far. Your B+ tree is a map of Strings to Dots, so your keys should be strings and your values should be dot objects, which we will print in the XML as coordinate pairs (x,y). This command cannot fail, so it produces a non-empty \( \text{\success}_e \) tag. This \( \text{\success}_e \) tag will have one child element, \( \text{\bdptree}_e \). The \( \text{\bdptree}_e \) element will print out structure of your B+ tree. Since the structure of a B+ tree is nondeterministic, we will not be able to diff the output of your XML files for this part of the project. Instead, we will use a tester class which examines the structure of your \( \text{\bdptree}_e \) element and determines whether it firstly contains all the data it should contain and secondly that it is a legal B+ tree. The structure of the \( \text{\bdptree}_e \) is listed in the DTD, but more specifically, the XML structure should be as follows:
- **bptree** - the root of the B+ tree xml. This tag has three required attributes: *cardinality*, whose value should be the size (number of keys) held by the B+ tree; *height*, the number of levels in the B+ tree; and *order*, the number of keys per node in this B+ tree.

- **guide** - represents a guide node in the B+ tree. This element has no attributes. Its children should alternate between (either *leaf* or *guide*) and *key* nodes, which logically expresses the configuration of a *guide* node. Clearly, a *guide* node’s children are going to be *key* nodes and either all *guide* nodes or all *leaf* nodes.

- **leaf** - represents a leaf node in the B+ tree. A leaf has no attributes, and contains *entry* child nodes corresponding to the key/value pairs mapped by this B+ tree.

- **key** - represents an an index key in a guide node. A *key* node must appear between either two *guide* nodes or two *leaf* nodes. The *key* node has one attribute, *value*, which represents the actual key.

- **entry** - represents a key/value pair as exist in leaf nodes. The *entry* element has two attributes, *key*, the key, and *value*, the value. Because our B+ tree will map dot names to dot objects, the key attribute’s value should be the dot name and the value attribute’s value should be the coordinates of that dot, printed as "(x,y)" where x and y are the x- and y- coordinates of that dot, respectively.

An example of a B+ tree XML output will be provided.

- **rangeSearch** - requests a radial range search of your spatial map. This command will succeed if at least one dot exists in that range, in which case it will produce a non-empty *success* tag. The rangeSearch tag has three attributes, *x* which specifies the x-coordinate of the search circle’s center, *y* which specifies the y-coordinate of the center, and *radius* which indicates the radius of the search circle. The *success* tag for this method will contain a single child element, *dotList* whose format is the same as the *listDots* command. This *dotList* should contain only the dots which fall inclusively inside the search circle (inclusively means that dots falling exactly on the edge of the circle will be included in this list).

As always, dots should be printed in ascilietical order according to their name.

If the range search does not return any dots, an *error* tag should be produced whose message is:

No dots exist in the given range

### 4.4.2 Input DTD

```
<!ELEMENT commands (createDot|deleteDot|clearAll|listDots|printBPTree|rangeSearch)>
<!ATTLIST commands
  autoGraph (false) #REQUIRED
  bpOrder CDATA #REQUIRED
  spatialWidth CDATA #REQUIRED
  spatialHeight CDATA #REQUIRED
  resolution CDATA #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 clearAll EMPTY>

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

<!ELEMENT printBPTree EMPTY>

<!ELEMENT rangeSearch EMPTY>
<!ATTLIST rangeSearch
x CDATA #REQUIRED
y CDATA #REQUIRED
radius CDATA #REQUIRED
>

4.4.3 Output DTD

<!ELEMENT results (success|error)>*

<!ELEMENT success (bptree|dotList)>*
<!ATTLIST success
command CDATA #REQUIRED
message CDATA #IMPLIED
>

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

<!ELEMENT dotList (dot)>*
<!ELEMENT bptree (guide|leaf)>*
<!ATTLIST bptree
cardinality CDATA #REQUIRED
order CDATA #REQUIRED
height CDATA #REQUIRED
>
<!ELEMENT guide (key|guide|leaf)>*
<!ELEMENT leaf (entry)>*
<!ELEMENT key EMPTY>
<!ATTLIST key value CDATA #REQUIRED>
<!ELEMENT entry EMPTY>
<!ATTLIST entry
key CDATA #REQUIRED
value CDATA #REQUIRED
>

<!ELEMENT dot EMPTY>
<!ATTLIST dot
name CDATA #REQUIRED
x CDATA #REQUIRED
y CDATA #REQUIRED
radius CDATA #REQUIRED

34
5  Part 3: SortedMap Strikes Back, More B+ tree fun, PM Quadtreess, and the Fibonacci Heap

5.1 SortedMap, part 2

For this project, in addition to fixing any problems with your SortedMap implementation from Part 2, you will be required to finish implementing the remaining SortedMap methods. The new methods you will have to implement for Part 3 are as follows:

- `Object remove(Object key);`
- `Set keySet();`
- `Collection values();`
- `Object firstKey();`
- `Object lastKey();`
- `SortedMap headMap(Object toKey);`
- `SortedMap subMap(Object fromKey, Object toKey);`
- `SortedMap tailMap(Object fromKey);`

The major function, of course, is remove(), whose implementation will carry the most weight toward your grade. keySet() and values() behave exactly like entrySet() with the exception of what they "contain"—the entrySet() contained Map.Entry objects containing both the key and the value, while keySet() will "contain" only the keys and values() will contain only the objects mapped to by the keys. Note that values() is a Collection and not a set since this can contain duplicates whereas keys must be unique. As with entrySet(), you are expected to implement all of the appropriate methods in the set returned by keySet() and all the methods in the Collection class for the object returned by values().

UnsupportedOperationExceptions should only be thrown by methods explicitly unsupported according to the API (e.g. add() and addAll()).

firstKey() and lastKey() are trivial; for the former, just descend your tree following the left most child, for the latter, descend using the rightmost child, and pick the appropriate keys out of those nodes.

For the headMap(), subMap(), and tailMap() you’re going to have to implement SortedMap a second time. Like the entrySet() method which returns an inner class of your BPTree whose methods simply call the methods in the BPTree class, you will be doing something similar with subMap(), except this time you will check the arguments before blindly forwarding the method call to BPTree. For instance, when invoking put() on the submap returned by the method, if the specified key does not fall between fromKey and toKey according to the rules of the API, the method throws an IllegalArgumentException instead of forwarding the call to put in the BPTree class. Although there are three methods here (headMap, subMap, tailMap), you only need to write a SubMap inner class in your BPTree. This SubMap class will inevitably contain the start and end keys (i.e., the class needs to store the fromKey and toKey parameters passed into it by the method call in order to check them when other methods are called). Your SubMap class can store either of these values as null to indicate that there is no limit (i.e., if the start key you store in the SubMap class is null, you can write the code to assume that means you don’t have to check that incoming keys are greater than some start key). Thus, for headMap and tailMap you would simply return a new SubMap whose start or end key is null, respectively, and code your SubMap methods accordingly.

If this is confusing the hell out of you, take a look at the SkipList implementation I provided. It shows exactly what I’m talking about.

Note that not all of the methods in this SubMap you will have to write are as simple as a call to the enclosing BPTree object. For instance, the size() method will have to operate in theta(m) where m is the size of the SubMap. There’s no way to store a size variable inside the SubMap and increment it when its put() method is called like you are probably doing with the BPTree class because just like the entrySet, changes in the parent map reflect the subMap. If I create a SubMap s from BPTree b, the size of s could
be changed either by a call to put() on s or on b. So whenever size() is invoked on s, it will have to be
calculated.
Also note that since this is a SortedMap, it has entrySet(), keySet(), and values() methods. Implementing
these for your SubMap is optional (worth extra credit if you do). If you choose not to, they should throw
UnsupportedOperationException. Note that a SubMap can be generated from another SubMap only if
the range it represents is within the range of the SubMap. To create a SubMap from within a SubMap, you
should check the arguments and if they are valid, pass the call directly back to your BPITree class.
(Otherwise, you might think you need a SubSubMap, etc...)
Again, I suggest looking at the SkipList code, since an implementation is worth a thousand words. And
honestly, a lot of that code is not implementation specific. In other words, you could probably get away
with copying and pasting a lot of the SubMap code out of my SkipList since very little of it falls outside
the range of the public interface. Save yourself some time; this is a long project.
As always, refer to the API. Make sure your methods do what the API says they're supposed to, because
we will be checking that they do. :) 

5.2 More B+ tree fun

In addendum to the B+ tree specifications from the part 2 specifications:
In this part of the project you will be required to implement key deletion functionality in your B+ tree.
Under no circumstances should you do what I have seen some students do in the past: delete by creating a
new tree and reinterseting every other key except the one they are supposed to be deleting. Your deletion
algorithm must result in a correct tree and achieve a complexity of O(log n). This means you will be
required to pass a speed test for this portion of your project. Also, although the spec didn't explicitly
mention complexities required for the B+ tree, let me say this: key search, add, and remove must be O(log
n). This is as per the definition of a B+ tree. It's no more accurate to call a BST in array form a BST if all
your algorithms are linear in the average case than it is to call a B+ tree in any form a B+ tree if all of its
operations are linear. A B+ tree is defined by its algorithms as much as it's defined by its structure.
Also, I forgot to mention this in the last spec, so let me assure the masses. Given a B+ guide node with
the following keys:

\[
\begin{array}{ccc}
X & Y & Z \\
\end{array}
\]

My understanding of what pretty much everybody does with the B+ tree is that the first link from that
guide would contain only keys strictly less than X. The second link would contain keys greater than or
equal to X and strictly less than Y. However, this is an arbitrary decision. It's also valid to say that the
first link contains keys less than or equal to X, and the second link contains keys strictly greater than X
and less than or equal to Y. This caused a bit of last-minute confusion, but don't worry. We caught it in
time and your projects were graded with two different validators, one which obeys the first rule and the
second which obeys the second rule. The result is that if you passed either one of those tests you got
credit. I just thought I would point it out. The first scenario is preferred since it is more commonly used,
so keep that in mind if you ever find yourself coding a B+ in the future. It didn't even cross my mind that
anyone would do it a different way. Learn something new every day, eh?

Now for the bad news...

After the snafu in the last project in which people mistakenly took cardinality to mean the number of
keys in the leaf nodes (different from the number in the guides as determined by the tree’s order), and to
punish those students who did not take my advice in separating the guide nodes and leaf nodes into
separate structures, this time around your B+ tree will contain an arbitrary number of keys, separate from
the order of the tree. This number will be issued in the commands.xml XML tag via an attribute called
leafSize. So, for example, we might have you create a B+ tree whose order is 4 (meaning a maximum of 3
guide keys per node) whose leaves have a capacity for 100 keys.
If you designed your B+ tree correctly, this will result in relatively minor changes to your code, since your
guide nodes and leaf nodes are separate objects with separate data members and separate methods. If you
didn't, you're screwed now. Have fun totally rewriting large portions of your code or giving up on your own
and borrowing somebody else’s. If you’re in the screwed boat, hopefully you learned your lesson. O
design is extremely important – modularity makes for easy code extensions when the spec changes, which it has. Hope you enjoyed part 2 because you really had it easy, and now I’m getting revenge. ;)

As a result, you must also provide a new constructor, whose signature is as follows:

```cpp
BPTree(Comparator comp, int order, int leafSize);
```

### 5.3 PM Quadtrees

As forecasted, this project introduces a vastly more powerful spatial data structure, the beloved PM Quadtree. PM Quadtrees have orders, which you will learn about in class and will read about it in Samet.

For this project you have a few implementation options worth varying credit:

- **"I want to pass"**: code a PM1 Quadtree. This is the "full credit" option. Note that we aren’t implying that you are going to fail the class if you don’t, but you won’t be eligible for full credit if you code anything less.

- **"Evan screwed you over with the B+"**: code a PM3 quadtree. This will net you only 50% of the possible points for the spatial map, but it’s easier to code.

- **"Fail with pride"**: code a PR Quadtree. This will net you only 25% of the possible points and honestly is only slightly easier than the PM3. A PR Quadtree does not contain line data, only points, so it has the same functionality of the spatial map in part 2 but nothing more.

- **"F--"**: code nothing. This will net you only 0% of the points but considering the size of this project skipping the spatial map (and passing the other parts) will still qualify as a good faith effort.

- **"Overachievers Anonymous"**: provide both a PM1 and a PM3 quadtree. This will net you up to 125% of the possible points on the spatial map portion of the grade, so it can make up for deficiencies in other areas. Given that I will show you how to design a PM Quadtree that can assume any order by changing fewer than 10 lines of code, overachievement is an overstatement.

- **"The Greg Crosswhite Special"**: provide a PM1, PM2, and PM3 quadtree. We only really expect that Greg will do this, and he probably would have done it anyway. This grading option is just for him. But if you think you are hot stuff, you can do it too. This option will net you up to 150% of the points for the spatial map, and is super cool.

We will work out submission instructions for you to indicate which option you implemented so we can pass you the appropriate test files.

Because there is literature written about PM quadtrees and you will hear at least one lecture on the subject, I will not go into detail here about their structure and how they are built.

I will also give you a lecture about how to implement PM Quadtrees. If you pay attention, I will show you how to use good OO design to make implementing the Crosswhite Special trivial. Yes, I mean trivial – no harder than a BST.

For this project, and depending on which option you implement, we have a few new fun operations we can perform on spatial maps. Since the PM quadtrees also store lines, we’re back to creating edges. And once we have edges in this map, we can party.

The radius search is still in, plus this time around the radius search will also report any edges that exist in that radius. We also introduce two new operations:

**Nearest Point to Segment**: Given a segment s, locate the nearest point. Java gives you the geometric calculations so you won’t have to muck around with the math (damn, you get off easy!) but you will still need to come up with a suitable algorithm to do this efficiently.

**Nearest Segment to Point**: Given a point p, locate the nearest segment. If you’re clever you can roll this operation and the previous one into a single operation where the type of geometry is irrelevant (my geometry classes attempt to make the type transparent by providing common functions like intersects() and distance()) – if you use these or roll your own with similar characteristics, you can write one "nearest object to object" where it’s irrelevant whether those objects are points or lines. It’s all a matter of distance, after all.
Naturally if you choose to implement only a PR quadtree then you will not be able or required to implement these new operations since you have no lines to work with. One of the convenient features of PM quadtrees is that they are deterministic, meaning all quadtrees ARE created equal. The order by which objects is irrelevant to the resulting structure, and that structure is invariable. All your quadtrees should look exactly the same, which means we can and will be using diff to test your quadtrees.

Thus, you’re not required to implement any specific interface, Java provided or otherwise. You can implement your quadtrees however you want.

For this project, to make your quadtrees convenient, the spatialWidth and spatialHeight attributes of the commands tag will always be powers of 2 (not necessarily the same). e.g. 128, 256, 512, 1024, so on. This makes dividing the regions a very quick operation. Java has bitshift operations that you are welcome to take advantage of (although javac probably converts x/2 into x/2). Recall shifting the bits of an integer one to the right has the effect of dividing it by 2 (and rounding down) – similarly x*2 multiplies x by 2. x*42 multiplies x by 4, and so forth. Using the % operator on ints for division gives you that smart and doesn’t leave compiler optimization to chance. And obviously it doesn’t work on floats. If you didn’t know that, please sign up for CMSC311 immediately.

Also, for this project we will not create partitions in your PM Quadtree deep enough to create regions whose area is less than 1x1. Thus we are limiting the height of the tree based on max(spatialWidth,spatialHeight). (Note that I said “based on” not “exactly equal to”). Figure out what the height will be as a result yourself, that’s part of coding the B+ tree and also a fun exam question.

Hoohah!

Note: In Samet’s text he suggests that PM borders are closed on only two sides (bottom and left, like the PR quadtree). We disagree with this for reasons known only to Meesh (she’s explained it a bunch of times but I still don’t understand it; hopefully you’ll have better luck). For our PM Quadtrees, if a point or line lands on any of a region’s borders, that point or line is added to that region. For instance, suppose you had a simple instance where your quadtree occupied an area defined by (0,0), (1024,1024). The center point of the first partition that will be created, therefore, would be at (512,512). If the first point you add is at (512,512), this point would be added to the data dictionaries of all four of the black nodes (the children of the resulting grey node), since the point touches the corners of each of those four regions. Got it? Good. The same rule applies for the spatial map from the last part in terms of points falling on the boundaries of region defined by spatialWidth and spatialHeight. The resolution attribute no longer has a meaning. We will not remove it from the DTD to prevent you from having to comment your code that handles it, but be aware that PM Quadtrees do not have “resolutions” and thus the attribute will not be used.

Our test files will include a new attribute in the commands tag, pmOrder, whose value can be one of the following:

0 - PR Quadtree
1 - PM1 Quadtree
2 - PM2 Quadtree
3 - PM3 Quadtree

We will run your code through tests of various orders, and give you credit for all of the ones you passed, scaling to reflect the total amount of credit we’re offering for each of the options. If you are not supporting certain orders in your project, you can print anything you want (or nothing at all) when you encounter a pmOrder attribute whose value indicates something you didn’t do. In the end you’ll just fail the test anyway regardless of what you print since we’ll be checking via diff. It would be nice if you at least printed a results tag so the XMLParser doesn’t throw an exception when we reroute your output back into the parser to autoformat it for diffing, but we’ll cover our bases. It will speed up our testing a little bit since exceptions are slow to throw, but whatever.

We will not speed test your PM Quadtree specifically because its complexity is not good as far as add and remove are concerned. We will speed test your nearest to nearest algorithms in the same way we speed tested your spatial map from the previous part, so you’ll have to spend some time coming up with a reasonable average case algorithm. Note that no matter what you do the complexity will still be O(n), but on average this should take drastically less. ;)

38
We will grade your PM primarily by using `printPMQuadtree /i`, since unlike the B+ tree, all PM Quadtrees given the same geometry should be identical. Another nice feature of the PM is that they are also identical regardless of the order of operations. In other words their structure is deterministic for inputs. Fun, huh?

Since I was feeling exorbitantly generous while writing this, I've decided to write up the pseudo-code for a Java PM quadtree. Note that in doing this, I've eliminated the possibility of there being ANY excuse for not providing me a project that passes all of the tests of orders 0-3 and result acquiring for yourself a sizeable hunk of extra credit.

### 5.3.1 PM Pseudocode

If you're having trouble understanding this, make sure not to miss my PM lecture when I give it in the next few weeks. I'll explain it and you can ask questions, etc.

class PMQuadtree {

class Node; // define this yourself; superclass for Grey and Black; I used an interface, but this is pseudo

White SingletonWhiteNode = new White();

class Grey extends Node;

Node[] children;
Rectangle[] regions; // a grey node is a partition into 4 regions; these rects are those regions

// parameters here are stuff like the geometry you're adding and any other info you need on the stack
Node add(Geometry g, ...) {
    if (g intersects regions[0]) children[0] = children[0].add(...)
    ...
    if (g intersects regions[3]) children[3] = children[3].add(...)
}

Node remove(Geometry g, ...) {
    if (g intersects regions[0]) children[0] = children[0].remove(...);
    ...
    if (g intersects regions[3]) children[3] = children[3].remove(...);

    if (all children are white) return SingletonWhiteNode;
    if (three children white, one child black) return the black child;
    if (no children are grey and at least 2 are black) {
        Black b = new Black;
        add all geometry in each black node into b
        if (b is valid) return b;
        else return this;
    }
    // this grey is still necessary, so keep it in the tree
    else return this
}
}

class Black extends Node {
List geometry;

Node add(Geometry g, ...) {  
    add g to geometry  
    if (this node is valid) return this;  
    else return partition(...);  
}

Node remove(Geometry g, ...) {  
    remove g from geometry  
    if (geometry is empty) return SingletonWhiteNode;  
    else return this;  
}

Node partition(...) {  
    Grey g = new Grey(...);  
    for each item i in geometry {  
        g.add(i);  
    }  
    return g;  
}

}

class White extends Node {

Node add(Geometry g, ...) {  
    return new Black(g);  
}

Node remove(...) { error }

}

Now, naturally it's up to you to fill in the elipsis (the parameter lists) with whatever you need. You will need info about the depth of tree, primarily, so you can figure out how to make the rectangles for the grey. Note that you don't necessarily have to store those regions. I do it as an efficiency issue (space vs. time). You could compute the rectangles on the fly each time, but I think that's wasteful, so I chose to store them. Also note: the key to making your PM Quadtree a PM1, PM2, PM3, etc is the line in Black.add() where I say:

if (this node is valid) ...

This is the only part of the code that differs between the PM1, PM2, and PM3 (well, the "validity check" is done in the Grey.remove() also, but you get the idea). A PM3, for instance, is valid as long as it only has zero or one points in it. It is allowed to have any number of q-edges as long as they don't intersect each other. If you attempt to insert a second dot into this list, it becomes invalid, and then partition is called.  

OO design here is key. There are some details I omitted from the pseudocode. For instance, the way I setup my code to allow for easily adapting the generic PM Quadtree into a PM1, PM2, and PM3 was not straight subclassing as you might expect. The reason for this becomes apparent when you try.  

Suppose you subclass PMQuadtree into PM1Quadtree and PM3Quadtree. The problem here is that the PMQuadtree class is not the one that changes, its inner classes do. What you really need to subclass is the black and grey nodes, since they are the ones that contain the valid() code. The easiest way to do that to prevent you from having to rewrite the entire add() and remove() methods of Black and Grey respectively to change only a few of its lines would be to isolate the validation operation into a method, say valid(),
that you can override and let dynamic dispatch handle it. But there’s a problem with subclassing the inner classes.

The Grey and Black classes refer to each other. Also, presuming you write the methods for PMQuadtree itself (such as add, remove, etc – I showed only the add and remove for the node classes), you will refer to Grey and Black by name in the base class. This former is really the bigger problem because, suppose you subclass Grey and Black and only change their valid() method, for instance in a PM1Grey and PM1Black class. If you only changed the PM1Black.valid() method, then when it calls partition() it will return a new Grey, not a new PM1Grey. You’d have to override that method to change the type it instantiates when it partitions, and now you’ll quickly realize that you’re rewriting the code. Boo!

What you really want is to isolate the validation operation and be able to subclass that without changing anything else.

You’ve heard of comparators: external objects that can be passed into sorted structures to change their behavior.

Why not create a validator object? Make your PMQuadtree contain a Validator object as a member, and make your PMQuadtree constructors require that you pass in one of these Validator objects. Make it an interface. Now create three implementations: PM1Validator, PM2Validator, PM3Validator. The code for the validator’s valid() method is only about 10 lines long. You’ve just magically done what I’ve done (because I told you how) and now you have a PMQuadtree that becomes one of three orders with only one method changing (and very little code). The best thing to do is create three subclasses, PM1Quad, PM2Quad, PM3Quad who contain private static inner classes which implement the validator and hardcode the PM1’s default constructor (the one that takes zero parameters) to pass its private validator to the superclass constructor (via a call to super(…, ) as the first line of the ctor), and make PMQuadtree an abstract class. Note that you can still provide constructors for abstract classes (for the purpose of their subclasses calling them).

This is technically an instance of the Decorator design pattern. Read more about it if you’d like. Design patterns actually do rock. Take CMSC433, ;)

Based on the fact that I just essentially told you how to get the extra credit on this project really easily, I should make it mandatory. But I’m aware that despite my brilliance some of you insist on ignoring my advice, and I am not adverse to punishing you for that. Suit yourself. ;) As long as you don’t go double or nothing on grabbing a tennis ball from under a running lawn mower and lose all of your fingers on both hands, there’s no excuse for not passing all of the PM order tests.

PS: The PR is a bit harder to implement, honestly. It’s not as easily adaptable because a PR is defined to be closed on only two of its borders instead of all four. If I were to implement it, I would actually subclass my point geometry and modify its intersects() method so that when comparing against a rectangle it only reports true if it lands on the 2 closed borders instead of all four, and pass those point objects into a PMQuadtree with a PRValidator. Note that if you take this approach it definitely makes a difference whether you say (regions[0].intersects(g)) or (g.intersects(regions[0])) in your Grey node. The former will not work. The latter will. The reason is that your PMQuadtree base class creates a Rectangle[] and uses that for intersection checking. To be honest it’s not that hard – just make the instantiation of that array (the part where you say regions = new Rectangle[4]) in a separate, overrideable method that you could change for the PRQuadtree so that regions = new PRRectangle[4]) would be fine. And honestly, that’s the preferred implementation because if you want to do it right, your trees will allow anything that implements some Geometry interface. This is not a requirement by any means, but it’s a nice touch if your tree can accept any point object that implements the Geometry interface instead of specific point objects, and that’s better accomplished by subclassing the rectangle than the point.

Also: if you want to be real fancy, make your trees operate in any precision (byte, short, int, long, float, double). Mine does. ;P You won’t get any extra credit for it but any of you who think you’re leet coders might consider giving it a shot.

### 5.4 Fibonacci Heap

As if we haven’t assigned enough already, in addition we would like you to implement a fibonacci heap (f-heap). The f-heap is a spectacular min-heap that performs in amortized constant time. This makes
Dijkstra’s fun.

What’s that I hear? Yes... that’s right, a ghost of projects past... Dijkstra’s is back, suckers! We’ll be testing you on it again with your new f-heap for even better performance, because we love double jeopardy and making your lives difficult. With any luck you’ll be thankful for a faster shortest path implementation this year. Who needs turkey when you have Java?

This time around, in addition to bringing back Dijkstra’s tests, we are also going to provide commands that will independently test your f-heap to make sure you actually coded the beast. This will involve introducing new commands, `enqueue`, `dequeue`, ad `printQueue`. We’ll queue up some strings, remove one, and then see if your f-heap looks like it should. Note: not sure at this point whether we’ll be doing validating like with the B+ or straight diffing. I’m not sure if f-heaps are deterministic or not. More on that later when I come up with the answer to that question.

How to code the f-heap? Well I’m no expect on the subject myself having never coded one, but I’ll actually implement it too, just for you, so that I can hopefully sound intelligent when attempting to answer questions to that effect on the newsgroup and in office hours provided you remember to e-mail me and ask me to show up. I usually don’t recommend CLR for pseudocode but it’s got reasonable coverage on f-heaps and will hopefully serve you in stead.

5.5 Grading

We understand that this is a significantly large project – this is a result of rolling what would have been project 4 into a bigger project 3. We did this so we could give you more time for part 2 and part 3, but also as a way to mix-n-match in such a way that will optimize your credit opportunities without spending the entire month of November writing code. As a result, this project is worth a total of more than 100

- B+ remove: 25%
- SortedMap implementation: 25% + 5% extra credit for entrySet/keySet/values() in subMap()
- SpatialMap: 35% (full credit option)
- Spatial Map applications: 15% (5% range search, 5% for nearest pt to segment, 5% for nearest segment to pt)
- Fibonacci Heap: 35%

The total without any extra credit is 25% + 25% + 35% + 15% + 35% = 135%.

There is also a late penalty available for this project – you will be allowed to submit part 3 up until the last day of classes. Submissions on or before Saturday, December 4 are eligible for up to 75% submissions on or before Friday, December 10 are eligible for up to 50% submissions after December 10 are not guaranteed to be accepted and are done so at Dr. Hugue’s discretion on a per-case basis.

So, although this looks like a huge project, you can earn points in a variety of ways, so all hope is not lost. Any two of the three structures will earn you a passing grade, and partially working versions of all 3 will likely yield the same.

5.6 XML Command Specification

Yeah, yeah. TBI.

5.6.1 Input

In the event that there is a problem with the XML that prohibits normal processing of a command, your processor is required to produce an error. The conditions for faulty input are as follows:

- An undefined element is present in the input – In this event, this command element is not defined in the DTD but exists in an input file. An error element should be generated in this event. The value of the command attribute is the name of the undefined element. The value of the message element should be as follows:
Element <command> is undefined

where <command> is the name of the element (the value of the command attribute).

- **A required attribute is missing from a defined command element** – in this event, you are not required to provide the name of the missing attribute, only indicate that at least one required attribute is missing. An error tag should be produced whose message is as follows:

**Missing required attribute**

- **An attribute exists in the element that has not been defined** – in this event, you are also not required to provide the name of the illegal attribute, only indicate that at least one attribute exists in the element that has not been defined. An error tag should be produced whose message is as follows:

**Element contains an undefined attribute**

- **An attribute’s value is outside the list of it’s defined enumeration** – for attributes like color in createDot, the DTD provides an enumerated list of legal values. If the value of the attribute is outside this list, the DocumentValidator will throw an exception. The error message reported in this event is:

**Attribute value is not a member of defined enumeration of legal values**

- **An attribute’s value is semantically illegal** – this occurs when the value of an attribute does not match the corresponding regular expression for that value’s type. This is not enforceable by a DTD (you need to use the vastly more complicated XML Schema, which serves the same purpose as a DTD, for that), so you need to manually enforce this on a per-attribute basis. There are two relevant error messages that you’ll need to produce, in two cases. The first is when an alpha attribute, such as name, contains characters other than lower and upper case A-Z and the underscore character. The other is when a numeric attribute, such as radius, cannot be parsed into a number. The Integer.parseInt method will throw a NumberFormatException in this event; catching the exception indicates that this error has occurred. The error messages for these two events are as follows:

*Illegal string value for attribute <attribute>*
*Illegal numeric value for attribute <attribute>*

where <attribute> is the name of the offending attribute. You will need to produce multiple <error> tags if multiple attributes contain illegal values. You should produce these error tags in alphabetical order according to the name of the attribute, thus for a createDot command whose 4 attributes (except color) are all incorrect, you would produce the following set of <errors>:

```xml
<error command="createDot" message="Illega string value for attribute name" />
<error command="createDot" message="Illega numeric value for attribute radius" />
<error command="createDot" message="Illega numeric value for attribute x" />
<error command="createDot" message="Illega numeric value for attribute y" />
```

**Disclaimer:** In the event of a really nasty XML command that might commit every one of these sins, I am not sure which exception the XMLDocumentValidator class will throw first. Thus, the precedence of these errors is whatever precedence the XMLDocumentValidator decides. If this becomes a problem, this decision may be overruled on the newsgroup and in future versions, so stay tuned. The goal is to test each of these errors individually so it won’t be an issue.

**Clarification on ”round to 3 decimal places”:** print out the numbers according to their significant figures. In the first version of the spec in the example output I used ”10.00” as an example when I should have instead used ”10.345”. If the number is ”10.0004” this should be printed as ”10”, whereas ”10.0005” should be printed as ”10.001”. In other words, don’t print 3 zeroes if they are not significant. Print as few digits as are necessary to accurately deliver the result.
• **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. This element has several new attributes which will affect the setup of your data structures for this part of the project. *autoGraph* is back in play; its legal values are now **true** and **false**, whose meanings are identical to part 1's. The new attributes are **bpOrder**, whose value is an integer. This number represents the order of the B+ tree you will use for this input. The number must be positive and greater than or equal to 3. If the attribute's value is less than 3, you should produce one `error` tag, whose command attribute's value is "commands" and whose message is:

**Illegal order (**<order>**) for B+ tree**

where `order` is the invalid attribute.

A new attribute this time around is **leafSize** whose value determines how many keys are stored in the leaf nodes in the B+ tree. This value must be greater than or equal to **bpOrder**. If it's not, the following error message is generated:

**Illegal leaf size (**<leafSize>**) for B+ tree**

Another new attribute is **pOrder** whose values are enumerated and defined as follows:

0 - PR Quadtree  
1 - PM1 Quadtree  
2 - PM2 Quadtree  
3 - PM3 Quadtree

In the event that this value is not in that range, the error is that the value is outside of its defined legal enumerated values. **If your project does not support the value given**, in other words you did not implement a PR/PM of the given order, you should produce a single `error` tag for the `commands` tag, whose format looks like:

```
<error command="commands" message="Spatial map of order <order> is not supported by this implementation.
```

The next attributes are **spatialWidth** and **spatialHeight**, both unrestricted integers (we will not provide integers that are too large or small for 32 bit 2's complement to represent). These two attributes define the rectangular region the spatial data structure can store. Note that the lower left corner of this region is always (0,0), the origin, as we will not be dealing with negative coordinates in this project. Thus, the spatial structure's bounding volume is the rectangle whose lower left corner is (0,0) and whose width and height are given by these two parameters. You must setup your spatial structure to be centered accordingly. Note that although all of our coordinates will be given as integers, you may need to center your spatial structure around a coordinate that is not an integer, so make sure you plan accordingly.

Observe that these two values will affect the success of the `createDot` command, as it is now the case that dots must fall strictly inside this range (i.e., exclusively - this means dots that fall on the border of the rectangle defined by (0,0) and the values of width and height are considered illegal).

**resolution** is now deprecated, and note that it is no longer a member of the DTD. Thus, it should now be considered an illegal attribute.

• **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 \( \text{name}_i \), \( \text{x}_i \), and \( \text{y}_i \) are the values of the name, \( x \), and \( y \) attributes of the \textit{createDot}\( _i \) 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 \textit{<name>} already exists
Dot \textit{(\langle x\rangle,\langle y\rangle)} already exists

In this part of the project, dots must also fall strictly (exclusively) inside the boundary defined by the attributes \textit{spatialWidth} and \textit{spatialHeight} given in the \textit{commands}\( _i \) tag. In the event that at attempt is made to create a dot outside of this range, print the following error:

Dot \textit{(\langle x\rangle,\langle y\rangle)} is out of range

Also, in this project, rather than create a new command to explicitly map pre-existing dots into the quadtree, just as in projects past you will automatically insert the newly created dot into whichever spatial structure has been specified via the \textit{pManualOrder} attribute in the \textit{commands}\( _i \) element. As a result, it is possible that the dot will not be legal because it would require partitioning the PM quadtree deeper than we allow. In this event, the error message should be:

Dot \textit{(\langle x\rangle,\langle y\rangle)} requires partitioning the quadtree deeper than allowed

- \textbf{deleteDot} - removes a vertex with the specified name. The criteria for success here is simply that the dot exists. The success message is:

Dot \textit{\langle name\rangle} 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 \textit{deleteEdge}\( _i \) command had been issued for each of these edges. Thus, this command may produce many other \textit{success}\( _i \) tags.

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

Dot \textit{\langle name\rangle} does not exist

- \textbf{createEdge} - creates an edge between the two dots (vertices) named by the \textit{start} and \textit{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:

\textbf{Edge \langle\textit{start\rangle},\langle\textit{end\rangle} \rangle successfully created

\textbf{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 \( \langle\textit{start\rangle} \rangle \) does not exist
End point \( \langle\textit{end\rangle} \rangle \) does not exist
Edge \( \langle\textit{start\rangle},\langle\textit{end\rangle} \rangle \rangle \) already exists

45
where $\%\text{start}$ and $\%\text{end}$ are the values of the start and end attributes, respectively.

If $\text{pmOrder}$ has been set to 0 in the $\text{commands}$ element, your spatial map is a PR quadtree and therefore does not support linear data. In that event, the error message is:

**Spatial map does not support linear data**

You must also check two other conditions: first, that this new edge does not intersect any other edge already created except at their endpoints (this is a requirement of the PM quadtree family). Also, you must check that inserting this edge into the PM quadtree (if it’s a PM1 or PM2) will not cause the tree to be partitioned beyond the legal depth. The respective error messages for these events are as follows:

Edge (<$\text{start}$>,<$\text{end}$>) intersects existing edge <$\text{e2start}$, $\text{e2end}$>)

Edge (<$\text{start}$>,<$\text{end}$>) requires partitioning the quadtree deeper than allowed

where $\%\text{e2start}$ and $\%\text{e2end}$ are the names of the start and end points of the pre-existing edge which is intersected. These values should be asciibetical such that $\%\text{e2start}$ is asciibetically less than $\%\text{e2end}$.

- **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 (<$\text{start}$>,<$\text{end}$>) 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 (<$\text{start}$>) does not exist
End point (<$\text{end}$>) does not exist
Edge (<$\text{start}$>,<$\text{end}$>) does not exist

where $\%\text{start}$ and $\%\text{end}$ are the values of the start and end attributes, respectively.

If $\text{pmOrder}$ has been set to 0 in the $\text{commands}$ element, your spatial map is a PR quadtree and therefore does not support linear data. In that event, the error message is:

**Spatial map does not support linear data**

- **clearAll** – resets all of the structures, clearing them. This has the effect of removing every dot. This command cannot fail, so it should unilaterally produce a $\%\text{success}$ 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 $\%\text{success}$ tag will be produced in the output XML. This $\%\text{success}$ tag will have no message attribute. The resulting $\%\text{success}$ tag will have one child element, $\%\text{dotList}$. A $\%\text{dotList}$ has one or more child elements of type $\%\text{dot}$. A sample $\%\text{success}$ for this command:

```xml
<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.compareTo()` 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**

- **enqueue** – adds a string to your Fibonnaci heap. Note that if you implement the f-heap you should create a new, empty f-heap when your program starts that we will interact with in this way. (you could choose to create it when the first string is added).

  This element has one attribute, `string`, whose value is the string to enqueue into the f-heap. Your f-heap is a min-heap which will enqueue strings in ascibietical (per `java.lang.string.compareTo()`) order.

  Duplicate strings are allowed in the f-heap, so this command can only fail if `string`’s value is illegal (i.e., doesn’t match the appropriate regular expression).

  Aside from a faulty attribute value, this command cannot fail, so it should produce a `success` tag with the following message:

  String "<string>" successfully enqueued

- **dequeue** – dequeues the first item in the f-heap. This command can fail if the f-heap is empty. The error message for this event is:

  Priority queue is empty

  Otherwise, a `success` tag is otherwise generated with the following message:

  String "<string>" sucessfully dequeued

  where `<string>` is the value of the first item in the f-heap (the item that was dequeued)

- **printQueue** – prints the structure of the f-heap. The output structure for the f-heap is as follows:

  - `fibanacciHeap` – the root of the f-heap XML. This tag has one required attribute, `cardinality`, which specifies how many items are currently enqueued in the heap. This element has one or more child elements of type `node`.

    - `node` – a node in the min-heaps stored by the f-heap. The `node` tag has one required attributes, `value`, whose value is the enqueued string at that location in the f-heap. The `node` element can have 0 or more `node` child elements.

Here is an example of the f-heap from figure 1 at the Wikipedia coverage of the Fibonacci heap at:

http://en.wikipedia.org/wiki/Fibonacci_heap

```xml
<fibanacciHeap cardinality="9">
  <node value="1">
  <node value="3" />
  <node value="4" />
  <node value="7"/>
</fibanacciHeap>
```
• **printSpatialMap** – prints the PM/PR quadtree. Since PM and PR quadtrees are deterministic via input, your XML should match exactly the primary input/output. The output structure of the PM/PR is as follows:

  - **quadtree** – the root of the quadtree XML. This tag has one required attribute, order, which should be the value of the pmOrder attribute in the pmCommands tag for the given input.
  
  - **grey** – a grey node. A grey node has four child nodes (either grey, black, or white). A grey node has the following required attributes: x – the x coordinate of the center point of this partition y – the y coordinate of the center point of this partition
  
  - **black** – a black node. A black node has either dot or edge children (as defined by use in other commands). It has one required attribute, cardinality, which indicates how many geometric primitives are contained in this region (i.e. the sum of the number of dots and the number of edges in this region).
  
  - **white** – a placeholder node which has no attributes and no child nodes.

• **printBPTree** – prints the structure of your B+ tree. Since your B+ tree will be used in place of TreeMaps for your data dictionary (that maps strings to Dots), the B+ tree printed by this command should print the B+ tree containing all of the dots created so far. Your B+ tree is a map of Strings to Dots, so your keys should be strings and your values should be dot objects, which we will print in the XML as coordinate pairs (x,y). This command cannot fail, so it produces a non-empty success tag. This success tag will have one child element, bptree. The bptree tag will print out structure of your B+ tree. Since the structure of a B+ tree is nondeterministic, we will not be able to diff the output of your XML files for this part of the project. Instead, we will use a tester class which examines the structure of your bptree element and determines whether it firstly contains all the data it should contain and secondly that it is a legal B+ tree. The structure of the bptree tag is listed in the DTD, but more specifically, the XML structure should be as follows:

  - **bptree** – the root of the B+ tree xml. This tag has four required attributes: cardinality, whose value should be the size (number of keys) held by the B+ tree; height, the number of levels in the B+ tree; and order, the branching factor of this B+ tree, and leafSize, the number of keys per leaf in this B+ tree.
  
  - **guide** – represents a guide node in the B+ tree. This element has no attributes. Its children should alternate between (either leaf or guide) and key, which logically expresses the configuration of a guide node. Clearly, a guide node’s children are going to be key nodes and either all guide nodes or all leaf nodes.
  
  - **leaf** – represents a leaf node in the B+ tree. A leaf has no attributes, and contains entry child nodes corresponding to the key/value pairs mapped by this B+ tree.
  
  - **key** – represents an index key in a guide node. A key node must appear between either two guide nodes or two leaf nodes. The key node has one attribute, value, which represents the actual key.
  
  - **entry** – represents a key/value pair as exist in leaf nodes. The entry element has two attributes, key, the key; and value, the value. Because our B+ tree will map dot names to dot objects, the key
attribute's value should be the dot name and the value attribute's value should be the coordinates of that dot, printed as "(\(\hat{x}_i, \hat{y}_i\))" where \(\hat{x}_i\) and \(\hat{y}_i\) are the x- and y-coordinates of that dot, respectively.

An example of a B+ tree XML output will be provided.

- **nearestPointToSegment** - requests the dot whose geometric distance to the given segment is minimal, excluding the segment's endpoints. The segment will be given by a start vertex and an end vertex. The attributes for this command are start and end, whose values refer to dots by name. A semantic error can only occur if no dots have been created in addition to the end points of the given segment. In this event, an error tag should be produced with the following message:

  No candidate dots are present in the spatial map

If successful, a success tag is generated with the following message:

The closest dot to \(<\text{start}, \text{end}>\) is \(<\text{result}>\) with distance \(<\text{distance}>\)

where \(\hat{x}_i\) and \(\hat{y}_i\) are the values of the start and end attributes, respectively, and \(\hat{result}_i\) is the name of the located nearest dot, and where \(\hat{distance}_i\) is the floating-point geometric distance value (rounded to 3 decimal places).

Note that it is possible that an infinite number of dots with the same geometric distance from the given segment could exist in the map. In this event, return the dot whose name is lexicographically smallest.

As always, this command can fail if the specified segment is for some reason invalid, as per the same rules as other commands involving edges. 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 \(<\text{start}>\) does not exist
End point \(<\text{end}>\) does not exist
Edge \(<\text{start}, \text{end}>\) does not exist

where \(\hat{x}_i\) and \(\hat{y}_i\) are the values of the start and end attributes, respectively.

- **nearestSegmentToPoint** - requests the segment whose geometric distance to the given segment is minimal. Generally the distance from a point to a segment is the length of the segment orthogonal to the target line which intersects the target point and the target segment at some arbitrary point. In the event that such an orthogonal segment would intersect the target line at a point not on the segment, the distance is determined by the length of the segment connecting the appropriate endpoint and the target point. This command has two attributes, \(x\) and \(y\), integers, which define the target point.

This command can only semantically fail if there are no segments defined. This command is allowed to consider segments for which the point given by \((\hat{x}_i, \hat{y}_i)\) is an endpoint.

Otherwise, as long as the point is valid, and there is a candidate edge, this command should produce a success tag with the following message:

The closest edge to \(<x, y>\) is \(<\text{start}, \text{end}>\) with distance \(<\text{distance}>\)

where \(\hat{x}_i\) and \(\hat{y}_i\) are the values of the \(x\) and \(y\) attributes, respectively, and where \(\hat{start}_i\) and \(\hat{end}_i\) are the start and end point names of the located nearest segment, and where \(\hat{distance}_i\) is the floating-point geometric distance value (rounded to 3 decimal places).

Note again that it is possible for an infinite number of line segments to be equidistant from the given point. (Consider a circle as defined by being an infinite collection of infinitely small tangential segments, for instance; also, the centroid of an equilateral triangle comes to mind...) In this event, return only
the segment whose name, according to edge asciibetical sorting rules, comes first, e.g. (A,B) would be returned instead of (A,C).

The point given by \( \langle x,y \rangle \) does not necessarily have to be in bounds (accroding to the bounds of the spatial map). That means even if your area is only 1024x1024, I should be able to pass in the point (32767,32767) and find the nearest segment. This stipulation shouldn’t affect your algorithm, but if it does require that you provide a point in the bounds of the region, consider a line which passes through the origin and the target point. Consider the point at which this line intersects the bounding region of the spatial map (on the appropriate side) and use that point.

- **rangeSearch** – requests a radial range search of your spatial map. This command will succeed if at least one dot or edge exists in that range, in which case it will produce a non-empty \( \text{success} \) tag. The rangeSearch tag has three attributes, \( x \) which specifies the \( x \)-coordinate of the search circle’s center, \( y \) which specifies the \( y \)-coordinate of the center, and \( \text{radius} \) which indicates the radius of the search circle. The \( \text{success} \) tag for this method will contain a single child element, \( \text{geometryList} \), which can contain \( \text{dot} \) and \( \text{edge} \) child elements as defined by other commands. This \( \text{geometryList} \) should contain only the geometry which fall inclusively inside the search circle (inclusively means that dots falling exactly on the edge of the circle or edges that are tangential to the circle will be included in this list).

The \( \text{geometryList} \) element should list all dots followed by all edges. Dots should be listed in asciibetical order according to name. Edges should be listed asciibetically first by its start vertex and next by its end vertex. Edge start and end vertices, as always, should always maintain the invariant that start vertex name is less than or equal to end vertex name (asciibetically).

If the range search does not return any objects, an \( \text{error} \) tag should be produced whose message is:

**No geometry exists in the given range**

- **shortestPath** – prints the shortest path between the dots named by the \( \text{start} \) and \( \text{end} \) 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 \( \text{success} \) element in response to this command is empty and has no \( \text{message} \) attribute. It has one child element, \( \text{path} \), which itself has a number (at least one) \( \text{edge} \) elements as children. The \( \text{path} \) element has two attributes which must be set: \( \text{length} \) which reports the length of the total path rounded to 3 decimal places, and \( \text{hops} \) which indicates the number of unique \( \text{edges} \) travelled to get from the start to the end. A sample of a \( \text{success} \) tag for \( \text{shortestPath} \):

```xml
<success command="shortestPath">
  <path length="10" 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 \( \langle \text{start} \rangle \) does not exist
- End point \( \langle \text{end} \rangle \) does not exist
- No path from \( \langle \text{start} \rangle \) to \( \langle \text{end} \rangle \) exists

50
5.6.2 Input DTD

<!--COMMANDS (the Root element of the XML) -->
<!ELEMENT commands (createDot|deleteDot|createEdge|deleteEdge|enqueue|dequeue|printQueue|printSpatialMap)>
<!ATTLIST commands
autoGraph (true|false) #REQUIRED
bpOrder CDATA #REQUIRED
leafSize CDATA #REQUIRED
pmOrder (-1|1|2|3) #REQUIRED
spatialWidth CDATA #REQUIRED
spatialHeight CDATA #REQUIRED>

<!--CREATEDOT -->
<!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>

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

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

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

<!--CLEARALL -->
<!ELEMENT clearAll EMPTY>

<!--LISTDOTS -->
<!ELEMENT listDots EMPTY>
<!ATTLIST listDots
sortBy (name|coordinates) #REQUIRED>
<!-- ENQUEUE -->
<!ELEMENT enqueue EMPTY>
<!ATTLIST enqueue
string CDATA #REQUIRED>
>
<!-- DEQUEUE -->
<!ELEMENT dequeue EMPTY>

<!-- NEARESTPOINTTOSEGMENT -->
<!ELEMENT nearestPointToSegment EMPTY>
<!ATTLIST nearestPointToSegment
start CDATA #REQUIRED
end CDATA #REQUIRED>
>
<!-- NEARESTSEGMENTTOPoint -->
<!ELEMENT nearestSegmentToPoint EMPTY>
<!ATTLIST nearestSegmentToPoint
start CDATA #REQUIRED
end CDATA #REQUIRED>
>
<!-- PRINTBPTREE -->
<!ELEMENT printBPTree EMPTY>

<!-- PRINTSPATIALMAP -->
<!ELEMENT printSpatialMap EMPTY>

<!-- PRINTQUEUE -->
<!ELEMENT printQueue EMPTY>

<!-- RANGESearch -->
<!ELEMENT rangeSearch EMPTY>
<!ATTLIST rangeSearch
x CDATA #REQUIRED
y CDATA #REQUIRED
radius CDATA #REQUIRED>
>
<!-- SHORTESTPATH -->
<!ELEMENT shortestPath EMPTY>
<!ATTLIST shortestPath
start CDATA #REQUIRED
end CDATA #REQUIRED>
5.6.3 Output DTD

<!ELEMENT results (success|error)>*

<!ELEMENT success (bptree|dotList|fibonacciHeap|quadtree|path|geometryList)?>
<!ATTLIST success
command CDATA #REQUIRED
message CDATA #IMPLIED>

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

<!ELEMENT geometryList (dot|edge)>*

<!ELEMENT dotList (dot)>*
<!ELEMENT bptree (guide|leaf)>*
<!ATTLIST bptree
cardinality CDATA #REQUIRED
order CDATA #REQUIRED
leafSize CDATA #REQUIRED
height CDATA #REQUIRED>

<!ELEMENT guide (key|guide|leaf)>*
<!ELEMENT leaf (entry)>*
<!ELEMENT key EMPTY>
<!ATTLIST key value CDATA #REQUIRED>
<!ELEMENT entry EMPTY>
<!ATTLIST entry
key CDATA #REQUIRED
value 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 fibonacciHeap (node)>*
<!ATTLIST fibonacciHeap cardinality CDATA #REQUIRED>
<!ELEMENT node (node)>*
<!ATTLIST node value CDATA #REQUIRED>

<!ELEMENT quadtree (grey|black)>*
<!ATTLIST quadtree order CDATA #REQUIRED>
<!ELEMENT grey (grey|black|white)>*
<!ATTLIST grey
5.7 Miscellaneous Addendums

1. In the event that the spatial map is empty, generate an error when printing. The message is: "Spatial map is empty"

2. In the event that the pqueue is empty, generate an error when printing. The message is: "Queue is empty"

3. In the event that there are no segments for the nearestSegmentToPoint command, the message is: "No candidate segments are present in the spatial map"

4. The (x,y) for nearestSegmentToPoint only have to valid according to the spec's definition of numbers (i.e. the regexp). The numbers do not have to be in bounds according to the bounds of the spatial map.

   Standard error is produced for invalid attribute value. I thought I made this one clear?

5. Output DTD: <!ELEMENT success (bptree—dotList)?_j |ELEMENT grey (grey—black—white)_j |ELEMENT black (dot—edge)_j

   replace with

   <!ELEMENT success (bptree—dotList—fibonacciHeap—quadtree)?_j |ELEMENT grey (grey—black—white)_j |ELEMENT black (dot—edge)_j

   I am pretty sure the DTD parser will be able to handle that sequence correctly, if I coded it right. ;) If you run into trouble with that, post to the NG. The contingency for the black node will be: 

   <!ELEMENT black (dot—edge)_j

   p.s. the operator in DTDs is denotes a sequence.

6. I may not have made this clear for the pblack node - print all pdot items in ascibetical order (there should be only one), then print all pedge items, ascebitical as has been previously defined.

7. The graph is undirected. There is no semantic difference w/ dijkstra's between part1 and this project,

   just that you will use an heap for your shortest path instead of a logarithmic queue.

8. You can do whatever you want with edges in this project - you can use your adjacency list or come up with some other scheme. Just make sure it's quick. I wouldn't recommend the PM, although in theory it can work nicely - you will always find neighbor edges in the same region as the endpoint (so if you traverse edge AB and then want to quee up B's neighbors, all of B's neighbors are in the black node containing B).

9. clearAll: I just recycled the message. it should clear the heap as well. I didn't explicitly say that clearing should remove all the edges but I feel like that's implicit when I say all dots are removed. ;) To be crystal clear, remove all dots, remove all edges, and remove everything from every data structure. clearAll is supposed to reset the state of your command parser to as if it had just finished reading the commands tag and is ready to input it's first actual command.

10. listDots: go by the DTD, the value of the attribute must be "coordinate"

11. Good point about the double quotes. The messages for the f-heap should instead be: *enqueue* 

"String 'string' successfully enqueued" *dequeue* "String 'string' successfully dequeued"
12. Error message for empty B+ tree: "B+ tree is empty"
13. In the event that the createDot command creates a dot that falls on a pre-existing line segment thus
 intersecting it, error is: "Dot at (x1,y1) intersects existing edge pstart,pend;" 
14. The spatialWidth and spatialHeight attributes are always legal as long as they aren’t negative (and
 presumably neither are 0, although in theory you could still make a very limited spatial map out of that). I
 think the regexp given in the spec might account for negative numbers but we haven’t dealt with them
 and we won’t – if the values are negative or not numbers use the standard ”illegal attribute” error message.
15. B+ trees must be correct for order 3 (at least 3 keys per guide). The order is invalid if less than 4.

6 General Policies

6.1 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 cmosc420 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.
6.2 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 differ similar 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 differ 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 for 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.

6.3 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 manner. A submission that gets only 20 or 30 points is considerably better for you than no submission at all.

6.4 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 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 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.

6.4.1 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 ACCESS 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.