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, which build upon one another to produce the desired features of an wartime targeting system. For example, some of the data structures from part 1 will be expanded upon in part 2. 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 their projects. Those students who do not devote any time to design are the one most likely to receive a poor grade.

* 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. Hugé.
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. Good design and testing, not Mountain-Dew-Code-A-Thons the night before the due date, is the key to surviving 420.

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 PMI/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.\footnote{While this design pattern is presented on page 273 of [1], a Google search using "design patterns mediator" is cheaper than buying the book.}

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 Softwar program.

### 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/Comparator (i/s)</td>
<td>Fib Heap (i/delete min/merge)</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>*</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>PM1 Quadtree (i/s/d)</td>
<td>*</td>
<td>O(log n) insert</td>
<td>*</td>
</tr>
</tbody>
</table>

****NOTE: This roadmap is subject to change as the semester progresses.****

### 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.

Similarly, you should make regular backups of your work to ensure that your grade will not suffer from any catastrophic failure of your home system.

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 a part of your design and development process.

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)
- 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.print(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:

```java
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:

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

```java
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. ;)

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 (ie. CREATE_BASE and create_base should both be interpreted as the CREATE_BASE command). Note that now all commands should be parsed to upper case. So create_base(COLLEGE PARK,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(College PARK,5,6)).

Please note that this change in the error message format is necessary to facilitate on-line testing, making it easy to identify correct behavior in the presence of incorrect data.

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:

```
[\s,]*
```

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.

```
<InvalidCommand>:=Error: Invalid Command.
```

---

2After all, in real life, it's the summer intern or student worker who is usually given the task of creating test data.
6 Part 1: TreeMaps, Fibonacci heaps, and a Mediator

In Part 1 we will be getting our feet wet with JAVA (hopefully not for the first time) by implementing a treeMap which will store the sites, which can be bases or targets, sorted by site name (dictionary data structure); another TreeMap which will store these sites based on their geographic location (spatial data structure); a Fibonacci Heap to store base inventory data; and, a Mediator which is capable of parsing and executing some basic commands.

To help facilitate the interactions among these various objects, we would recommend that you design some basic objects which you will use to store the data in. For instance, design a Base class which stores all of the information related to a site (name, longitude, latitude, etc). However, when you design your Fibonacci Heap, and TreeMap, you are discouraged from expecting a Base object. Rather, try accepting any generic object (either "Object" or create a base class or interface which Base derives or implements which the other objects expect). The reason for this is that we may not be storing only Bases in future projects (hint hint). If the idea of accepting a generic object is confusing, see the TA or post to the newsgroup as this is a very useful design which will save you lots of time later on. If all fails, go ahead and have your objects expect a Base and you can worry about changing it in later parts of the project. Note: you will be creating "Bases" and "Targets" so you might want those to inherit from some super class (maybe a Site super class??).

The following commands should be supported for Part 1 of the project.

6.1 BNF

```
<Base>::=Base: <name> at (<longitude>,<latitude>)
<Target>::=Target: <name> at (<longitude>,<latitude>)
<Site>::= <Base> | <Target>
<SiteList>::=<Site><nl><SiteList>|<Site><nl>
<SiteDoesNotExist>::=Error: The specified site does not exist.<nl>
<UnableToDeleteSite>::=Error: The <Site> cannot be deleted because it has been mapped.<nl>
<NoSitesEntered>::=No sites have been entered into the dictionary.<nl>
<SiteNotMapped>::=Error: The <Site> has not been mapped.<nl>
<SiteAlreadyMapped>::=Error: The <Site> has already been mapped.<nl>
<SiteAlreadyExists>::=Error: The specified <Site>, already exists.<nl>
```

6.2 Commands

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

```
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, and inventory heap. (Hint: You don’t actually have to perform any delete operations on any of these objects.) A confirmation message should be printed.

```
Output summary:
<output>::=sites cleared.<nl>
```

**CREATE_BASE(name, longitude, latitude)** This command takes three parameters: the name of the Base as a case insensitive string, the longitude as an integer, and the latitude as an integer. Both latitude and longitude will be integers in the range [0, 1024]. Your Mediator should parse this information and store it in a base object. If the dictionary does not already contain a base/target (site) by the same name, you should add the base to the dictionary and print the <success> message. If there is a
site in the dictionary with the same name, you should print the <SiteAlreadyExists> error message. Note that the sites should be sorted asciibetically by their case insensitive name. This should be an O(log n) operation.

Output summary:
<output>:= <success>|<error>
<success>:= created base <Site>\n<error>:=<SiteAlreadyExists>

CREATE_TARGET(name, longitude, latitude) This command takes three parameters: the name of the Target as a case insensitive string, the longitude as an integer, and the latitude as an integer. Both latitude and longitude should be integers in the range [0, 1024]. Your Mediator should parse this information and store it in a Target object. If the dictionary does not already contain a base/target (site) by the same name, you should add the Target to the dictionary and print the <success> message. If there is a site in the dictionary with the same name, you should print the <SiteAlreadyExists> error message. Note that the sites should be sorted asciibetically by their case insensitive name. This should be an O(log n) operation.

Output summary:
<output>:=<success>|<error>
<success>:= created base <Site>.<nl>
<error>:=<SiteAlreadyExists>

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

Output summary:
<output>:=<success>|<error>
<success>:= deleted site <Site>\n<error>:=<SiteDoesNotExist>|<UnableToDeleteSite>

LIST_SITES() This command takes no parameters. This command should print out a list of all of the Site in the dictionary in asciibetically sorted order. If the dictionary is empty, print the <NoSitesEntered> message. If there are Sites in the dictionary, print them as described below. Note that Site names are case insensitive so if the name of one Site was "baltimore" and another Site was named "Reno" then "baltimore" would be printed before "Reno" (since "b" comes before "R" in the ASCII character set when case is ignored).

Note: This Command takes the place of PRINT_DICTIONARY().

Output summary:
<output>:=<SiteList>|<NoSitesEntered>
<SiteList>:=<Site>.<nl><SiteList>|<Site><nl>
<NoSitesEntered>:=No Sites have been entered into the dictionary.<nl>

MAP_SITE(name) This command takes one parameter: the name of the Site to map as a case sensitive string. You should search the dictionary for the Site. If you find the Site in the dictionary and it has not already been mapped, map it by adding it to the spatial data structure and print the
<success> message. If the Site was found but was already mapped, print the <SiteAlreadyMapped> error message. If the Site could not be found, print the <SiteDoesNotExist> error message. Recall that both longitude and latitude are integers in the range [0, 1024).

Note: Takes the place of MAP_DATA().

Output summary:
<output>::=<success>|<error>
<success>::=Mapped site <Site>,<nl>
<error>::=<SiteDoesNotExist>|<SiteAlreadyMapped>

PRINT_MAP() This command takes no parameters. This command prints the spatial data structure.

Output summary:
<output>::=<SiteList>|<NoSitesEntered>
<SiteList>::=<Site>|<SiteList>|<Site>|<nl>
<NoSitesEntered>::=no spatial data.<nl>

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 f-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.
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.  

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.  

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:

   - BPTree()  // defaults to order 3, assumes added elements implement the `Comparable` interface

   

---

3. [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)

4. [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(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
```

14
{  
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.

7.4.1 Encoding Site 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. While this method makes it easy for my staff to update test data, it also gives the human one less parameter to think about.

For example, if all my "Civilian sites", Friendly or Enemy, have names beginning (or ending) 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.
First Character | Encoded SiteType  
---|---  
A | Air Unit  
B | military Base  
C | Civilian territory  
E, F | Reserved-illegal site names  
G | Ground unit  
H | Hospital  
other | Unassigned  

For this project, the SiteType, like site names, is also case insensitive. However, we retain the notation above to help you visualize the correct behavior, and decrease the probability of incorrect assignment of site names to types.

### 7.4.2 Friend and Enemy Data Structures

One simplification in Part 1 was to assume that Friendly and Enemy sites were stored in the same data dictionary. That is, we actually had two separate commands to create a record in the system database. Furthermore, we allowed you to assume that the names of "bases" (now Friendly sites) and "targets" (now Enemy sites) were unique to prevent you from having to implement bucketing algorithms to accommodate duplicate keys.

In Part 2, you will modify the create commands to access two separate data dictionaries. First and foremost, since it is not highly likely that the inhabitants of the Friendly and Enemy territories will agree upon a global site-naming scheme, a simple solution is to implement the Friendly and Enemy site databases using separate data dictionaries.

Furthermore, this use of redundant databases also creates (and enforces) fault containment regions relative to an error in entering a site name. That is, it decreases the probability that a client (wetware, cyberware, or ETware) process, seeking the coordinates of Enemy site, will receive the coordinates of a Friendly base instead.

For example, suppose the database contains keys Babylon and Babylon5, where Babylon is a Friendly military Base in the middle east, and an Babylon5 is an Enemy Air unit. If we were using the common database of Part 1, an omission of "5" in the target site name, for whatever reason, would result in the coordinates for a Friendly military base being provided to client process. But, by implementing separate Friendly and Enemy data dictionaries, we remove this potential failure from the system.

To further protect any inadvertent crossing of the line between Friendly and Enemy data, we also prohibit the use of the first letters F and E for site names. This prevents the human operator from becoming desensitized to the meanings associated with either initial. Granted, this one may be overkill; however, in a world trained to stop for red, just think of how much trouble you could cause by choosing red in you display to mean it’s safe to go on.

### 7.5 BNF Conventions

The following BNF is used to describe the expected output for commands in Part 2.

```
<SiteType>::=military base|hospital|ground unit|air unit|civilian territory|unmapped|illegal  
<Site>::=<SiteType>: <name> at ( <longitude>, <latitude> )  
<SiteList>::=<Site> | <SiteList>  
<SiteDoesNotExist>::=Error: The specified site does not exist.<nl>  
<UnableToDeleteSite>::=Error: The Site cannot be deleted because it has been mapped.<nl>  
<NoSitesEntered>::=No sites have been entered into the dictionary.<nl>  
<SiteNotMapped>::=Error: The <Site> has not been mapped.<nl>  
<SiteAlreadyMapped>::=Error: The <Site> has already been mapped.<nl>  
<SiteAlreadyExists>::=Error: The specified <Site>, already exists.<nl>  
<EmptyInventory>::=Error: Inventory is empty.<nl>  
<HeapEmpty>::=Error: Heap is empty.<nl>  
<BPTreesAlreadyInitialized>::=Error: B+ tree already initialized.<nl>
```
InvalidBPTreeOrder: Error: B+ tree order must be greater than or equal to three (3).

NOTE: The <double> should be formatted to always have three (3) decimal places. This can be accomplished easily using the NumberFormat class in JAVA. Integers and doubles should NOT contain commas.

7.6 Commands

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 B+ tree order has already been initialized, then you should print the <error> message.

Noted for part 2 specifically:

No private data set will attempt to access the B+ tree prior to the B+ tree being initialized. However, this restriction will be removed for part 3 and part 4. This way, those of you who did it the hard way won't be penalized.

However, even if you don’t implement the B+ tree, and must submit using the treemap from Part 1, you must still implement this function, printing the appropriate <success> message. Because this is always the first command and diff is used in grading, if you skip this function your project will fail every test!

Output summary:
<output> ::= <success> | <error>
<success> ::= Initialized B+ trees to order <order>..<nl>
<error> ::= <BPTreesAlreadyInitialized> | <InvalidBPTreeOrder>

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

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, and inventory heap. (Hint: You don’t actually have to perform any delete operations on any of these objects.) A confirmation message should be printed.

Output summary:
<output> ::= sites cleared<nl>

CREATE_FRIENDLY_SITE(name, longitude, latitude) This command takes three parameters: the name of the site as a case insensitive SiteType encoded string; the longitude as an integer; and the latitude as an integer. Both longitude and latitude should be integers in the range [0, 1024]. Your Mediator should parse this information and store it in a friendly object in your Friendly Object Data Dictionary. If the dictionary does not already contain a friendly object by the same name, you should add the base to the dictionary, and then print the <success> message. If there is a object in the dictionary with the same name, you should print the <SiteAlreadyExists> error message. Duplicate data can exist between the separate data dictionaries. For example, you can have a "Gerogia" base in your friendly data dictionary and a "Georgia" base in your enemy data dictionary. Note that the sorted object order is ascibetically by their case insensitive name. This should be an O(log n) operation if you have implemented your data dictionary correctly.
Output summary:
<output>:= <success>|<error>
<success>:=created friendly site <Site>.
<error>:=<SiteAlreadyExists>

CREATE_ENEMY_SITE(name, longitude, latitude) This command takes three parameters: the name of the site as a case insensitive SiteType encoded string; the longitude as an integer; and the latitude as an integer. Both longitude and latitude should be integers in the range [0, 1024]. Your Mediator should parse this information and store it in an enemy object in your Enemy Object Data Dictionary. If the dictionary does not already contain an enemy object by the same name, you should add the base to the dictionary then print the <success> message. If there is a object in the dictionary with the same name, you should print the <SiteAlreadyExists> error message. Duplicate data can exist between the separate data dictionaries. For example, you can have an "Alexandria" Air unit in both your Friendly and Enemy data dictionaries. Note that the sorted object order is alphabetically by their case insensitive name. This should be an O(log n) operation if you have implemented your data dictionary correctly.

Output summary:
<output>:= <success>|<error>
<success>:=created enemy site <Site>.
<error>:=<SiteAlreadyExists>

DELETE_SITE(name) The delete site command will not be supported in Part 2. Any attempt to use DELETE_SITE or DELETE_DATA in Part 2 should result in the <InvalidCommand> error message.

LIST_SITES(dictionaryType) This command takes one parameter: a dictionaryType, "F" for Friendly, "E" for Enemy, that denotes the data dictionary to be printed. This command should print out a list of all of the sites in the data dictionary, indicated by the site type, in alphabetically sorted order. If the dictionary is empty, print the <NoSitesEntered> message. If there are sites in the dictionary, print them as described below. Note that site names are case insensitive so if the name of one site was "baltimore" and another site was named "Reno" then "Reno" would be printed before "baltimore" (since "R" comes before "b" in the ASCII character set).

Dictionary Types:
F = Friendly
E = Enemy

Output summary:
<output>:=<SiteList> |<NoSitesEntered>
<SiteList>:=<Site><nl><SiteList> |<Site><nl>
<NoSitesEntered>:=No sites have been entered into the dictionary.

PRINT_BPTREE(type) The purpose of this command is to print out the structure of the B+ tree using 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 parameter type determines which data dictionary is to be printed, where "F" indicates Friendly and "E" indicates Enemy.

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 commas; key values within leaf nodes are separated commas. 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) (COLLEGE PARK, HOLY CROSS)}

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

Even at the leaves print only the key (the site 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 above in the INIT_BPTREE command.

For our order m tree: The leaves contain between ceiling(m/2) and m-1 keys, inclusive. They may not have m keys. 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 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><n1>
<b+rows>::=<b+row><n1>|<b+row>
<b+row>::=<nodes>
<nodes>::=<node>,<nodes>|<node>
<node>::=<keys>
<keys>::=<key>,<keys>|<key>
<key>::=<SiteName>
<error>::=<NoSitesEntered>

MAP SITE(name, dictionaryType) This command takes two parameters: the name of the site to map as a case insensitive string and the dictionaryType( Friendly or Enemy) as character. You should search the dictionary for the site. If the dictionaryType equals "F" then you are searching for a Friendly site, and "E" if you are searching for an Enemy site. If the site exists in the dictionary and there is no site mapped to the same coordinates, map it by adding it to the spatial data structure and print the <success> message. Note, there is only one spatial data structure for both friendly and enemy sites. If the site could not be found, print the <SiteDoesNotExist> error message. If the site was found but a site was already mapped to the same coordinates (either the same site or a site with a different name but the same coordinates), print the <SiteAlreadyMapped> error message using the site which is already mapped in the spatial data structure. Please note that this may or may not be the same site that was specified as a parameter. For instance, if the site "Buda at (600,600)" is already in the map and we attempted to map "Pest at (600,600)" then we would get the error message "Error: The site Buda at (600,600) has already been mapped at those coordinates."

Recall that both longitude and latitude are integers in the range [0, 1024). This should be an O(log n) operation.

Dictionary Types:
F = Friendly
E = Enemy

Output summary:
<output>::=<success>|<error>
<success>::=Mapped site <Site>,<nl>
<error>::=<SiteDoesNotExist>|<SiteAlreadyMapped>
UNMAP_SITE(name, dictionaryType) Note: Replaces CLEAR_COORDINATE(longitude,latitude) from Part 1. This command takes two parameters: the name of a site as a case insensitive string and the dictionaryType as an character 'F' or 'E'. You should search the data dictionary for the site. If it is not found, print the <SiteDoesNotExist> error message. If the site does exist, you should search the spatial data structure for a site with the specified name. If the site cannot be found in the spatial data structure because it was not mapped, print the <SiteNotMapped> error message. If the site has been mapped, then delete the site from the map/spatial data structure and print the <success> message. This should be an O(log n) operation.

Dictionary Types:
F - Friendly
E - Enemy

Output summary:
<output>::=<success>|<error>
<success>::=Unmapped site <site>.<nl>
<error>::=<SiteDoesNotExist>|<SiteNotMapped>

PRINT_MAP() Note, this command takes the place of PRINT_SPATIAL This command takes no parameters. This command prints the spatial data structure (in Part 2, a PR 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 two (2) spaces for each level of depth in the tree. The site name itself should be printed on the same line as the last direction printed. For instance, a tree containing (Baltimore,0,1000), and (HolyCross,1000,0) would print as:

NW
NE
   NW military base: BALTIMORE at (0,1000)
   NE
   SW
   SE hospital: HOLYCROSS is at (1000,0)
   SW
   SE

If the military base (Bowie,500,600) were added, the output would look like:

NW
NE
   NW military base: BALTIMORE at (0,1000)
   NE
   SW
   SE hospital: HOLYCROSS is at (1000,0)
   SW
   SE military base: BOWIE at (500,600)

<output>::=<success>|<error>
<success>::=<black_node><nl>|<nl><grey_node>
<grey_node>::= NW <prtree> NE <prtree> SW <prtree> SE <prtree>
<prtree>::=<black_node><nl>|<white_node><nl>|<nl><grey_node>
<black_node>::=<Site><nl>
<white_node>::=
<error>::= No matching sites found.<nl>
**DRAW_MAP()** This command takes no parameters. This command will draw a graphical map by examining the quadtree and drawing the partitions, the points in the quadtree, and all of the roads that are in the adjacency list and print the <success> message. Although you are welcome to make use of any drawing mechanisms in JAVA, we recommend that you use the CanvasPlus drawing package on the class web page which was written specifically to support this feature.

Please use common sense in rendering your picture to a size that fits on an average size screen (1024x768 max). It is understood the small quadrants will be difficult to see in the picture, this is ok. We will only be eye-balling your drawing during grading; so, as long as you have a picture that represents the mapped cities and the roads somewhat correctly we’ll be happy.

Note that this function is primarily to help *you* debug your code. It will be very helpful in Part 2 to help you determine if your UNMAP_SITE function is working correctly and it will become an invaluable debugging tool for your PMI quadtree in Parts 3 and 4.

**Output summary:**
output := <success>
success := Drawing complete.<nl>

### 8 Preview: Part 3 and Part 4 Structures and Commands

As discussed earlier in Section 3, you will implement B+ tree deletion in Part 3. In Part 4, we expect you to implement REDUCE_KEY for the f-heap, then use the Fibonacci heap to implement Dijkstra’s algorithm. The information in this section is provided as a courtesy, and is exempt from the freezing rules for Part 2.

#### 8.1 Part 1 Command Modifications

The following commands from Part 1 will not be tested in Part 2, but are expected to appear in Part 3 and Part 4 with a high probability. So, we are including the modifications needed to update them for later use, in case you choose to develop them early for other reasons.

**TRANSFER_ITEM(name1, name2)** name1: case-sensitive string representation of a friendly base name2: case-sensitive string representation of a friendly base This transfers one item from name1 to name2. So, the min element of the Fibonacci heap (of name1) will be removed and then inserted to the inventory of the second argument. Note that there is NO dictionary type; this is because you can ONLY transfer between friendly bases. The children of the min element will be brought up one level in the exact position their parent was located. There will be a consolidation stage performed. Also, you will need to update the min element with the next smallest element in the root list. An error message will be printed if either of the bases do not exist or if either base in not mapped. Also, if the first base has an empty inventory, print the final message. Should be log(n) amortized runtime.

**Output summary:**
output := <success>|<error>
success := item transfer complete
<error> := <SiteDoesNotExist>|<SiteNotMapped>|<EmptyInventory>

**TRANSFER_ALL(name1, name2)** name1: case-sensitive string representation of a friendly base name2: case-sensitive string representation of a friendly base Put the heap of name1 into the heap of name2. See examples of Union. The first argument’s heap should be empty if this operation is successful. You will not consolidate the structure to have at most one binomial heap of a given degree. This should be done in constant time. Note that there is NO dictionary type; this is because you can ONLY transfer between friendly bases.
Output summary:
<output>:= <success>|<error>
<success>:=everything transferred
<error>:=<SiteDoesNotExist>|<SiteNotMapped>|<EmptyInventory>

TRANSFER_ALL_WITH_CONsolidate(name1, name2) name1:case-sensitive string representation of a friendly base name2:case-sensitive string representation of a friendly base Put the heap of name1 into the heap of name2. See examples of Merge. The first argument’s heap should be empty if this operation is successful. You will consolidate the structure to have at most one binomial heap of a given degree. Note that there is NO dictionary type; this is because you can ONLY transfer between friendly bases.

Output summary:
<output>:= <success>|<error>
<success>:=everything transferred
<error>:=<SiteDoesNotExist>|<SiteNotMapped>|<EmptyInventory>

CONSOLIDATE(name1) name1:case-sensitive string representation of a friendly base Explicit call to consolidate the heaps of the mapped base with specified name. You must maintain the heap property for each binomial heap of a given degree. The first error will be displayed if the base is not in the data dictionary. If the base is not on the map currently, print the second error message. If the heap is empty, you will print the final error message. You will consolidate the structure to have at most one binomial heap of a given degree. Note that there is NO dictionary type; this is because you can ONLY transfer between friendly bases.

Output summary:
<output>:= <success>|<error>
<success>:=consolidated heap
<error>:=<SiteDoesNotExist>|<SiteNotMapped>|<EmptyInventory>

PRINT_HEAP(name) name:case-sensitive string representation of a friendly base You will print each key in the heap in the following manner: enclosed in parentheses, print each node’s value in the root list. If that node has any children, print them immediately next to that node in parentheses. This process will be applied recursively. Therefore, on one line you will be displaying the entire heap. Color is used in the example to make the nesting more apparent. We will be testing each individual binomial heap for the heap property. Also, we will verify that there are at most one heap of each degree. Note: this display is followed by a new line (\n).

The heap is to be printed in the following manner: a) output left parenthesis b) beginning with the min value in the f-heap, for each binomial heap in the forest, output the root value output the roots of its child subtrees enclosed in parentheses repeat recursively until all subtrees have been printed c) output right parenthesis) output a new line (\n)

This is a standard linear representation of a forest of general trees that can be used to serialize such structures. It will allow us to verify that your fibonacci heap satisfies the constraints for such structures, regardless of the specific ordering of values in the f-heap.

SAMPLE FIBONACCI HEAP

```
    7-----------52-----------18
    |                   |
    |                   |
24------17------30   |
    |                   |
    |                   |
                  41--39
```
You must have at least one value within each set of parentheses. This will be called on the MIN element. This means printing will begin with the min element.

Output summary:
<output> := <success> | <error>
<success> := <PrintRow>
(PrintEachNode )
(PrintEachNode ) := <NodeDisplay> <PrintEachNode > | <NodeDisplay>
(NodeDisplay ) := <value of node > <PrintRow > | <value of node>
<error> := <HeapEmpty>

9 Deprecated Commands

Because we keep the frozen version of the text from each part intact through later releases of the specification, we include a list of the commands that will never be tested again. Or rather, commands which have an extremely low probability of ending up in any of the remaining project part.

9.1 Deprecated Part 1 Commands

We have no intention of testing these commands ever again.

• CREATE_BASE
• CREATE_TARGET
• DELETE_DATA

The following Part 1 commands have been retired for their own good and ours. *Garey... will fill this in for the next release of the specification.*

10 General Policies

10.1 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 ;):

```
~fhm20001/Bin/submit # part#.tar.gz
```
In all cases ’#’ represents the number of the project part you are submitting(1,2,3 or 4). The filename is not really important; It is important that the file is in .tar.gz format and that your Main.java and other required files are not in a subfolder of the tar file. Subfolders are allowed, such as the cmisc420 package folder.

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

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

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

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.

Every early submission will overwrite the previous early submission, every on-time submission will overwrite any previous on-time submissions, every 1day late submission will overwrite any previous 1day late submission and so on. So I will have one submission for every valid submission period. (Late policy TBA).

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

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

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

cell.o: cell.cpp cell.h bpdata.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
10.2 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 created correct (we hope) output. Your output will have all punctuation, blank lines, and non-newline whitespace stripped before diffling similarly cleaned files.

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

Some text output cannot always be graded by simply diffling 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. I may give points projects that fail a test because of ‘small’ errors after initial grading at my own discretion. 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 so that some ‘get lost’, you may still end up failing other tests since they all require a working dictionary.

10.3 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. Hue and do not put it off! While some may enjoy failing students, Dr. Hue 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 project. A submission that gets only 20 or 30 points is considerably better for you than no submission at all.

10.4 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.

10.4.1 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