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. The first two parts are pre-war design, where your goal is to replicate the conflict as closely as possible. 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.

Roadmap

For each part, you must implement each part using the specified data structures below. Do not worry about the Adjacency List until the 4th part. This will store roads between blacknodes in your PM1Quadtree. It will be used to aid shortest path calculations.

Key:
i-insert
s-search
d-delete

<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</td>
<td>Fibonacci Heap (i/delete min/merge)</td>
<td>None</td>
<td>Any implementation</td>
</tr>
<tr>
<td>2</td>
<td>B+ tree (i/s)</td>
<td>PR Quadtree (i/s)</td>
<td>*</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>B+ tree (i/s/d)</td>
<td>PM1 Quadtree (i/s)</td>
<td>*</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>B+ (i/s/d)</td>
<td>PM1 Quadtree (i/s/d)</td>
<td>*</td>
<td>Any implementation</td>
<td>*</td>
</tr>
</tbody>
</table>

This may change through the semester.

Short Descriptions:

**Dictionary:** a single key discriminating data structure. In the first part, it will store only bases and targets. Later it must store all types of friendly/enemy objects, as well as any other comparable object.

**Spatial:** a multi-key (longitude and latitude) discriminating trie. This will be the backbone of the battlefield map. It will be used to query and organize mapped objects such as roads, mobile objects and stationary locations.
Mediator: this will be used to parse the input commands. You must adapt it throughout the semester to account for new or modified commands. It will serve as an intermediary between the user commands and data structures with utilize those commands.

Part Overviews

In Part 1 you will simply be starting your task. You will utilize the TreeMap class which implements the SortedMap interface (among other interfaces). Using this implementation of SortedMap, you will gain valuable experience with Java interfaces as well as Comparators. Both your data dictionary and spatial representation functions will be implemented using TreeMap. Your data dictionary will store base objects and target objects. Thus, to represent these two objects, you may either create separate objects or use a simple boolean to differentiate the two, since both will be replaced in the future. However, utilizing different objects would be good practice for the B+ tree to come. Each one of your bases will have a Fibonacci Heap within, which contains its weapon inventory. This will be a min heap that will only need to support adding a single object, merging Fibonacci Heaps, and removing the min element. This design is analogous to sharing weapons between bases. Note: the government (ARMORY) is the only place that may receive weapons that were not traded from some other base. ARMORY will not appear in any of your data structures, but will be more a global variable that can never be deleted. The government will then supply each base as necessary with supporting commands. This structure will be in at least the next 2 part, so it is important that you implement it correctly. I will have further specifications for pointer arrangement on each node. Finally, you will write a command parser in any manner you decide. This will never have any restrictions placed on it.

Part 2 will replace your TreeMap with a B+ tree. Also, bases and targets will be extended to friendly/enemy objects. Some of these objects will have added functionality. For this part, you may treat all objects equally except civilian cities, which may not hold any weapons. You will also be adding a spatial data structures to mirror the objects that are contained in the data dictionary. Thus, whenever you create a friendly/enemy object, it will also be mirrored onto the PR Quadtree.

Part 3 will allow the data archive, currently the B+ tree, to support delete. Roads will be added between bases, civilian cities, and hospitals. Also, the functionality of the objects will become more apparent. For example, an enemy target may not be within range so inventories may be transferred to mobile units. Ground Units may only travel on roads. Air Units may move anywhere on the map. There are special rules for certain mobile objects that should be observed before you begin implementation. Bases, Hospitals, and Civilian Cities are stationary. Because of the enhancements to mobile units, you will now be able to mobilize utilities to aid in launching weapons. The PM1Quadtree will be introduced with only insert and search capabilities. Delete will not be necessary yet. Roads may also be added between non-mobile objects on the map. Your PM1Quadtree should store all roads that are mapped. An adjacency list is optional, and not necessary in this part. NOW, when you create an object, it is inserted into the data dictionary. only. A separate command will add objects to the spatial data structure.

Part 4 will allow objects to be destroyed based on the power of weapons in relation to the defense of an object. The way mobile objects move will also be altered. Shortest Path calculations will be introduced. Finally, you will be able to overtake weakened, but not destroyed, enemy objects.
Command Parsing

In order to grade projects we will establish a set of rules to follow for input and output. Firstly, ignore blank lines completely. For all non-empty lines of input, you should parse all commands regardless of case or internal spacing. The spacing before, after, and between parameters and commas is arbitrary. Note: there may not be spacing between the COMMAND_NAME and the first left parenthesis. You must verify that there is at most one command per line, that the parenthesis close, and that the total number and type of arguments is correct. Java has an excellent support for regular expressions.

COMMAND_NAME( ARG1 , ARG2 ) -GOOD
COMMAND_NAME ( ARG1 , ARG2 ) -BAD

Every time you read a line of input, you will echo it exactly with "===>" inserted before each command. If ExIt( ) were the input, the following should be output:
===>ExIt( )
mission complete

More Definitions:
A String is defined as:[_A-Za-z0-9]+  
An Integer is defined as:[0-9]+  
A Double is defined as:[0-9]+(+[.][0-9]*)?  

Note: there are commands that further restrict the input, for example an Integer[0,1024]. In this case, you will treat these inputs as invalid commands if there is not a corresponding error message.

Invalid Commands:
For all invalid commands, simply output the words "invalid command" on a new line after the echo. Example:
EXIT())
invalid command

Submission

First, generate a .tar file containing all files needed to make your project. You should be in a directory with all necessary files to compile your code.

tar -cvf part#.tar *

The '#' represents the part number you are submitting (i.e. either 1,2,3, or 4).

Next, gzip your tarball.
gzip part#.tar

Finally, submit the entire project via the following command.
~mh420001/Bin/submit # part#.tar.gz

You must include a README file in your tarball. It must be named exactly "README". Please include your name, account username, and SID.

Your project will be run using the following commands.
javac Main.java
df

We will redirect input and output. Note: there is a 100K filesize limit. We will be using java 1.4.1.

HIGHEST grade will be taken of all submissions. The late policy is to be announced and/or posted

http://www.wam.umd.edu/~rsgerard/general.html
on the newsgroup.

Freezing

Approximately one week before a project due date, we will freeze the specifications. This means that we can not change anything (BNF/descriptions) except the due date. This should benefit the students by not forcing them to constantly alter or redesign their code to conform to a variable spec. However, this encourages you to carefully find inconsistencies or unclear descriptions as soon as possible. After the spec is frozen, you must implement exactly what is described. This may include misspellings, poor syntax, or anything else. Usually, an updated spec will be produced once a week. You should carefully examine it for changes and updated sections. Each update may include new links as well.

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.

Code Sharing

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.

_Ryan Gerard_
_Last 2 sections by Kevin Conroy_