General Description

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. This structure will be in at least the next 2 parts, 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.

Note: Due to some trivial problems last semester all output lines will be followed by a carriage return. In commands with multiple lines of output, I made it more explicit. On ALL error messages, there should not be any capital letters or punctuation. Regard the clarifications in the next section.

Major Updates (not inclusive):
Within your data dictionary, you are only storing unique names. For example, if you have a base called "OMEGA" then you will NOT have a target called "OMEGA". This is more of a reminder than an update.
New Heap Property: Values stored in the heap need not be unique. You must maintain this property: Consider node P. The value stored in P must be less than OR EQUAL TO the values stored in the children of P.
Your Print Heap function has changed.
Draw_Map/GLOBAL_INSERT has been modified.
Negative Numbers will be detected by your parser.
Heap functions have been broken down further to assist in grading.
Armory is unsupported.

Part 1 Commands

EXIT()
The program will terminate naturally.

Output summary:
<output>:=mission complete

CLEAR_ALL()
A simple command that restores all structures to initial state. Therefore, all data in the data dictionary, inventory heap, and spatial data structure (if applicable) will be removed.

Output summary:
<output>:=structures cleared
CREATE_BASE(name, longitude, latitude)
name: case-sensitive string representation of a base
longitude: integer [0,1024], x coordinate of a base
latitude: integer [0,1024], y coordinate of a base
You will add the corresponding object to your dictionary data structure. If a base/target with the same name already exists, you will print the corresponding error message. Use name as the key for your dictionary. You are only archiving the data and not inserting it into the map (spatial data structure). This must be an \( O(\lg(n)) \) operation.

Output summary:
<output> := <success> | <error>
<success> := created base
<error> := already exists | invalid name

CREATE_TARGET(name, longitude, latitude)
name: case-sensitive string representation of a target
longitude: integer [0,1024], x coordinate of a target
latitude: integer [0,1024], y coordinate of a target
A target object with name, longitude, and latitude will be added to the dictionary data structure. Print an error message if there is already a base or target with the same name already added to the dictionary. Add it to the data dictionary but not the spatial data structure. This must be an \( O(\lg(n)) \) operation.

Output summary:
<output> := <success> | <error>
<success> := created target
<error> := already exists | invalid name

DELETE_DATA(name)
name: case-sensitive string representation of a base/target
You will remove the base/target with the name from your dictionary data structure. If it does not exist, print the corresponding error message. Also, if it is currently on the map, you may not delete the base/target. Therefore, print the corresponding error message. This must be an \( O(\lg(n)) \) operation.

Output summary:
<output> := <success> | <error>
<success> := deleted data
<error> := does not exist | currently mapped

MAP_DATA(name)
name: case-sensitive string representation of a base/target
You will find the base/target with name specified in your data dictionary and add it to your spatial data structure. If it does not exist, print the corresponding error message. Also, if the base/target is currently on the map, print the second message. Finally, if there is any other base/target at those coordinates, print the final message. This must be an \( O(\lg(n)) \) operation.

Output summary:
<output> := <success> | <error>
<success> := mapped <type>
<type> := base\n | target\n
CLEAR_COORDINATE(longitude, latitude)

longitude: integer[0,1024], represents the x coordinate
latitude: integer[0,1024], represents the y coordinate

You should search your spatial data structure with the specified coordinates. If there is either a base or a target at that location, remove it from your spatial data structure only. This must be an $O(\log(n))$ operation.

Output summary:
<output> := <success>|<error>
<success> := coordinate cleared
<error> := coordinate empty

ADD_ITEM(itemName, effectiveness, name)

itemName: case-sensitive string representation of an item
effectiveness: non-negative integer metric (lower valued items are used first). This is a measure of a weapon's ability to solely destroy its target, with little to no collateral damage. Thus, in the weapons cache(inventory), lower metrics will be higher priority.
namename: case-sensitive string representation of a base

These are instantly added to a mapped base's inventory (Fibonacci Heap). Insert should be O(1). If the name specified is not the name of a base in your data dictionary you will print the first message. If the base exists, but is not currently on the map, print the second message. If the base exists, but is not currently on the map, print the second message.

Output summary:
<output> := <success>|<error>
<success> := item added
<error> := base invalid|base not mapped|inventory added

TRANSFER_ITEM(name1, name2)

name1: case-sensitive string representation of a base
name2: case-sensitive string representation of a 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. 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> := first base invalid|second base invalid|first base not mapped|second base not mapped

TRANSFER_ALL(name1, name2)

name1: case-sensitive string representation of a base
name2: case-sensitive string representation of a 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. Error messages follow TRANSER_ITEM. You will not consolidate the structure to have at most one binomial heap of a given degree. This should be done in constant time.
TRANSFER_ALL_WITH_CONSOLIDATE(name1, name2)

- name1: case-sensitive string representation of a base
- name2: case-sensitive string representation of a 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. Error messages follow TRANSER_ITEM. You will consolidate the structure to have at most one binomial heap of a given degree.

CONSOLIDATE(name1)

- name1: case-sensitive string representation of a 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 four errors follow that of TRANSFER_ITEM. 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.

PRINT_HEAP(name)

- name: case-sensitive string representation of a 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 parenthesis. 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
d) output a new line ("\n")

The output corresponding the the sample f-heap below has been colored to help you identify the correspondence between subtrees and nested roots. No such coloring is necessary for your print function.

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 |
|                |
26--46  23      |
|                |
35
```

( 7 ( 24 ( 26 ( 35 46 ) 17 ( 23 30 ) 52 18 ( 41 ( 44 39 ) ) ) )

You must have at least one value within each set of parenthesis. This will be called on the **MIN** element. This means printing will begin with the min element.

Output summary:
<output>:= <success>|<error>
<success>:=<PrintRow>
<PrintRow>:= ( PrintEachNode )
<PrintEachNode>:=<NodeDisplay> <PrintEachNode>| <NodeDisplay>
<NodeDisplay>:= <value of node> <PrintRow>| <value of node>
<error>:=no items

**PRINT_DICTIONARY()**
An asciibetical traversal of your data dictionary will be done. Display each object on a new line as follows:
<type of object>: <name> at (longitude,latitude)
This should display all bases and targets entered into the dictionary. **Do not** display deleted objects. If empty, print the corresponding message.
Example:
base: Balogna at (7,4)
target: Rome at (4,5)
base: Saratoga at (23,54)
Iterators will make this task much easier to accomplish.

Output summary:
<output>:= <success>|<error>
<success>:=**as described**
<error>:=dictionary empty

**PRINT_SPATIAL()**
You will display each object in the order its corresponding key appears in the spatial data structure. Display each object on a new line as follows:
<type of object>: <name> at (longitude,latitude)
This will verify your comparator's correctness **Do not** display deleted objects. If empty, print the corresponding message.
Example:
target: Rome at (4,5)
base: Balogna at (7,4)
base: Saratoga at (23.54)
Note the changes in ordering.

Output summary:
<output> := <success>|<error>
<success> := **as described**
<error> := no spatial data

Comparators

If you are not familiar with comparators, you should begin by reading the interface description on Sun’s website. For the data dictionary, you should be inserting either bases and targets, or a single type of object that has some field to differentiate its type. It would be logical to use the String representation of the name as the key in which the Red-Black Tree (implemented in the background of TreeMap) makes comparisons for functions such as insertion, deletion, and query. However, the spatial data structure requires a comparator that supports multiple keys. Therefore, you must write your own class that implements the comparator interface. If you write your own comparator, you must specify exactly the ordering between objects. Each longitude/latitude pair successfully inserted should map to either a base or target object.

Rules for Comparator:

- Longitude/Latitude Pairs are considered equal if they have the same longitude as well as latitude.
- A Longitude/Latitude Pairs are always less than another if the former’s longitude is less than the other’s longitude.
- If both objects share the same longitude, latitude will used to break the tie. It will be compared in the same manner as longitude.

For redundancy:

x1 = x2 iff (x1.longitude = x2.longitude) AND (x1.latitude = x2.latitude)
x1 < x2 iff (x1.longitude < x2.longitude) OR ((x1.longitude = x2.longitude) AND (x1.latitude < x2.latitude))
x1 > x2 iff !(x1 < x2) AND !(x1 = x2) A row major ordering is the effect of such a comparator.

Fibonacci Heaps

Note: you must implement

- Single Insertion
- Union
- Min Removal
- Consolidate
- Merge
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.

---

**Links:**

READ THIS for help with fibonacci heaps  

More on Fibonacci Heaps for help with pointer arrangement  
http://compgeom.cs.uiuc.edu/~jeffe/teaching/373/notes/x02-fiboheap.pdf

Nice Definitions for fibonacci heaps  
http://sili.adnu.edu.ph/~rolladiv/teaching/resources/heap/tutor2.html
TreeMap  http://java.sun.com/j2se/1.4.1/docs/api/java/util/TreeMap.html
Comparators  http://java.sun.com/j2se/1.4.1/docs/api/java/util/Comparator.html
Upcoming Parts-you will be implementing this interface for Part 2
http://java.sun.com/j2se/1.4.1/docs/api/java/util/SortedMap.html

Ryan Gerard