1 General Description

You work for your country, the United States of Meesh, as chief military strategist. You will be using predefined data structures to build specific portions of a military planning application. Your country requires you to design the most realistic and efficient war simulator. Initially, you will be using a dictionary data structure to store both bases and targets. Later, it will be extended to store any object we require. You are also responsible for storing records of the exact inventory each base contains. Later, new objects will be mappable such as roads, hospitals, airports, civilian cities, mobile forces and more. Also, you will have firing capability. This brings the additional challenge of damage assessment based on the range of weapons, power of a weapons, and much more. We will replace some data structures with new ones in order to better suit the problem, optimize the efficiency, or at least give you experience with some advanced data structures.

The project comprises four parts, each part building on its predecessors, and challenges you to develop a wartime targeting system based on a long, volatile, and occasionally complex specification. For example, some of the data structures from part 1 will be expanded upon in part 2. In part 3, you will add deletion to the data dictionary from part 2, and you will expand the spatial data structure from part 2 to map roads and weapons targeting tracks (graph or polygon edges), as well as the Sites of part 2, and additional class of objects, Loci. All Sites are Loci, but not all Loci are Sites.

In some circumstances, you will be replacing data structures from earlier parts with more efficient data structures in later parts. SoftWar functions will be added with each part, resulting in a very powerful piece of software at the end of the semester.

1.1 Disclaimer

This is a large project that has been carefully designed over several years to provide a challenge to every level of programmer in the class. Each part will take the full time you have been allotted to complete. (In all honesty, many of you will feel that we have not given you enough time—yes, even you hot shots.) But don’t panic! There are plenty of ways to complete each part on time for full credit.

Here’s the secret: start early and start by spending a few days sketching out what objects/classes you will use and how these objects/classes will fit together. You are encouraged to discuss your ideas with classmates (but don’t write any code together!) and to bring your design ideas to the TAs during office hours, or on the newsgroup, for comments/suggestions. The most successful students have spent time carefully planning

* Participation in this project may prove HAZARDOUS to your health. Unfortunately, failure to participate early and often will definitely have an adverse effect upon 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.
their projects. Those students who do not devote any time to design are the ones most likely to receive a poor grade.

Many of you have probably been able to sit down and do many of the projects in the lower level classes in an evening or two with little or no planning. Well the honeymoon is over. If you don’t start early and spend time carefully planning each part, you will be very hard pressed to get a fully working solution. A sustained effort involving good design, testing and regular mental health breaks is the key to success in Dr. Hugue’s 420 class. Mountain-Dew-Code-A-Thons the night before the due date pave the road to failure.

2 Project Components

There are four major components to this project each of which will upgraded with each part: a dictionary data structure, a spatial data structure, an Fibonacci heap, and a mediator (command parser).

2.1 Dictionary Data Structure

A dictionary data structure is one which is capable of storing objects in sorted order based on key such as a string or an integer. For instance, say you have several hundred Base objects which consist of the name of the Base, the longitude and latitude at which it is located, and its armaments. One way of storing these cities is to sort them by name; another is to store them in decreasing order by armaments; yet another is in increasing order by longitude. Primarily, the dictionary component will store objects based on some sort of a string key, such as the name of a Base or Target. The main purpose of the dictionary is to provide us with an easy way to see what data points we have already entered into our SoftWar database. The term data dictionary will be used in class to refer to this component, or to a collection of components having this role.

2.2 Spatial Data Structure

A spatial data structure is a data structure which is capable of storing and sorting objects based on multi-dimensional keys. The two-dimensional structures, such as the PR quadtree and PM1/PM3 quadtree, can store objects based on two values, such as the longitude and latitude \((x, y)\) of a base. Three-dimensional structures, such as a K-d tree or a PM1 octree, can store objects based on three values, such as the longitude, latitude, and sea level \((x, y, z)\) of a base.

2.3 Fibonacci Heap

You will be implementing a min heap. This will be used for storing item objects, which have a name field and corresponding metric, which will be used as the key. I suggest that you use a min pointer which will always point to the min value on the top list of nodes. The total number of items could also be useful. You should also designate the head and tail of the top node list. This could be used in the print function to decide which node to call the function with. It is important to differentiate when you have printed each node once.

For insert, add the new element as a root in the forest of binary trees and update the MIN reference as needed. For delete min, remove the node with the minimum value, add its child subtrees to the root-level of the f-heap, update the MIN reference and then consolidate by combining binomial heaps of like degree until at most one of each degree is present at the root level. For union, add one heap to the root level of the other as in an insert, and update the MIN reference. For consolidate, combine binomial heaps at the root-level of like degree until at most one of each degree remain. For merge, take the union of the two heaps, update the MIN reference and then consolidate. As long as your f-heap satisfies the min-heap property; its component subtrees satisfy the structural requirements of binomial trees; and you have printed it out as described in the print heap section, you should receive full credit.

Although the linked-list implementation described in the Fibonacci tree references is not efficient for a JAVA implementation, you are neither required to use it nor will you be penalized for not doing so.
2.4 Mediator

A Mediator is a design pattern that is described in the famous book Design Patterns by Gamma et al, also referred to as the Gang of Four (GOF) book.[1]

The intent of a Mediator is to **define** an object that encapsulates how a set of objects interact. Mediator promotes a loose coupling among objects by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently\(^1\).

In other words, the idea is to have one or more objects (hint: more!) which are capable of reading in commands from the input stream and executing them by calling the right functions in the dictionary, spatial, and adjacency list data structures to perform the requested action(s).

Minimally, the Mediator could be a class named CommandParser which would read commands from the standard input stream, parse the data, pass it onto the correct component for further processing, analyze the return values from this component, print the correct success or failure message back to the user, and then loop until the EXIT() command is read.

It would be wise (hint hint) to break this functionality into several classes which perform one or more of these tasks. These objects, when combined together, form the abstract notion of a Mediator for our Software program.

2.5 Adjacency List

The polygonal maps in the name of the PM1 quadtree are essentially finite graphs (vertices and edges), which we will call **loci** (plural of **locus**) and **tracks**. They are stored in the PM1 quadtree to support identification of the closest track to a given point in 2-space. However, the method of capturing vertex-edge relationships in the PM1 is not appropriate for finding the shortest set of edges between two vertices, or other applications that explore the connections between a vertex and all vertices adjacent to it. So, you will also be implementing adjacency lists to reflect combinations of tracks (edges) that have been inserted into the PM1 quadtree.

There are many ways to implement an adjacency list—just an array or arrays or the simple matrix that you may have learned in the lower levels courses! You can do a skiplist of skiplists, or a TreeMap of TreeMaps, or any number of various objects which provide the right traversal capabilities. The principal requirement for this project is that insert and search for the loci connected to a specific locus must be in \(O(\log n)\).

Regardless of the implementation of the adjacency list, its purpose is to provide us with a quick and easy way to perform shortest path calculations that are impractical if the adjacency list must be derived from the quadtree representation of the graph.

3 Roadmap

This is a roadmap of the major component that we will use in each part of the project. An asterisk (*) indicates that we will be reusing the structure from the previous project with little or no modification. "i" stands for insert, "s" stands for search, and "d" stands for delete. (The command parser will have new commands added with each project, but the overall design need not change after project 1 unless you want to make it more efficient or elegant.)

<table>
<thead>
<tr>
<th>Part</th>
<th>Dictionary</th>
<th>Spatial</th>
<th>Inventory</th>
<th>Adj. List</th>
<th>Mediator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TreeMap (i/s/d)</td>
<td>TreeMap w/Comp(i/s)</td>
<td>Fib Heap(i/d/min/mg)</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B+ tree (i/s)</td>
<td>PR Quadtree (i/s/d)</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>B+ tree (i/s/d)</td>
<td>PM1 Quadtree (i/s)</td>
<td>*</td>
<td>O((\log n)) insert</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>PM1 Quadtree (i/s/d)</td>
<td>*</td>
<td>O((\log n)) i/d</td>
<td>*</td>
</tr>
</tbody>
</table>

\(^{****}\text{NOTE: This roadmap is subject to change as the semester progresses.****}\)

\(^1\)While this design pattern is presented on page 273 of [1], a Google search using "design patterns mediator" is cheaper than buying the book.
4 General Notes on Java

This semester all projects are to be written in Java. The version on the detective cluster is 1.4 and can be downloaded at:

http://java.sun.com/j2se/1.4/download.html

The online version of the documentation is at

http://java.sun.com/j2se/1.4/docs/api/index.html

We highly recommend that you download the Java sdk and do most of your work from home—if nothing else this will lighten the load on the now overworked DC machines, which can become very slow as lower-level project due dates approach. However, you should be aware that this approach has inherent risks.

As in most CMSC classes, your projects will be graded according to how they execute on the detective cluster machines (dc.umd.edu). In fact, you will not be able to submit your CMSC 420 projects for grading unless they compile and execute the primary data correctly. In fact, you will receive no credit for the project if you are unable to submit. However, since the submit is typically a proper subset of the test data for the prior part of the project, there is little excuse for being unable to pass those tests.

While there should not be any portability issues long as you develop your code using the correct Java version, it’s still a good idea to download your code regularly to be sure that a copy of your project exists on the cluster and works. In fact, you should make regular verification that your program executes correctly on the target platform as part of your design and development process.

Whether working on your home machine or on the detective cluster, it is your responsibility to protect your code from silly human errors, such as typing \texttt{rm *.} One way is to take a few minutes and add commands to your makefile such as \texttt{make clean} (to remove old files safely) and \texttt{make submit} (to prevent a bad tar command from overwriting your main file). Another, more formal way, is to use CVS to manage the evolution of code in your project development process.

While there should not be any portability issues long as you develop your code using the correct Java version, it’s still a good idea to download your code regularly to be sure that a copy of your project exists on the cluster and works. In fact, you should make regular verification that your program executes correctly on the target platform as part of your design and development process. You should make regular backups of your work to ensure that your grade will not suffer from any catastrophic failure of your home system. Similarly, it never hurts to make your own backups of code developed on the detective cluster, because their backups are infrequent, and restore may not be timely relative to the project due date.

We’ll eventually be working with graphics, and if you use ssh to access the detective cluster from home, you will have to figure out how to set up an X-server to support drawing on your own machine. The difficulty of this task will vary depending on your computer’s configuration. So, even if you use JAVA on your local machine, and don’t have to worry about messing around with an X-server, you should visit an on-campus machine to test your drawing functions.

4.1 IDEs

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

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

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

Do a Google search to find the URLs to download these IDEs or look for them on the WAM machines (I have no idea which of these are installed on the UMD networks 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.

4.2 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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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 :)

### 4.3 Comparators

JAVA has two basic tools for doing comparison: the `compareTo()` method of `Comparable` objects; and, the `compare()` method of the `Comparator` class. We will try to explain this with a quick example. Suppose you wanted to sort a collection of strings in alphabetical (ignoring case for the moment) order. One might use a `TreeSet` to do this:

```java
SortedSet sorter = new TreeSet();
sorter.add("hello");
sorter.add("world");
sorter.add("cat");
sorter.add("dog");

Iterator i = sorter.iterator();
while(i.hasNext())
    System.out.println(i.next() + ""); //prints "cat dog hello world"
```
In the above example the string constants are automatically cast to String, which implements the compareTo() method (just like c/c++ strcmp). The sorted map assumes its elements are Comparable and uses compareTo() to sort them. Note that this is a unique case where i.next() can be used without a cast, since in JAVA ALL objects implement the toString() method which is automatically called here. Back on topic, what if I wanted to sort the words backwards? One way is to wrap the strings in another class that implements compareTo() backwards like:

class MyString
{
    String s;
    MyString(String s){this.s=s;}
    public int compareTo(Object other){return -1*(s.compareTo(other));}
}

This would work, but it is a bit of a mess. We can’t extend String directly, since Sun has made it a final class. In any case it will not be obvious what it is you are doing. However, there is a better way that you can use with all of JAVA’s sorted classes (and which you will implement in your own sorted maps later this semester). You can use a comparator:

class ReverseCompare implements Comparator
{
    public int compare(Object a, Object b)
    {
        return(-1*((Comparable)a).compareTo(b));
    }
}

SortedSet sorter = new TreeSet(new ReverseCompare());
sorter.add("hello");
sorter.add("world");
sorter.add("cat");
sorter.add("dog");

Iterator i = sorter.iterator();
while(i.hasNext())
    System.out.print(i.next()+" ");//prints "world hello god cat"

Because the TreeSet was given a Comparator in its constructor, it will no longer assume its elements are Comparable, and will use the Comparator for sorting instead. Comparators allow you to easily have sets with different types of objects which aren’t natively comparable with each other, or to impose your own sorting rules on other people’s classes like String) with ease. Cool stuff. You’ll hopefully find this useful for doing your coordinate checking in Part 1, as well as implementing a priority queue for your adjacency lists.

4.4 Interfaces

One of the goals of object oriented programming is to create modular code that can easily be reused for various tasks. One good way to achieve this goal in JAVA is to use an interface.

An interface defines a set of public methods that a class must implement. The joy of using an interface is that if two different data structure implement the same interface, you should be able to switch freely between them without any problems!

Consider two classes: LinkedList and TabularList (both reside in the java.io package). The TabularList class stores values using a private tabular while the LinkedList uses dynamic memory management. Both classes, however, implement the List interface, which allows for the following:
List list;
TabularList tabularList = new TabularList();
LinkedList linkedList = new LinkedList();

list = tabularList; // compiler casts these for us automatically
list = linkedList; // because both implement the List interface

The real joy of interfaces is that you don’t need to worry about the implementation of the class - you only need to know what the interface is. Say that you wrote a large program that has a large dictionary of information and you decide to store this information using the java.util.LinkedList class. You could pass this variable around in your program as a LinkedList, but you realize that you don’t really care about the fact that you are using a *linked* list, you only care about the fact that you are using a List. So you pass the dictionary around as a List object (since LinkedList implements the List interface).

public static void main(String[] args) {
    List masterDictionary = new LinkedList();

    // ... now 400,000 lines of code that does something fun and profitable
}

After 5 months of hard work you ship your product but your customer comes back to you and complains that the program is too slow - the O(n) access time of your LinkedList isn’t fast enough! Thankfully you had the foresight to pass the LinkedList around as a List object, so you can quickly replace the LinkedList object with any other class that implements the List interface. You just happen to have a SkipList class in your code base that implements the List interface. By changing a single line of code you are able to drastically improve performance without the headaches of having to search and replace for every instance of LinkedList.

Of course, this only works if you had the foresight to use interfaces. ;) Thus, we will help you get comfortable with design-by-interface by requiring you to implement a few of them.

5 Command Parsing Guidelines

You are all blessed with a professor with a Ph.D. in fault tolerance. Because of this you will be expected to implement some error checking mechanisms in your programs. This means bounds checking for input numbers and checking a number of possible error conditions for each command that your Mediator parses.

It also means that your Mediator/parser should never fail or crash because of a malformed command, or because a correctly formatted command that you did not implement is present in the inputs. A really useful command interpreter would give useful error messages about commands, such as "wrong number of arguments", or "invalid argument type", or even better "second argument was int, expected string". For our purposes it will be sufficient to print a single error message regardless of the error:

*****
Error: Invalid Command

Note that the asterisks (*****are printed. Do NOT echo the erroneous command. You will only echo valid commands. This includes ALL valid commands at this level of abstraction, even if you detect an error in the site name later.

Your parser must ignore blank lines completely. For all other lines which are not fully formed and correct commands you must print the above error. As with other parts of the project, your fault tolerant command parsing capabilities will be tested separately from the remaining requirements of the project. So if you can’t get a fully error checking parser working you will not be hurt on the other parts of the project. You may assume that for commands other than error checking your parser, all commands will be upper case and there will be no spaces within a command. There will be no blank lines and all commands will be valid. We’re not out to get you.
However, to receive full credit for fault tolerant command parsing, a working parser cannot make any of the assumptions described in the previous paragraph. Commands should be correctly interpreted regardless of case (i.e. CREATE_BASE and create_base should both be interpreted as the CREATE_BASE command). Note that for the Spring, 2004, edition of the course, all commands should be parsed to upper case. So create_base(CollegePark, 5, 6) should be parsed as CREATE_BASE(COLLEGE PARK, 5, 6). This is very easy to do using the String.toUpperCase() command on the input line. Blank lines should be ignored (in particular do not print extraneous ****'s). Whitespace, in general, should be ignored when parsing with one exception - any string (command name or string argument like base name) cannot contain internal spaces. So a command like create_base(College Park, 5, 6) should be flagged as invalid (a valid version might look like create_base(CollegePark, 5, 6)).

Please note that we must require you to adopt our error message format (and content) to support online testing. The size of the class and the complexity of the project make semi-automated project grading essential. So, you will be required to print out information in a way that would be appropriate only for debugging or grading purposes, making it easy to identify correct behavior in the presence of incorrect data, and vice versa.

5.1 Input

Many of the commands have parameters that are strings, integers, or doubles. The following rules apply regarding valid formatting of these types.

5.1.1 Strings

String parameters will satisfy the following regular expression:

```
[A-Za-z0-9]+  
```

Any string that does not satisfy this regular expression should be considered invalid and will cause an <InvalidCommand> message to be printed. Please note that all strings are CASE INSENSITIVE. That is to say that when a parameter is said to be a string type, it is case insensitive. Strings should be sorted in alphabetical order. (HINT: The String.compareTo() method does this by default!)

5.1.2 Integers

Integer parameters will satisfy the following regular expression:

```
-?[0-9]+  
```

Any integer that does not satisfy this regular expression should be considered invalid and will cause an <InvalidCommand> message to be printed. (HINT: Integer.parseInt())

5.1.3 Doubles

Double parameters will satisfy the following regular expression:

```
-?[0-9]+([.][0-9]*)?  
```

Any double that does not satisfy this regular expression should be considered invalid and will cause an <InvalidCommand> message to be printed. (HINT: Double.parseDouble())

5.1.4 Parameter Delimiting

Each parameter will be separated by zero or more spaces, a comma, and zero or more spaces. Thus, the parameter delimiter matches the following regular expression:

```
[ ]*[,][ ]*  
```

Please note that all parameters may be padded with whitespace like so:
CREATE_BASE(  CollegePark,  3,  5)

  but this should be processed as if it were:

CREATE_BASE( CollegePark, 3, 5)

  Also, certain commands may contain no parameters. In this situation, there may be zero or more white spaces between the parentheses like so:

PRINT_DICTIONARY(  )

  But this command should also discard the whitespace and process identically as if it were

PRINT_DICTIONARY()

5.2 Output

For all valid commands, you should print "*****\n" followed by a " => " and an echo of the command given. For instance, the entire valid output to EXIT() is

*****

=> EXIT()

mission complete

  The sample output should make this clear. This is done to negate the effects of input redirection and to assist in grading. Note that although it is done in the samples that will appear later, you are not required to reformat the original command (fixing spacing, for instance) in any way. And, you are required to satisfy the specification as written, and not merely the sample output, which might have errors.²

  At certain times multiple error messages may be applicable. In this situation, the left most error message in the BNF takes precedence. So if you have a list of errors, eg:

<error>:=<DoesNotExist>|<AlreadyExists>|<RoadAlreadyExists>

  and all of the errors are applicable to this situation, you would only print the <DoesNotExist> error since it is the left most error in the BNF.

5.3 Invalid Commands

In any case where there is an invalid command, either due to an unknown command name, an invalid parameter, missing parameters, or extra parameters the <InvalidCommand> message should be printed. If a command is valid, then the output for that command should match the BNF described for that command, unless some exceptional condition arises. Exceptional conditions are those which differ from correct behavior, but were not anticipated in the BNF spec. Rather than leave ourselves unprotected from unexpected error conditions, the <InvalidCommand> message should be printed as the default error message.

<InvalidCommand>:=Error: Invalid Command

5.4 A Warning from One Who Knows

I would encourage you as strongly as possible to consider software design using assertions or explicit testing of your preconditions and post conditions as a testable project requirement. Why? Because the scope of the project makes integration and testing difficult unless you have made some effort to help the debugging process a meta-rule in your design. In fact, failures to complete the project in a timely fashion are almost always traceable to sloppy or not-existent design methodologies in use by the programmer, and not mere laziness. Learn to exploit the JAVA IDE's and methods such as design pattern usage, and J-Unit testing.

²After all, in real life, it's the summer intern or student worker who is usually given the task of creating test data.
Learn to let software or the JAVA API do the work for you, instead of insisting on rolling your own JAVA code.

Until you master the guts of the JAVA virtual machine, and understand the semantics of its memory model and byte-compilation of your code, I sincerely doubt that your hand-rolled My-class-api will show improvement over an equivalent portion of the API. The computational model you’ve used in programming in C and C++ is, in many cases, totally inappropriate for efficient JAVA programming. So your “gut feeling” for how things should work in JAVA is often wrong if you have not explored the weaknesses and strengths of the JAVA framework.

6 Part 1: TreeMaps, Fibonacci heaps, and a Mediator

By popular demand, you are hereby directed to the frozen version on the website. After all, if there is no bogus stuff in the spec, you can’t accidentally use the early spec.

7 Part 2: Robust Site and Coordinate Databases: B+ tree and PR quadtree

In Part 2 we will be upgrading our data structures to allow for a greater variety of SoftWar commands and improved performance for larger data sets. We will use a B+ tree that supports insert and search operations as our data dictionary object. We will also be replacing our spatial structure with a PR quadtree. We will also add support for displaying the map as graphical output using the CanvasPlus package found on the class Web page.

7.1 The PR Quadtree

While the treemap with comparator from Part 1 was sufficient to satisfy all queries related to the spatial coordinates of sites whose data had been mapped into the treemap, anecdotal evidence alleges that range searches cannot be done efficiently using such a structure. So, in Part 2, your job will be to implement the spatial database using a PR quadtree, a 4-ary tree organized as a 4-way search trie, where the data values stored in internal nodes are called “guides” that are selected to permit quick access to the leaf level. The only leaf node that will be examined will be the one containing (or that should contain) the target data value, called a ”key”.

Warning: The convention of referring to the data values in internal nodes as guides or search guides, and data values in leaf nodes as keys or search keys will be used for the rest of the course.

Unlike the l-heap and the B+ tree, discussed below, there is agreement upon the definition of a PR quadtree. So, for the time being, visit your local reference book. And check out google using ”PR quadtree” as the search term for a neat honor’s thesis.

7.2 B+ Tree Design Requirements

Like the binomial trees of Part 1, the degree of the root of a member of a B tree family is often used structures in the B tree family based on the degree of the root. A B+ tree of order m is typically implemented as an m-ary tree, where the degree of the root of any subtree is m, and leaf nodes do not have any space reserved for children.

To facilitate exhaustive testing, you will implement a B+ of order order, where the parameter order represents the degree of each node. Your B+ trees must be constructed using the following rules and conventions; no exceptions can or will be made.

- Internal nodes: must always contain between floor((order – 1)/2) and order – 1 (non-null) guides, inclusive, with exactly one more child than the number of guides at all times. This implies between ceiling(order/2) and order non-empty child subtrees per node, inclusive.
• Leaf nodes: must be at least half full—the number of keys this requires is typically an application-specific constant.\(^3\)

**Part 2:** A leaf must always contain between ceiling((order – 1)/2) and order – 1 (non-null) keys, inclusive.

**Part 3:** A leaf must always contain between ceiling((order)/2) and order (non-null) keys, inclusive.

• Remember the root is an exception, in that it never has a lower bound on the number of guides or keys it contains. Furthermore, the root node is defined to be either a leaf or an internal node that has at least two (2) non-empty subtrees.

• Actual search keys (leaf data values) must be unique. However, no such restriction exists within the set of acceptable guide values, and guide values may match keys. So, to accommodate such data value replication, we adopt the convention that a guide (internal node data value) must be less that or equal to the data values (guide or key) in the node to the right of it and below it. There are no other rules regarding the selection of guide values (!).

These rules ARE mandatory, meaning that no credit will be given for the B+ tree if these rules are not observed. Just like the l-heap, there can be multiple B+ trees of a given order that are correct; so, we will exploit your adherence to the above criteria to grade your B+ tree, since grading by diff-ing with a sample output is impossible.

### 7.3 B+ Tree Implementation Requirements

Your B+ tree is required to implement the `SortedMap` interface in the `java.util` package. (In earlier versions of JAVA, we would have had you extend the `Dictionary` class but it has become deprecated and replaced by the `Map` and `SortedMap` interfaces.)

A `Map`, in short, is:

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.\(^4\)

The `SortedMap` extends the `Map` interface to further specify:

A map that further guarantees that it will be in ascending key order, sorted according to the natural ordering of its keys (see the Comparable interface), or by a comparator provided at sorted map creation time.\(^5\)

Since a B+ tree guarantees that the actual values, in this case what we've called "search keys" will be stored in sorted order in the external/leaf nodes, the `SortedMap` interface is the most appropriate choice for allowing us to design our B+ tree so that it can be used in place of any other data structure that implements the `SortedMap` interface.

By requiring that you implement the `SortedMap` interface, we can test your B+ tree separately from your code! That is, we will write our own Mediator object which does something that requires the use of a `SortedMap`. We will plug your B+ tree into this Mediator to test it beyond the scope of the commands for Part 2.

You must adhere to the following rules for your B+ tree:

1. **Names:** Your class should be called "BPTree" and be contained in a file named "BPTree.java"

2. **Constructors:** You must minimally implement the following four constructors:

\(^3\) It’s logical to make this a parameter, specifying the B+ tree you implement using two parameters: the order, \(m\), the number of keys, say \(n\), required in the leaves. However, we won’t test that sort of thing here.

\(^4\) [http://java.sun.com/j2se/1.4.1/docs/api/java/util/Map.html](http://java.sun.com/j2se/1.4.1/docs/api/java/util/Map.html)

\(^5\) [http://java.sun.com/j2se/1.4.1/docs/api/java/util/SortedMap.html](http://java.sun.com/j2se/1.4.1/docs/api/java/util/SortedMap.html)
BPTree() // defaults to order 3, assumes added elements implement the Comparable interface
BPTree(Comparator c) // defaults to order 3, uses a Comparator and never tries to cast an added object to a Comparable
BPTree(int order) throws IllegalArgumentException // initializes the tree to the specified order and assumes added elements implement the Comparable interface; throws IllegalArgumentException if order is less than 3
BPTree(Comparator c, int order) throws IllegalArgumentException // initializes the tree to the specified order and uses a Comparator and never tries to cast an added object to a Comparable; throws IllegalArgumentException if order is less than 3

To make your life easier, we won’t be asking you to actually implement the headMap, tailMap, or subMap functions. If someone calls these functions, you may simply throw an UnsupportedOperationException and move on with life like this:

```java
public SortedMap headMap(Object key)
{
    // you don’t need to specify that this exception
    // is thrown in the function signature since it
    // is an unchecked exception
    throw new UnsupportedOperationException("Not required");
}
```

For Part 2, you may also throw an UnsupportedOperationException for the remove method as this will not be required until Part 3.

For any function in Map/SortedMap that takes a "key" as a parameter make sure that your functions will work if an object of type "String" is passed in as the parameter. For any function in Map/SortedMap that takes a "value" as a parameter make sure that your functions will work if an object of generic type "Object" is passed in as the parameter. This means that you should never try to cast your "value" parameter to type "Site". A good way to make sure that you did not do this would be to try to save objects of type Integer to your B+ tree during testing. (HINT: A random number generator, like Math.random(), would also make generating test data very easy. You could easily generate large sets of input that you will be quickly able to analyze for correct sorted order.)

There are ways to code your B+ tree to accept a "key" parameter of type Object and using dynamic type checking determine the best method to generate a guide value. This is the recommended method for implementation. However, you are minimally required to accept objects of type String for all parameters named "key".

We also promise that when testing your SortedMap interface we will only try to insert homogeneously typed objects. That is, if we have inserted numerous Integer values into a tree, we won’t try to insert a Double or Site into the same tree. Having values of different types in your leaves may or may not be a problem given your design, but the behavior for comparing a Site object to an Integer, for instance, is undefined so we won’t be testing it.

(Note that the Map/SortedMap "key" is the same as the guide value discussed in lectures.)

Your B+ tree is not required to support null keys or values, however you may choose to do so. If a null values are ever passed in for a "key" or "value" parameter you are permitted to throw a NullPointerException if your implementation does not support null values.

You can find the exact specifications for the SortedMap interface in the JAVA API at:

http://java.sun.com/j2se/1.4.1/docs/api/java/util/SortedMap.html

One hint if anyone really likes a class name other than BPTree: you can just add an extra BPTree.java file with the following contents and you’ll be set:

```java
//file BPTree.java
```
public class BPTree extends YOURNAME
{
    public BPTree(){super();}
    public BPTree(Comparator c){super(c);}
    public BPTree(int order){super(order);}
    public BPTree(Comparator c, int order){super(c, order);}
}

7.3.1 Extra Credit: entrySet, keySet, values

For extra credit (5 to 10 points), you may implement the entrySet, keySet, and values method for your BPTree class.

As the Java specifications for entrySet, keySet, and values, note, your BPTree should return a Set or Collection that is backed by the BPTree. When a Set/Collection is "backed" by another data structure the set/collection always reflects the state of the backed data structure. That means that if someone called "BPTree.entrySet" then "BPTree.put", the Set obtained from entrySet would allow you to access the element added from the put operation even though the entrySet was returned prior to the call to put.

This is not as hard as you might think it is. There are many approaches to this problem but may we suggest that you take a peak at the source code for TreeMap. You are allowed to adopt the code from TreeMap for entrySet, keySet, and values as you see fit. As always, document your source.

Be sure that your BPTree.entrySet().iterator(), BPTree.keySet().iterator, and BPTree.values().iterator() calls return correctly working Iterators!

If you decide to not implement entrySet, keySet, or values, you should throw an UnsupportedOperationException.

7.4 Fault Tolerant Design Features

An important consideration when developing life-critical applications is to include design elements that mitigate the effects of run time errors. One way to do this is by using fault-containment regions, where critical components are isolated from other parts of the system to prevent errors in one area from propagating to another, otherwise unfailed portion of the system. Another way is to include sanity checks on process inputs and outputs, where pre-conditions and post-conditions are checked and enforced at runtime.

To this end, Part 2 requires you to store the base-related information of Part 1 in a Friendly site data dictionary, and the target-related information of Part 1 in an Enemy site data dictionary. That way when we are destroying enemy targets, we do not instead accidentally blow up little Billy’s house, much less Dr. Samet’s.

Because Dr. Hugue’s research area is dependability, you will have the unenviable pleasure of implementing fault isolation and transformation strategies that will decrease the probability that user errors can result unexpected and unwanted civilian and non-combatant casualties.

7.5 Where’s Part 2?

By popular demand, you are hereby directed to the frozen version on the website. After all, if there is no bogus stuff in the spec, you can’t accidentally use the earlier specification. Just trying to prevent any more errors on your part or mine.

8 Part 3: B+ tree delete, PM1 Quadtree and Adjacency Lists

In Part 3 we upgrade our data structures and add new commands to utilize these changes. You will be adding new functionality to SoftWar by accepting tracks and locations of interest, or loci in your spatial structure. To do this, the PR Quadtree will be upgraded or replaced (your design decision) with a shiny new PM1 Quadtree that is more appropriate than the PR when modeling graphs that can be interpreted as polygonal maps. The implementation of a PM1 quadtree (instead of, say a PM2 or PM3) will permit
us to quickly identify the closest track to a given coordinate, with search time in $O(\log n)$, instead of the search time given by $O(n)$ of the PM2 and the PM3. Additionally, we will complete the B+ Tree from Part 2 by adding delete capabilities, and give extra credit to those of you who finish implementing the SortedMap interface as well.

You will also maintain adjacency lists corresponding to graphs of loci connected by tracks, which we shall call routes. We use this collective term because collections of tracks connecting different types of loci may have different roles than say, roads. However, road would not be an appropriate term for the set of edges between say, a weapon and its target.

In the Part 3 specification, we will attempt to repair the errors and inconsistencies that have plagued earlier specs. We will be removing punctuation at the end of error messages. This will improve the ease of quick visual checks for output correctness. Furthermore, since the complexity of Part 3 is about double that of Part 2, we are eliminating the separation of data dictionaries to make grading easier.

*Any other changes of this nature will be added here.*

### 8.1 Adjacency List

The polygonal maps in the name of the PM1 quadtree are essentially finite graphs (edges and vertices), which we will call loci (plural of locus) and tracks. They are stored in the PM1 quadtree to support identification of the closest track to a given point in 2-space. Note that this is a major change from Part 2; so, to lessen the impact of the change, we are maintaining the same definition of sites as in Part 2, but treating them as a special case of the more general locus. However, the method of capturing vertex-edge relationships in the PM1 is not appropriate for finding the shortest set of edges between two vertices, or other applications that explore the connections between a vertex and all vertices adjacent to it. So, you will also be implementing adjacency lists to reflect combinations of tracks (edges) that have been inserted into the PM1 quadtree.

There are many ways to implement an adjacency list—just an array of arrays or the simple matrix that you may have learned in the lower level courses! You can do a skiplist of skiplists, or a TreeMap of TreeMaps, or any number of various objects which provide the right traversal capabilities. The principal requirement for this project is that insert and search for the loci connected to a specific locus must be in $O(\log n)$.

Regardless of the implementation of the adjacency list, its purpose is to provide us with a quick and easy way to perform shortest track calculations so we will be able to produce full SoftWar functionality in part 4.

### 8.2 Routes, Adjacency Lists, and the PM1

It is often convenient to maintain representations of distinguished subgraphs because they cannot be constructed on demand in a timely fashion. Furthermore, we may need to distinguish various combination of edges (tracks) between vertices (loci) according to the types of loci and use of tracks.

Because our application involves connecting loci of several LocusTypes, we must have terminology to discriminate between, say, the sequence of tracks to be taken between two friendly loci, passing through only friendly loci; and, the sequence of tracks to be taken between a friendly base firing on an enemy locus that is considered a target. Why? for the same reason that we differentiate between an interstate highway and a fight track. So, we will adopt the term route as a connected graph consisting of two or more distinct loci, and the set of tracks (edges) and loci (vertices) in between. Specifically, we may need to distinguish among various combinations of edges (tracks) between vertices (loci) according to LocusType and EnemyType or FriendType. However, Part 3 will address only a subset of possible route classes including, but not limited to, routes for military, civilian, and refugee movements; and trajectories for targeting purposes. Each class can be further partitioned according to whether routes in the set are made of tracks between FriendlyLoci, or may involve one or more tracks connecting an EnemyLocus to a FriendlyLocus. Part 4 will add routes for managing weapons stores, and for distribution of essential supplies.

### 8.3 B+ Tree Design Requirements

As shown in the roadmap, you will now be required to support deletion of keys (loci) from the data dictionary. Thus, one change from Part 2 to Part 3 is that the remove operation of the SortedMap interface should no longer throw the UnsupportedOperationException. However, you are not required to implement remove
for entrySet, keySet, or values because remove will remain an optional operation, with an extra credit point value, for those of you who choose to do it.

There is one major change to the B+ tree definition in the Part 3 specification:

**Part 2:** A leaf must always contain between ceiling\((order - 1)/2\) and \(order - 1\) (non-null) keys, inclusive.

**Part 3:** A leaf must always contain between ceiling\((order/2)\) and \(order\) (non-null) keys, inclusive.

This will let us give those of you with no Part 2 credit for a working B+ tree a few points for understanding someone else’s code well enough to modify it. It will also demonstrate the benefit of object oriented design in dealing with unexpected changes in the specification. You may refer back to the section 7.2 (in the Part 2 portion) of this specification should you have any other questions regarding the B+ tree implementation.

### 8.4 PM1 Design Requirements

PM1 quadtrees can contain **points** and **edges**. Edges are defined to have two different points as endpoints. Individual (or isolated) points can be viewed as “edges with the same start and end point”, and should be inserted that way. Thus, a PM1 quadtree, in a sense, can be treated as a structure in which only edges are inserted. Feel free to use whichever abstraction makes the more sense to you. However, I will never use the term edge to refer to an isolated points.

The portion of a segment or edge contained in a PM1 quadtree node is called a **q-edge**, while the term **point** can be either an **endpoint** or an **isolated point**, unless otherwise stated.

The edges in a PM1 quadtree may not intersect. Given two lines, it is possible that they don’t mathematically intersect but they may be very very close together, which for our purposes will be considered an intersection. (The reason is because if you get two lines that are too close together, you will continue to partition your PM1 quadtree until your stack overflows.) Therefore, we will set a maximum depth for your PM1 tree of 20 levels. If you discover that you must create more than 20 levels in your PM1 quadtree to store two edges, we will consider them to be an intersection and will reject the new edge.

Additionally, your PM1 quadtree should adhere to the following rules:

1. A black node can contain exactly one (1) isolated point.

2. A black node cannot contain a point and a segment unless that point is an endpoint of the segment or the q-edge.

3. A black node can contain exactly one q-edge, unless the q-edges share an endpoint in the black node.

4. If a point falls on the boundary between more than one node, then it must be added to every node that it intersects (a point may be contained in up to four leaf nodes).

5. If even one point of a q-edge falls on the boundary of more than one node then it must be added to all of those nodes as above. (A single q-edge can of course appear in a lot of nodes).

6. The tree must at all times be minimal— that is, there must never be a smaller tree which follows all of the above rules and still contains all of the same data. This should only be a complicated issue during deletion.

7. The maximum depth of your PM1 quadtree is 20 levels. (Warning: try to count correctly, meaning that the PM1 quadtree of maximal depth has levels numbered from 0 to 19 inclusive.)

Also, by now I hope you’ve been taking advantage of the Java API, especially the java.awt.geom libraries. <cough> Point2D, Line2D, Rectangle2D, Ellipse2D <cough> Ahem. Sorry about that...cold season.

Please note that although our PM1 quadtree is capable of storing all of the points of interest (e.g. sites, loci, etc) and tracks, it does not provide a very simple interface for performing shortest track calculations. As a result, you should still maintain an adjacency list to perform these types of calculations.
8.5 Expanding Beyond a Site Object

In order to increase functionality of SoftWar we want to move beyond just having a collection of Site objects. Instead, we will now say that a Site has a radius.

Thus, we can identify sources of food, shelter, munitions, medical assistance, and other attractions within the radius of a site.

These new objects, which we will generically call Loci, differ from sites in that they are modeled as two-dimensional points (meaning they do not have a radius...or have a radius of zero, depending on how you look at it).

There will be several types of loci that can be placed inside the radius of a site. If you use JAVA correctly you will have to create very little new code in order to implement these new objects.

Please note that all Sites are Locis, but not all Locis are Sites. That is, Site is a type of Locus.

8.5.1 Site Limits

Each Site now has a radius. A Site is assumed to be centered at the specified \langle(latitude, longitude)\rangle coordinates and extends out to the specified radius around this center point. The resulting circle denotes the "site limits" for that site. Any Locus is said to be "in the site" when the distance from the the center of the site to the Locus is less than or equal to the radius of that site (i.e. falls inside the "circle" of the site centered at \langle(latitude, longitude)\rangle with the specified radius, bound inclusive).

Please note that it is possible for two sites to overlap (e.g. like a Venn diagram), although every mapped site will be centered at a different locus.

8.5.2 Encoding Locus Types

In the remainder of the project, we will adopt the the method of encoding the type of the site in its name that was discussed in the newsgroup in glorious detail by Dr. Hugue while everyone else was trying to understand the Part 1 specification. For example, if all my "Civilian sites", Friendly or Enemy, have names beginning with the letter "C", it's really hard to confuse one of them with an actual target name, such as "military Base", that begins with the letter B. While this method makes it easy for my staff to update test data, it also gives the human one less parameter to think about. We have embedded the SiteTypes of Part 2 in the LocusType table shown below. The LocusTypeCode is new for Part 3, and is merely the first character required in the names of all loci of that LocusType.

<table>
<thead>
<tr>
<th>LocusTypeCode &amp; First Character</th>
<th>Encoded LocusType</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Air Unit Site</td>
</tr>
<tr>
<td>B</td>
<td>military Base Site</td>
</tr>
<tr>
<td>C</td>
<td>Civilian territory Site</td>
</tr>
<tr>
<td>E, F</td>
<td>Reserved-illegal locus name</td>
</tr>
<tr>
<td>G</td>
<td>Ground unit Site</td>
</tr>
<tr>
<td>H</td>
<td>Hospital Site</td>
</tr>
<tr>
<td>I</td>
<td>Interesting Places</td>
</tr>
<tr>
<td>J</td>
<td>Junction</td>
</tr>
<tr>
<td>M</td>
<td>MASH Unit</td>
</tr>
<tr>
<td>P</td>
<td>Pantry and Supplies</td>
</tr>
<tr>
<td>S</td>
<td>Shelter</td>
</tr>
<tr>
<td>W</td>
<td>Weapons</td>
</tr>
<tr>
<td>other</td>
<td>Unassigned</td>
</tr>
</tbody>
</table>

8.5.3 Friend and Enemy Data Structures

In Part 2, we used two create commands to access two separate data dictionaries, one for Friendly territory names and the other for Enemy territory names. We are abandoning this feature for Part 3 since its principle use was for the Part 4 functions that we have annulled.
8.6 Part 3 BNF–The Latest and Greatest

The BNF for Part makes use of the properties of dynamic dispatch. That is, your dictionary, spatial data structure, and adjacency list will be storing Site and Locus objects. Generically all Site objects are considered to be Loci. Thus, all of the "Site" related BNF commands have been renamed to "Locus." However, it is important to note that an actual Site object and the generic Locus object have different output types. That is, if you get a <LocusAlreadyExists> error message and the object that already exists is actually of type Site, use the <Site> format for printing the error message. If the actual type of the object is a generic Locus (not a Site), use the <Locus> format.

The same applies to <Track> which may be of type <OneWayTrack> or <TwoWayTrack>.

This should be a fairly easy task in JAVA if you set up your classes properly. (Hint: overload the toString() method).

<SiteType> ::= air unit|military base|civilian territory|ground unit|hospital
<SiteName> ::= <name>

<StartName> ::= <SiteName>
<DestinationName> ::= <SiteName>
<StartLocus> ::= <Locus>
<DestinationLocus> ::= <Locus>
<LocusOne> ::= <Locus>
<LocusTwo> ::= <Locus>

<LocusList> ::= <Locus><nl> <LocusList> | <Locus><nl>

<FindInSiteLocusList> ::= <Locus> at distance: <double><nl> <FindInSiteLocusList> | <Locus> at distance: <double><nl>

<Track> ::= (<StartName><Direction> <DestinationName>)
<Direction> ::= <OneWay> | <TwoWay>
<OneWay> ::= ==
<TwoWay> ::= <>

<TrackGraph> ::= <Track><nl> <TrackGraph> | <Track><nl>
<ListTracks> ::= Follow <Track><nl> <ListTracks> | Follow <Track><nl>
<Route> ::= <ListTracks><nl> Total cost: <double><nl>
<AdjListEmpty> ::= No tracks entered<nl>
<NoLociInRange> ::= No loci names in range <LocusOne> to <LocusTwo><nl>

<BPTreeOrderAlreadyInitialized> ::= Error: B+ tree order already initialized<nl>
<BPTreeOrderNotYetInitialized> ::= Error: B+ tree has yet to be initialized<nl>

<InvalidBPTreeOrder> ::= Error: B+ tree order must be greater than or equal to three (3)<nl>
<InvalidLocusType> ::= Error: Invalid locus type<nl>

<LocusAlreadyExists> ::= Error: The specified locus cannot be created because <Locus> already exists<nl>
<LocusAlreadyMapped> ::= Error: The locus <Locus> has already been mapped at those coordinates<nl>
<LocusDoesNotExist>::Error: <LocusName> does not exist

<LocusNameIsSite>::Error: Cannot perform operation on a Site

<LocusNotMapped>::Error: The locus <Locus> has not been mapped

<LociEntered>::Error: No loci entered

<NoRouteExists>::Error: No route found from <StartLocus> to <DestinationLocus>

<NotSitePosRadius>::Error: Only Sites may have non-zero radii

<SiteDoesNotExist>::Error: <SiteName> does not exist

<SiteNameIsLocus>::Error: Cannot perform operation on a Locus

<SiteWithZeroRadius>::Error: Sites must have non-zero radii

<SpatialIntersection>::Error: Spatial intersection

<StartAndDestinationSame>::Error: Start locus and destination locus must be different

<TrackAlreadyExists>::Error: The specified track cannot be created because <Track> already exists

<TrackDoesNotExist>::Error: No such track

<UnableToDeleteLocus>::Error: <Locus> cannot be deleted because it is mapped

8.7 Required Part 3 Commands: Very Stable

You are required to attempt the following commands. Please check your spelling carefully for Part 3, because we will be extremely stingy when it comes to fixing careless errors, given that you’ve had two parts already to get the details down.

INIT_BPTREE(order) This command takes one parameter: the order of the B+ tree to be created, which is an integer that is greater than or equal to three (3). All B+ trees that are created during a given execution of your code must have this order. If the B+ tree order has not been initialized, this command will initialize the B+ trees to have the specified order and print the <success> message. If the parameter order is less than three (3), print the <error> message <InvalidBTreeOrder> If the B+ tree order has already been initialized, then you should print the <error> message <BTreeOrderAlreadyInitialized>.

If any other command involving the B+ tree is attempted before the B+ tree order has been initialized, then print the <error> message <BTreeOrderNotYetInitialized>. Although this error message is a "user-level" exception thrown by several other commands, we mention it here because consideration of how to deal with it might have an impact on this command as well. And, just in case we forget to include it in the BNF for a given command, this error message should if any other command is encountered before the B+ tree order is initialized.

Output summary:
<output>::=<success>|<error>
<success>::=Initialized B+ trees to order <order>|<nl>
<error>::=<BTreeOrderAlreadyInitialized>|<InvalidBTreeOrder>

EXIT() This is the simplest command. It should end the program. Print mission complete and exit the program. Your program should also naturally terminate (with no exit message) when the end-of-file is reached.

Output summary:
<output>::=mission complete|<nl>

CLEAR_ALL() This command takes no parameters. This command should delete all entries from the dictionary, spatial data structure, adjacency lists, and inventory heap. (Hint: You don’t actually have to perform any delete operations on any of these objects.) The confirmation message structures cleared should be printed.
CREATE_LOCUS(LocusName, longitude, latitude, radius) This command takes four (4) parameters: the name of the locus as a case insensitive string; the longitude (which represents the x coordinate) as an integer; and the latitude (which represents the y coordinate) as an integer; and the radius as a non-negative integer. Unless otherwise stated, the radius of a locus is zero (0).

Both longitude and latitude should be integers in the range [0, 1024]. Your Mediator should parse this information and store it in a Locus object. If the dictionary does not already contain a locus by the same name or same coordinates, you should add the Locus to the dictionary and print the <success> message. If there is a Locus in the dictionary with the same name or coordinates, you should print the <LocusAlreadyExists> error message using the Locus that is currently in the data dictionary, NOT the locus specified in the command parameters. Note that loci should be sorted ascibly by their case insensitive name.

Sites are a distinguished subclass of Loci having a non-zero radius. For the purposes of Part 3, a Loci is a Site iff (if and only if) the radius is positive. And, the radii of all Loci are non-negative (meaning either zero or positive).

Since the type of loci is encoded in the naming conventions, you should check for type mismatches between the LocusName, and the radius and print the corresponding error message. For example, if someone where to try to add a junction with a radius > 0, you should print the <NotSitePosRadius> error. Likewise, if someone tried to add military base with the radius equal to zero, you should print the <SiteWithZeroRadius> error. And, finally, detection of coordinates already present in the specific data dictionary, using, perhaps, the TreeMap used for spatial structures in Part 1. All of these operations should be in O(log n), and we will test this.

This command functions almost identically to CREATE_SITE except it creates a generic Locus rather than a Site.

<table>
<thead>
<tr>
<th>LocusTypeCode &amp; First Character</th>
<th>Encoded Locus Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Air Unit Site</td>
</tr>
<tr>
<td>B</td>
<td>military Base Site</td>
</tr>
<tr>
<td>C</td>
<td>Civilian territory Site</td>
</tr>
<tr>
<td>E, F</td>
<td>Reserved-illegal locus name</td>
</tr>
<tr>
<td>G</td>
<td>Ground unit Site</td>
</tr>
<tr>
<td>H</td>
<td>Hospital Site</td>
</tr>
<tr>
<td>I</td>
<td>Interesting Places</td>
</tr>
<tr>
<td>J</td>
<td>Junction</td>
</tr>
<tr>
<td>M</td>
<td>MASH Unit</td>
</tr>
<tr>
<td>P</td>
<td>Pantry and Supplies</td>
</tr>
<tr>
<td>S</td>
<td>Shelter</td>
</tr>
<tr>
<td>W</td>
<td>Weapons</td>
</tr>
<tr>
<td>other</td>
<td>Unassigned</td>
</tr>
</tbody>
</table>

DELETE_LOCUS(LocusName) This command takes in one parameter: the name of the site or locus as a case insensitive string. Your Mediator should search the dictionary dictname for a locus with
the specified name. If you find a locus with that case insensitive name and the locus has not been
added to the spatial data structure, remove it from the dictionary and print the <success> message.
If the locus is in the dictionary but has already been added to the spatial data structure, then print
the <UnableToDeleteLocus> error message. If the locus cannot be found in the dictionary, print the
<LocusDoesNotExist> error. This should be an O(log n) operation.

Output summary:
<output> := <success> | <error>
<success> := Deleted locus <LocusName><nl>
<error> := <BPTreeOrderNotYetInitialized> | <LocusDoesNotExist> | <UnableToDeleteLocus>

LIST_LOCI() This command takes no parameters. This command should print out a list of all of the loci
in the dictionary in ascibetically sorted order. If the dictionary is empty, print the <NoLociEntered>
message. If there are loci in the dictionary, print them as described below. Note that locus names are
case insensitive.

Please note that the error message has changed for this command.

Output summary:
<output> := <success> | <error>
<success> := <LocusList>
<error> := <BPTreeOrderNotYetInitialized> | <NoLociEntered>

PRINT_BPTREE() The purpose of this command is to print out the structure of the B+ tree used as a
data dictionary by doing a breadth first search. However, if you used links between internal nodes you
can produce the same results as BFS with much cleaner code.

The output is formatted so that each level of the tree is enclosed in braces { }; each node is enclosed in
parentheses; guide values in each internal node are separated by spaces; key values within leaf nodes
are separated spaces. Each level of the tree should appear on its own line and in order. Thus, a tree
of order three (3) with five keys (names of sites) might look like this:

{{BOWIE}}
{{BALTIMORE} (COLLEGE PARK)}
{{ANNAPOLIS} (BALTIMORE) (BOWIE) (COLLEGEPARK HOLY CROSS)}

Note that by convention, the leaf containing the key "COLLEGEPARK" (the key of the object of type
"civilian territory" with site name "COLLEGEPARK") is a right child of the internal node containing
the guide value " COLLEGEPARK". That is, we have a rule for dealing with equal guide and key
values.

Even at the leaves print only the key (the locus name). If the data dictionary is empty, print the
<NoSitesEntered> message. Your tree is not expected to match ours exactly. Your grade will be
based on your tree displaying the properties described below.

For our order m tree in Part 3, the leaves will contain between ceiling(m/2) and m keys, inclusive.
Internal node must have between ceiling(m/2) and m children, inclusive. For each internal node, the
number of guide values printed must be one less than the number of child nodes, meaning that (no
'extra' key on the far left should be printed, even if you used one in your implementation. Your tree,
of course, must also contain the correct data at the leaves!

Output summary:
<output> := <success> | <error>
<success> := <b+rows><nl>
<b+rows> := <b+row><nl><b+rows>|<b+row>

21
<b+row>:{<nodes>}
<nodes>:=<node> <nodes>|<node>
<node>:=(<keys>)
<keys>:=<key> <keys>|<key>
<key>:=<LocusName>
<error>:=<BPTreeOrderNotYetInitialized>|<NoLociEntered>

**INSERT SITE**(*SiteName*): This command takes one parameter, the name of the site to add to the map as an isolated site. If the string *SiteName* does not satisfy the naming conventions for a site, print the *SiteNameNotASite* error. If site *SiteName* is not in the data dictionary, you should print the *SiteDoesNotExist* error. If site *SiteName* does exist, attempt to add it to the spatial map. If an intersection is detected, you should print the *SpatialIntersection* error, otherwise, success.

**output summary:**
<output>:=<success>|<error>
<success>:=Inserted Site *SiteName*<nl>
<error>:=<BPTreeNotYetInitialized>|*SiteNameIsLocus* |<LocusDoesNotExist>|*SpatialIntersection*

**INSERT_TRACK**(*startLocus*, *destinationLocus*, *weight*, *two-way*) This command takes four (4) parameters: the name of the start and destination loci as a case insensitive strings, the weight (cost) of the track connecting these loci as a double, and the type of track to input (1 for a OneWayTrack, 2 for a TwoWayTrack). Consider creating an abstract Track and a concrete OneWayTrack and TwoWayTrack.

You are permitted to assume that when grading, we will enter values greater than or equal to zero for the weights (even though you'd really be silly to assume this for debugging purposes, or in case your user makes a mistake). Similarly, you can assume that the parameter *two-way* will be a valid integer equal to 1 or 2, with the obvious definitions.

*Note also that non-site loci cannot exist in the PM1 quadtree unless it is one vertex of a track.*

If startLocus and destinationLocus are the same, print the *StartAndDestinationSame* error message. If either locus has not been added to the dictionary, print the *LocusDoesNotExist* error message. (The *LocusDoesNotExist* message for startLocus takes precedence over destinationLocus.) If a track connecting the same 2 loci already exists, then you should print the *TrackAlreadyExists* error message. If either the start or destination locus is of type Site, then you should print the *SiteNameIsLocus* error message.

If none of the above errors has been detected, you should attempt to insert a Track object of the appropriate type (one (1) or two (2) way) starting at startLocus and going to destinationLocus in the PM1 quadtree. If you detect an intersection while attempting to insert the Track into the spatial data structure, print the *SpatialIntersection* error message. If there is no intersection (i.e. the Track was successfully inserted into the spatial data structure), add the Track to the track graph, an adjacency list containing all the tracks that have been inserted into the quadtree, and print the *success* message. Please remember to handle OneWayTrack and TwoWayTrack objects properly in the adjacency list.

Note: One way tracks are directed edges going from startLocus to destinationLocus. Two-way tracks are undirected edges connecting startLocus and destinationLocus. It is not possible to have two or more tracks with connecting the same two sites. Thus the startLocus, and destinationLocus are the unique key for the track.

**output summary:**
<output>:=<success>|<error>
<success>:=Inserted track *Track*<nl>
<error>:=<BPTreeOrderNotYetInitialized>|*LocusDoesNotExist* |<StartAndDestinationSame* |*LocusIsSite* |*TrackAlreadyExists* |*SpatialIntersection*

22
PRINT_TRACK_GRAPH(): This command takes no parameters, and will be used to output the adjacency list corresponding to the graph of tracks connecting non-site-loci only. This adjacency list is constructed by adding tracks after the INSERT_TRACK() associated with the track has been successful. If no tracks have been inserted in the adjacency list, then print the <AdjListEmpty> success message. The only other error is the ubiquitous <BTreeOrderNotYetInitialized> error message to be printed should this command be encountered prior to B+ tree initialization.

Output Summary

<output>::=<success>|<error>
<success>::=<TrackGraph>|<AdjListEmpty>
<TrackGraph>::=<Track><nl><TrackGraph>|<Track><nl>
<error>::=<BTreeOrderNotYetInitialized>

REMOVE_TRACK(LocusOne, LocusTwo) Remove track will be an optional command. If you have not implemented this portion the PMI delete, then any attempt to use REMOVE_TRACK in Part 3 should result in the <InvalidCommand> error message. Otherwise, implement the command as described below in Section 8.8.

REMOVE_SITE(SiteName) this is an optional command. If you have not implemented this portion of the PMI delete, then any attempt to use REMOVE_SITE in Part 3 should result in the <InvalidCommand> error message. Otherwise, implement the command as described below in Section 8.8.

FIND_IN_SITE(SiteName, LocusTypeCode) This command takes in two parameters: the name of the site in which you are searching as a case insensitive string, and the LocusTypeCode as the one letter code of the LocusType. It indicates the loci for which your are searching. This command finds all of the loci of the specified type that fall within the limits of the specified site.

If the LocusTypeCode does not fall within the set of valid locus types print the <InvalidLocusType> error message. Search the data dictionary to see if the Site exists. If the site could not be found, print the <LocusDoesNotExist> error message. If the Site is not actually a site, print the <SiteNameIsLocus> error message. Search the spatial structure for the site to ensure that it is mapped. If it is not, print the <LocusNotMapped> error message. If the site is found, you search the spatial data structure for all loci of LocusTypeCode that reside within the site limits, which is the circle determined by the radius and center of the site.

Please note that you should just use simple Euclidean geometry to determine the distance. That is, pretend that the world is flat. ;)

The <FindInSiteLocusList> should be printed in order from the closest point to the farthest point. You should break ties using the asciibetically ordering of the Locus’s name. You should also print the distance from the Site’s coordinates (the circle center) as a double. This should print all mapped loci, including sites, that are within the radius of the given sites. A site Alpha is in the radius of site Beta if and only if the center of Alpha is within the radius of Beta.

Output summary:
<output>::=<success>|<error>
<success>::=<FindInSiteLocusList>
<error>::=<BTreeOrderNotYetInitialized>|<InvalidLocusType>
|<LocusDoesNotExist>|<SiteNameIsLocus>|<LocusNotMapped>

PRINT_MAP() This command takes no parameters. This command prints the spatial data structure (in part 3, a PMI quadtree) in the order (NW, NE, SW, SE). Before going in any direction, you should print out the direction you are about to go. Only one direction should appear on a line. You should also indent 2 spaces for each level of depth in the tree.

---

6 Mr. James Shen’s contributed BNF is gratefully acknowledged.
There are two types of output for a black node: If the black node contains a point and any number of q-edges, print the locus corresponding to that point on the same line as the last direction printed. If the black node contains a single q-edge without its endpoints, print the track corresponding to that q-edge on the same line as the last direction printed.

For instance, a tree containing the Wawa (WAWA, 0, 1000), Lasicks (LASICKS, 1000,0) and two-way track (WAWA, LASICKS, 5) would print as:

```
NW weapons: WAWA at (0,1000)
NE
SW (WAWA <-> LASICKS)
SE unassigned: LASICKS at (1000,0)
```

If Mandalay (MANDALAY, 500,600) were added, the output would look like:

```
NW
NW weapons: WAWA at (0,1000)
NE
SW
SE
NW (WAWA <-> LASICKS)
NE
SW (WAWA <-> LASICKS)
SE
NW (WAWA <-> LASICKS)
NE MASH unit: MANDALAY at (500,600)
SW (WAWA <-> LASICKS)
SE (WAWA <-> LASICKS)
NE
SW (WAWA <-> LASICKS)
SE unassigned: LASICKS at (1000,0)
```

Output summary:
```
<output>:=<success>|<error>
<success>:=<black_node>|<grey_node>|<NoLociEntered>
<grey_node>:=<nl><pm_tree>NE <pm_tree>SE <pm_tree>
<pm_tree>:=<black_node>|<white_node>|<grey_node>
<black_node>:=<Locus><nl>|<Track><nl>
<white_node>:=<nl>
<error>:=<BPTreeOrderNotYetInitialized>
```

**DRAW_MAP()** This command has been annulled, and will disappear prior to the frozen spec.

### 8.8 Optional and Extra Credit Commands (50 pts and 50 pts)

Your Part 3 score will come from the points for the commands listed in section 8, and from whatever functions you choose to implement in this section. Please note that 50 points of the functions in this section are considered “required.” The commands in this section will be explained prior to the time that the spec is frozen, and are included here as a courtesy.

They will be addressed when the first part of the specification is more stable.
8.8.1 Optimized Shortest Path, and 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 $O(\text{Elg}V)$ time. In order to achieve this, you may need to restructure parts of your adjacency list.

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). However, since we had the Fibonacci heap in Part 1, everyone has access to at least one working F-heap program.

Note that 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.

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 $O(V*\log(V+E))$ (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, so I will explain it below, but it suffices that if you implement the algorithm correctly, that is the runtime you will have.

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 priority 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 $O(\text{Vlg}E+E)$, 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.*

8.8.2 The Optional Commands

As before, assistance in fleshing these out will be rewarded.

**REMOVE TRACK** *(LocusOne,LocusTwo)* This command takes two (2) parameters: the names of the loci that are endpoints of the track as case insensitive strings. If either LocusOne or LocusTwo does not exist, print the `<LocusDoesNotExist>` error message (note that the LocusOne error message takes precedence over the LocusTwo error message. If there is a track in the spatial data structure that connects LocusOne and LocusTwo, remove the track from the spatial data structure and from the
adjacency list and print the <success> message. Remember that you will need to ensure that your PM1 quadtree adheres to the minimal node rule after each delete operation. Furthermore, unless the loci associated with the track are also contained in another track, no trace of the loci should appear in the PM1 after the track has been removed.

Output summary:
<output>::= <success>|<error>
<success>::= Removed track ><nl>
<error>::= BPTreeOrderNotYetInitialized>|<LocusDoesNotExist>
|<TrackDoesNotExist>

REMOVE_SITE(SiteName) This command takes one (1) parameter: the name of the site to be removed as a case insensitive strings. If site <SiteName> does not exist, print the <SiteDoesNotExist> error message. If site <SiteName> has not been mapped, print the <SiteNotMapped> error message.
If site <SiteName> has been mapped, remove it from the PM1 quadtree. Remember that you will need to ensure that your PM1 quadtree adheres to the minimal node rule after each delete operation.

Output summary:
<output>::= <success>|<error>
<success>::= Removed site <SiteName><nl>
<error>::= BPTreeOrderNotYetInitialized>|<SiteNameIsLocus>|<SiteDoesNotExist>|<SiteNotMapped>

BEST_ROUTE(startLocus, destinationLocus) This command takes two parameters: the name of the start and destination loci as case insensitive strings. This command should find the best (lowest cost) route, based on the sum of the weights of the tracks between the two loci The route should be printed along with the total weight of the route, using the <success> message format. If either locus is not in the dictionary, print the <LocusDoesNotExist> error message. If either locus is not in the map, print the <LocusNotMapped> error message. As with the INSERT_TRACK command, startLocus error messages take precedence over destinationLocus error messages. If there is no route to connect the two sites, print the <NoRouteExists> error message using the locus objects found from the data dictionary that match the specified locus names. Note that this should be implemented using the Fibonacci heap implementation of a priority queue for Dijkstra’s algorithm.

Output summary:
<output>::= <success>|<error>
<success>::= Route
<error>::= BPTreeOrderNotYetInitialized>|<StartAndDestinationSame>|<LocusDoesNotExist>|<LocusNotMapped>|<NoRouteExists>

SPAN_LOCI(StartName, EndName) TBD. This is a range search in the B+ tree. The names of all Loci with a names in the closed interval determined by the StartName and EndName should be printed, where valid StartName and EndName match the rules for locus names. If the set of names is empty, print the <NoLociInRange> message. If there are no loci in the data dictionary, then we consider this an error and output the <NoLociEntered>

Output summary:
<output>::= <success>|<error>
<success>::= LocusList|<NoLociInRange>
<error>::= BPTreeOrderNotYetInitialized>|<NoLociEntered>

NEAREST_LOCUS(LocusType, longitude, latitude) This command takes three arguments—a LocusType, indicating the type of locus of interest, and coordinates <longitude,latitude>. The command returns the name of the locus of the specified type in the map that is closest to position <longitude,latitude>. If the LocusType is not valid, output the <InvalidLocusType>.
Output summary:
<output>::=<success>|<error>
<success>::=<LocusName>| No Loci of <LocusType> Present
<error>::=<BPTreeOrderNotYetInitialized>|<InvalidLocusType>

NEAREST_TRACK(longitude, latitude): Description to appear. Output should be description of
track as it was inserted—using endpoints and direction.
Volunteers solicited to fill this one out, too. Description and BNF.

Output summary:
<output>::=<success>|<error>
<success>::=<Track> is <double> units away <nl>
<error>::=<BPTreeOrderNotYetInitialized>|<InvalidLocusType>

8.9 Part 3 Point Breakdown: TBD
The Part 3 point breakdown will appear here when the specification is frozen.

- B+ tree insert: 10 points (primary input)
- B+ tree delete: 40 points (secondary inputs)
- PM1 quadtree insert: 65 points (secondary inputs)
- PM1 applications: Find in Site TBD.
- Adjacency List: TBD.

These numbers may be subject to change during the grading process, but only in your favor (i.e. extra
credit, ignored tests, etc.).

9 General Policies

9.1 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 cmsc420 package
folder.

You must include the following with every submission: All necessary source files (*.java etc.) to com-
pile 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.
You are welcome to use a makefile for development (JAVAc doesn’t track dependencies very well) but I
should be able to run your project with the following two commands:
javac Main.java
java Main

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

There is a 100K filesize limit. Please don’t include .class files - I will probably strip them out before testing your projects anyway.

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.

Here is a (c++) makefile that you might use as a hint for how to set up dependencies for make. Note that that’s a TAB before $(CC), and make does care. (You’ll get an ’invalid separator’ error, or something like that if you use spaces).

CC = cxx
FLAGS =
LFLAGS = -lm

proj4: bnnode.o bptree.o cell.o main.o pmedge.o pmpoint.o pmquadtree.o util.o
$(CC) $(LFLAGS) *.o -o proj4

bnnode.o: bnnode.cpp bnnode.h bptdata.h
$(CC) -c $(FLAGS) bnnode.cpp

bptree.o: bptree.cpp bptree.h bptdata.h
$(CC) -c $(FLAGS) bptree.cpp

cell.o: cell.cpp cell.h bptdata.h pmpoint.h pmedge.h celledge.h
$(CC) -c $(FLAGS) cell.cpp

main.o: main.cpp bptree.h cell.h celledge.h pmedge.h pmpoint.h util.h psdraw.h $$(CC) -c $(FLAGS) main.cpp

pmedge.o: pmedge.cpp pmedge.h pmpoint.h util.h
$(CC) -c $(FLAGS) pmedge.cpp

pmpoint.o: pmpoint.cpp pmpoint.h pmedge.h
$(CC) -c $(FLAGS) pmpoint.cpp

pmquadtree.o: pmquadtree.cpp pmquadtree.h pmpoint.h pmedge.h util.h psdraw.h
$(CC) -c $(FLAGS) pmquadtree.cpp

util.o: util.h util.cpp
$(CC) -c $(FLAGS) util.cpp

clean:
rm -f *.o
rm -f proj4
rm -f core
9.2 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 input correct (we hope) output. Your output will have all punctuation, blank lines, and non-newline whitespace stripped before diffing similarly cleaned files.

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

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

Typically, each test file will be worth 10 points, and you will be eligible for either 10 or 0 points depending on whether you pass 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.

9.3 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.

9.4 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 an earlier part of the project with anyone must inform Dr. Hugue and receive approval for code sharing prior to releasing or receiving said code.

- Second, since we recognize that the ability to modify code written by others is an essential skill for a computer scientist, and that no student should be forced to share code, we will make working versions of critical portions of the project available to all students once grading of each part is completed, or even before, when possible.
• Dr. Hugue is the sole arbiter of code use and reuse, and reserves the right to fail any student who does not make a good faith effort on the project. Violators of the policies stated herein will be referred to the Honor Council.

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

9.4.1 Code Sharing Policy

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

Furthermore, any portion of your code that contains any portion of the distributed work should contain identifying information in the comments. That is, note which distributed solution your code is based in the file where it was used. It is a good idea to wrap shared code with comments such as "Start shared code from source XYZ" and "End shared code from source XYZ." You may also use comments such as "Parts of this function/file were based on code from source XYZ." You cannot err by including this information too often.

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

References