CMSC 420 Project–Summer, 2004
Version 1.01

The BIG 420 project. *

Last Modified June 6, 2004

1 General description

The primary motivation of this project is of course to get you to build and make use of data structures, and to take a look at benefits and shortcomings for each one.

The TA's motivation though is to be able to make pretty pictures with some rudimentary drawing primitives.

In project one we'll have you build a dictionary to keep track of named dots (of various sizes and colors) using java's api. You'll also get a warmup implementing an Adjacency List along with Dijkstra's shortest path algorithm.

In future parts of the project we will replace some data structures to examine performance tradeoffs. In particular you can probably expect to see the prof's old time favorite the PM3 quadtree, and the TA's favorite the B+ tree. We'll probably start off project 2 with a skiplist and a kdtree for working with spacial data.

Expect a plot to evolve by project 2, possibly about a pinball planet. Or not. :)

1.1 Part 1: Crash Course in Java, Graphs, Shortest Path

This is the introductory project meant to aquaint you with Java, the language required for all projects this semester. You will hopefully learn the meaning of packages, interfaces, Iterators, the uses of Comparator and Comparable, and how to work with the very robust java API. You'll also see that this will be a fast paced class, with this project due at 10:00pm 6/12/2004. Future projects will usually have between 2 and 3 weeks to allow you scheduling flexibility, but will also be much larger and more technically challenging. It is expected that most of you are familiar with Graphs and Dijkstra's shortest path algorithm, even if you don't quite remember them perfectly.

*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 someone who took this course last semester.

1.1.1 CMSC420: Introduction to Command Parsing, where nothing could possiblie go wrong.

This section is left in for completeness from previous semesters in case you want to 'roll your own' Command Parser. However, this semester you will be provided with the TA's 'adequate' command parser. It's not optimal(either in speed or coding style), but it is more than enough for what we will be doing.

Begin original commentary:

You are all blessed with a professor with a Ph.D. in fault tolerance. Because of this you will be expected to develop fault tolerant programs. This means bounds checking for input numbers and checking a number of possible error conditions for each command. It also means that your parser should never fail or crash because of a malformed command.

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 the standard asterisks are printed. Do not echo the erroneous command.

Your parser must completely ignore blank lines. 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 command parser will be tested separately from the remaining requirements of the project. So if you can't get a fully error checking parser you will not be hurt on the other parts of the project. You may assume that for commands other than error checking 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.

A working parser cannot make any of those assumptions. Commands should be correctly interpreted regardless of case (ie. CREATE_DOT and creaTe_DOT should both be interpreted as the CREATE_DOT command). Blank lines should be completely ignored (in particular do not print extraneous "*****"s). Whitespace in general should be ignored when parsing with one exception- any string (command name, dot name, or string arguement like the colors "RED" "BLUE", etc) cannot contain internal spaces. So a command like create_dot(foo bar, 5,6, BLUE) should be flagged as invalid.

1.1.2 Comments 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/1.4.2/download.html

The online version of the documentation is at

http://java.sun.com/j2se/1.4.2/docs/api/index.html

I highly recommend 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 ;) They can be very slow around lower level project due dates ;) Of course as in most cs classes your projects will be compiled and run on the detective cluster machines (dc.umd.edu), so you should check to be sure your projects work there. However, there should not be any portability issues as long as you develop with the correct java version. We'll eventually be working with graphics, and if you work over ssh from home you will have to figure out how to set up an X-server to do this on your own machine, something I haven't figured out in the last 2 years :) (largely due to just using java locally on my own machine).

I've heard good things about Borland's free jbuilder (registration required); You may wish to look into that.

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 project one, it will show in the not to distant future.

1.1.3 Sets vs. Maps

For your work in this class Maps will generally be preferred over Sets. Maps are like sets, except that each key entry is associated with some value which can be anything. So, for instance, if you are using a class with a name like our "Dot" class, you can use the String name as a key and the whole Dot as the value. Maps are good because you need only know the key to get back the value object you want. (ie., read in a string name from the user, and use it to look up the whole corresponding dot).

In a Set in java there is no easy way to pull out an object with a specific key other than to iterate over all the elements of the set. There is an add(Object) method which lets you add to the set, but there is no corresponding get(Object x) method. The assumption is that if you already have x, why do you need the set to return you another copy? They're the same thing! Sets in java instead have a contains(Object) method, which won't be of much use as a dictionary.

1.1.4 More on Java- pass by reference, but not really

Every semester a new group of students get cought up by the same thing in Java. They start out hearing "Java is always pass by reference" and they do silly looking things like the following:

```
void foo(String t)
  {
  t = new String("World");
  }

String s = new String("Hello");
foo(s);

System.out.print(s); //prints "Hello".  Why didn't it change?
```

In true pass by reference C++ this would have worked. But what is happening is not really pass by reference, *it is pass by value*, except what is being passed is a pointer. If you were to tranfer the above to C++ it would look like:

```
void foo(String *t)
  {
  t = new String("World");
  }

String *s = new String("Hello");
foo(s);

cout<<*s<<endl; //prints "Hello".  Hopefully obvious why
```

You can see in the second example that t is only a local copy of s. If you alter the value t is pointing at then s will see the change. However, if you point t at something else s will never know. In this example there is actually no way for foo to change s, since java Strings are *immutable* after creation. An error less obvious than the above is:

```
void foo(String t)
  {
  t = t+"World";
  }
```

This looks like concatenation, not reallocation, but that '+' operator actually allocates a new String. The above is actually just a shortcut in java for:

```
void foo(String t)
  {
  StringBuffer temp = new StringBuffer();
  temp.append(t);
  temp.append("World");
  t = temp.toString();
  }
```

It's important to realize what's going on in the background! Of course in the above example, foo still doesn't change t, but what you could do instead is:

```
void foo(StringBuffer t)
  {
  t.append("World");
  }
```

This time, since t always points to the same place 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 array. There is a quick and dirty hack to get a similar effect without building and entire class wrapper, pass a 1 element array instead:

```
void foo(String[] t)
  {
  t[0] = new String("World");
  }

String s[]=new String[1];
s[0] = new String("Hello");
foo(s); //s[0]= "World"
```

This works for a similar reason. t points to the same array in memory that s does. When an element of the array is updated by t, s will see the change as well. This ends my FYI on pass by reference, try not to get cought up by this common error :)

1.1.5 Java- last part, comparators

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

```
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:

```
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 be non obvious what it is you are doing. However, there is a better way that you can use with all of Java's sorted classes (and which you will implement in your own sorted maps later this semester). You can use a comparator:

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

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

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

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

1.1.6 Adjacency List

The only data structure actually have to implement for project 1 is and adjacency list. You'll recall that an adjacency is conceptually a 'list of lists', ie:

```
A->C->D
C
D
F->K->J
H->K
J
K
```

You should implement this with TreeSets, TreeMaps, Comparators, and any other java classes you find handy. (You of course may not use an existing graph class if one already exists. You are also admonished to not use HashMaps/Sets over TreeMap/Sets.)

To implement dijkstra's algorithm you will also need a priority queue which the Java api sadly does not provide. This can be implemented through clever use of a TreeSet/Map and Comparators(the problem with a plain set is that you can't have two elements with the same priority, so you need a way to always break ties).

Note that insertion/deletion from this structure is O(log(n)log(m)), where n is the number of nodes in the graph and m is the degree of the graph (the max number of edges incident to a single vertex). This represents a binary search to find the correct row of the list to find the starting vertex, followed by a a binary search to check for existance of the ending vertex.

You'll use the graph to implement shortest path using dijkstra's algorithm. If you have a fibbonacci heap handy the run time of this algorithm is O(V*lg(V)+E) (where E is the number of edges and V is the number of vertices). And a fibonacci heap is what you ask? Well, I have no idea, but those of you who go on to take algorithms classes will no doubt hear about fibonacci heaps again, and I just wanted to be the first to share. Let's just say that they are magical, and that you will probably *never* learn how they work. If you've printed this spec on paper, feel free to X is this paragraph to avoid reading it ever again. End digression.

So anyway, you will be required to do shortest path in O(ElgV). It would be nice if you understand where this bound comes from, so I will explain it below, but it suffices that if you implement the algorithm correctly, that is the runtime you will have.

Allow me to sketch the algorithm to explain the running time(I'm not trying to teach the algorithm here- see your book/class/newsgroup/google). Every iteration of this algorithm you are guaranteed to find the correct shortest path to exactly one node- so we know right away there will be V iterations. At each state your graph is split in two sections- the 'solved' section, for which you know the correct distances, and the rest, which have some distance values associated with them which may or may not be accurate- this set is stored in some kind of priority queue. An iteration begins by selecting the node (call it 'N') from this queue with the best distance value, adding it to the 'solved' set, and 'relaxing' all it's edges. Relaxing is when for each node adjacent to N we see if it is faster to get to that node through N than it's current best known path. We update distances and back-pointers appropriately (so we know what the shortest path actually is when we finish), and that ends the round. Note that if a node's distance value is changed, its position in the priority queue has to be fixed up somehow. (this is where the magical fibonnaci heap would come into play, it's got an advantage in this 'fix up' step). One way to do this is just to have multiple copies of the same node in the queue and ignore them when they come back up (this is the approach I will use in the explanation), or else to remove the old value before reinserting it. Either works.

Now, how long does all this take. There are V rounds, and in every round we have to pull something out of the front of the priority queue- which with a treeMap is an O(lgE) operation, so each round takes O(VlgE) time. Rather than try and deal with how many elements are added to and removed from the queue in any single round, it is easier to think about how many such operations can occur in the life of the algorithm. Every single edge in the graph has exactly one opportunity to add a vertex to the queue (during a relax operation), so there are O(E) possible insertions. If we allow duplicates, the size of the queue can grow to O(E), so we may treat each queue operation as O(log(E)). That gives O(ElogE) running time for all queue operations during the life of the algorithm. Together that gives a running time of O(VlgE+ElgE), which since E is bounded by V^2, is O(ElgV). And that's the required running time of your search;)

Please be careful with your implementation details. In particular, *do not* use a linear time priority queue. This nailed a lot of people last semester. I will time your project on *very* large inputs. Check out:
http://www.cs.umd.edu/class/spring2003/cmsc420/p2graphs/

If you scroll down you'll runtimes for the large shortest path test from last semester. A negative time indicates that the test was failed. The bottom third of students *timed out*. They may have found the right answers eventually, but all the same they got no credit for that test. The middle third did not find the correct paths. Only one third of the class got the correct output in the 60 second time limit, and you can see that most of them were much slower than the best solutions. Lots of people like to blame java for the slowness of their own code, but before you start to pick on java you need to make sure you implement your code intelligently ;) That is all, you've been warned.

1.1.7 Part 1 Command Specification

You will build a command decoder with a small set of commands(you may of course use the framework provided by the TA); you will expand it later to accommodate commands required for future parts.

The following is a list of commands you should support for part 1 and a description of the output you should give for each one. Note that for all functions, you should print `''*****\n''` followed by a `'' ==> ``` and an echo of the command given. For instance, the entire valid output to EXIT() is

```
*****
==> EXIT()
Have a nice day!
```

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.

The definitions below will use the following standard BNF definitions.

```
<dotlist>:=<dot><nl><dotlist>|<dot><nl>
<dot>:= <name> at (<int>,<int>) color:<color>
<color>:= RED|GREEN|BLUE|BLACK|WHITE
<DNE>:=Error: The specified dot does not exist.<nl>
```

Whenever a `<double>}`appears(not in this part, but it will come up later), it means a floating point decimal number printed with exactly three digits after the decimal place (including trailing zeros as necessary).

Also, when looking at the list of errors, eg.

```
<error>:= <DNE>|<NR>|<AI>|<DC>|<NZ>
```

the leftmost applicable error should always be the one printed.

EXIT() ends the program. Your program shoudl also naturally terminate (with no exit message) when end-of-file is reached. Print a goodbye message (as specified below) and exit the command interpreter.

```
Output summary:
<success>:=Have a nice day!
```

CREATE_DOT(name, x, y, radius, color) creates a 'dot' object with the appropriate name, coordinates, radius, and color. The dot will then be added to a TreeMap sorted based on the name (the data dictionary) and to one based on the coordinates. The latter structure is used to check for duplicate coordinates.

Coordinates will be non-negative. This should be an O(logn) operation where n is the number of dots already in the dictionary. The dots should be stored in an asciibetically sorted SortedMap(TreeMap for project 1). If a dot with the same name already exists print an error. If a dot already exists at the specified coordinates print an error.

```
Output summary:
<output>:=<success>|<error>
<success>:=Created dot <dot>.<nl>

<error>:=<AE>|<DC>
<AE>:=Error: Dot <name> already exists.<nl>
<DC>:=Error: Dot <other\_dot\_name> already exists at the specified coordinates.
```

DELETE_DOT(name) Removes the dot with the given name from the data dictionary(and the coordinate checking dictionary). If the dot does not exist print an error. If the dot was previously in the AdjacenctList remove it and all connecting edges from there as well. Delete should be O(logn) if that dot was not previously in the Adjacency list.

```
Output summary:
<output>:=<success>|<error>
<success>:=Deleted dot <name>.<nl>

<error>:=<DNE>
```

LIST_DOTS() Lists all dots in the data dictionary in increasing asciibetical order. This function will be used as a measure of success for the CREATE_DOT function. (Hint: You should probably be using SortedMap.values().iterator() to implement this;)

```
Output summary:
<output>:=<success>|<error>

<success>:=<dotlist>
<error>:= Dictionary is empty.<nl>
```

COLOR_DOT(name,color) Changes the color of the dot with the specified name to the color given. The change should be reflected in the map, however the kd tree itself should not be accessed for this operation.

```
Output summary:
<output>:=<success>|<error>
<success>:=Color of <name> changed from <oldcolor> to <color>.<nl>

<error>:=<DNE>
```

CREATE_PATH(name1, name2) adds a bidirectional path from the first dots specified to the second in the Adjacency List. If one or both dots does not exist print the ¡DNE¿ for the first argument not found. If the path already exists print an error. If name1==name2 you may still print the success message, however no changes need to be made to the adjacency list. (All nodes are assumed to have a path to themselves of zero length).

```
Output summary:
<output>:=<success>|<error>
<success>:=Created path (name1,name2).<nl>
<error>:=<DNE>|<AE>

<AE>:=Error: The specified path already exists.
```

DELETE_PATH(name1, name2) Deletes an edge from the Adjacency list. If either endpoint does not exist, or the edge does not exist print an error.

```
Output summary:
<output>:=<success>|<error>
<success>:=Deleted path (name1,name2).<nl>

<error>:=<DNE>|<SDNE>
<SDNE>:=Error: The specified path does not exist.
```

SHORTEST_PATH(name1,name2) Prints the shortest path from the first dot to the second dot based on the path in the Adjacency List. If either name is not in the dictionary print an error. If there is no path between the two dots (possibly because one of the endpoints was never added to the adjacency list) then print an error. Otherwise print out the shortest path, followed by the total length of the path. The weight of an edge should be based on the Euclidean distance between the endpoints of the edge. $(sqrt((x1-x2)^2+(y1-y2)^2))$

```
Output summary:
<output>:=<success>|<error>

<success>:= <name1> -> <moredots> <nl>Total length:<double>.<nl>
<moredots>:= <dotname> -> <moredots>| <name2>
<error>:= <DNE>|<NP>
<NP>:= Error: No path exists.
```

1.2 Part 2: SkipList and Drawing–Tentative

1.2.1 SkipList.java

As with last semester there is a requirement that some of your data structure code implement a standard interface and to follow certain common practices. This will allow the TA to include your data structure in his own code. All of your dictionary classes this semester (currently a skiplist, and later a 2-3 or bplus tree) will implement java's Map interface specified at:

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

This differs slightly from last semester when students had to partially support the sortedMap interface. Instead, you will just implement the Map interface, but you will still hold to the general contract given in the api for SortedMap. You will have to support at least the following constructors:

SkipList() //assumes elements Comparable SkipList(Comparator c) //assumes elements are Objects, uses c to compare

For this project you should not have any *public* methods that are not specified by the sortedMap interface(other than constructors). At the end of P2 it should be transparent to your project whether you are using TreeMaps or SkipLists (this requires that you not use the headMap(), tailMap(), subMap(), or comparator() methods provided by the treeMap class). Since the Adjacency List will not be tested in project 2, you will be required to *never* use any of java's SortedMaps or SortedSets in your own code (something easily checked with grep). Whenever you feel the need for a sorted data structure, use your skiplist ;)

The easiest way to implement the Map interface is to extend AbstractMap, which requires only that you implement the entrySet() method. The easiest way to write the entrySet is to extend AbstractSet, which requires you to implement size() and an iterator for your Map. The set that you return should have an O(1) size, that is, it is just a wrapper for your SkipList. In sun's terms, the set will eb 'backed' by the SkipList, so that changes in one are reflected in the other. However, you do not have to support any of the optional operations for sets in your entreSets, which means in particular that you don't have to deal with add or remove.

I don't have time for much more on this here, but check out both the java api and the actual TreeMap.java included with the SDK distribution(in a file src.zip in the root install folder, for win32 anyway). I'm sure much will appear about this in the newsgroup and in office hours ;)

The class must be named SkipList.java (note the case) and be located in a package called cmsc420. For a tutorial on packages see:

http://java.sun.com/docs/books/tutorial/java/interpack/

1.2.2 Part 2 Command Specification

You will build a command decoder with a small set of commands(you may of course use the framework provided by the TA); you will expand it later to accommodate commands required for future parts.

The following is a list of commands you should support for part 1 and a description of the output you should give for each one. Note that for all functions, you should print `''*****\n''` followed by a `'' ==> ''` and an echo of the command given. For instance, the entire valid output to EXIT() is

```
*****
==> EXIT()
Have a nice day!
```

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.

The definitions below will use the following standard BNF definitions.

```
<dotlist>:=<dot><nl><dotlist>|<dot><nl>
<dot>:= <name> at (<int>,<int>) color:<color>
<color>:= RED|GREEN|BLUE|BLACK|WHITE
<DNE>:=Error: The specified dot does not exist.<nl>

<framelist>:<frame>|<frame><framelist>
<frame>:Frame <int><nl><dotlist>
```

The <int>}in a frame is the 0 based frame number for the current drawing command. For instance, if you are drawing the 300th frame of ANIMATE_HORIZONTAL_PATH, then the int should be 299. The points should be printed as they appear in the ordering where $P1 < P2$ if P1.x ¡ P2.x or P1.x==P2.x and P1.x¡P1.y
P1¿P2$if P1.x > P2.x or P1.x == P2.x and P1.x > P1.y$
$P1 == P2 otherwise$

Whenever a <double>appears(not in this part, but it will come up later), it means a floating point decimal number printed with exactly three digits after the decimal place (including trailing zeros as necessary).

Also, when looking at the list of errors, eg.

```
<error>:= <DNE>|<NR>|<AI>|<DC>|<NZ>
```

the leftmost applicable error should always be the one printed.

EXIT() ends the program. Your program shoudl also naturally terminate (with no exit message) when end-of-file is reached. Print a goodbye message (as specified below) and exit the command interpreter.

```
Output summary:
<success>:=Have a nice day!
```

CREATE_DOT(name, x, y, radius, color) creates a 'dot' object with the appropriate name, coordinates, radius, and color. The dot will then be added to a SkipList sorted based on the name (the data dictionary) and to one based on the coordinates. The latter structure is used to check for duplicate coordinates.

Coordinates will be non-negative. This should be an O(logn) operation where n is the number of dots already in the dictionary. The dots should be stored in an asciibetically sorted SortedMap(SkipList for project 2). If a dot with the same name already exists print an error. If a dot already exists at the specified coordinates print an error.

```
Output summary:
<output>:=<success>|<error>
<success>:=Created dot <dot>.<nl>

<error>:=<AE>|<DC>
<AE>:=Error: Dot <name> already exists.<nl>
<DC>:=Error: Dot <other\_dot\_name> already exists at the specified coordinates.
```

DELETE_DOT(name) Removes the dot with the given name from the data dictionary(and the coordinate checking dictionary). If the dot does not exist print an error. Delete should be O(logn).

```
Output summary:
<output>:=<success>|<error>
<success>:=Deleted dot <name>.<nl>

<error>:=<DNE>
```

LIST_DOTS() Lists all dots in the data dictionary in increasing asciibetical order. This function will be used as a measure of success for the CREATE_DOT function. (Hint: You should probably be using SortedMap.values().iterator() to implement this;)

```
Output summary:
<output>:=<success>|<error>

<success>:=<dotlist>
<error>:= Dictionary is empty.<nl>
```

COLOR_DOT(name,color) Changes the color of the dot with the specified name to the color given. The change should be reflected in the map, however the kd tree itself should not be accessed for this operation.

```
Output summary:
<output>:=<success>|<error>
<success>:=Color of <name> changed from <oldcolor> to <color>.<nl>

<error>:=<DNE>
```

SET_DRAW_MODE(mode, xsize, ysize, step) Changes the mode for all graphical output commands. The mode will be either "TEXT", "DRAW", or "BOTH". If BOTH, then you should print text before drawing to the screen. "TEXT" will be the mode used when grading, the others will mostly be for your benefit in debugging and for impressing your friends. If the input is invalid (I will not test this) stay in the current mode. To be safe I will always call this function before using a drawing command, so you may use whatever default you find handiest.

xsize and ysize will specify the size of the view window used on the map. When drawing a frame the current coordinate should be treated as the center of the view window, with xsize units above and below the center and ysize units to the left and right, so that the actual window is 2*xsize by 2*ysize. The step is only used for animation commands and says how far to move each frame in the current direction of travel(always east/right now). If you are following a particular line and the next step will cause you to pass the end of that line then you should have a frame exactly at that endpoint instead.

DRAW_FRAME(xpos,ypos) Processes the first frame only of the ANIMATE_HORIZONTAL_PATH command, centered at (xpos,ypos).

```
Output summary:
<output>:=<optional textoutput><nl><success>
<success>:=Draw Frame Complete.<nl>
<optional textoutput>:=<frame>
```

ANIMATE_HORIZONTAL_PATH(xmin,xmax,ypos) The first frame should be centered at (xmin,ypos), and then you should procede right moving 'step' units each frame (as defined SET_DRAW_MODE) till reaching a frame centered at (xmax, ypos).

There is no such thing as failure, and there are no bounding requirements for xmin,xmax, and ypos (well, they will be non negative). If the current mode is BOTH or TEXT then each frame should also have a textual version printed according to the BNF.

For efficiency you should use a range search in your coordinate based skiplist (the coordinate checking structure for CREATE_DOT). If it is assumed that all y coordinates are in bounds for every frame, then the running for drawing an individual frame should be O(logn +k) where k is the number of elements in the frame, and the entire animation should be O(logn +m) where m is the total number of points drawn in every frame(points will be counted once for each frame they appear in). (ie. you

need to do a single lookup to get an iterator at xmin-xsize, and then iterate to xmax+xsize. The way you'd do this with a TreeSet is to use a subSet().iterator(), but you may accomplish in other ways (for instance, by implementing an iterator that knows how to seek() in logn time)).

```
Output summary:
<output>:=<optional textoutput><nl><success>
<success>:=Animation Complete.<nl>
<optional textoutput>:=<framelist>
```

1.3 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 ;):

```
~mhs20001/Bin/submit # part#.tar.gz
```

In all cases '#' represents the number of the project part you are submitting(1,2,3 or 4). The filename is not really important; It is important that the file is in .tar.gz format and that your Main.java and other required files are not in a subfolder of the tar file. Subfolders are allowed, such as the cmsc420 package folder.

You must include the following with every submission: All necessary source files (*.java etc.) to 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 ontime submission will overwrite any previous ontime 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: bpnode.o bptree.o cell.o main.o pmedge.o pmpoint.o  pmquadtree.o util.o
    $(CC) $(LFLAGS) *.o -o proj4

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

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

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

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

clean:
    rm -f *.o
    rm -f proj4
    rm -f core
```

1.4 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 diffing 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 diffing because there is no guarantee that we will have the same output. In these cases your project's output will be pre-processed. In the case of the B+ tree, for instance, this program will verify that each node has the correct number of keys, that they are correctly ordered, and that all the correct data is at the leaves(and any other rules I may have left out).

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

Typically each test file will be worth 10 points, and you will be eligible for either 10 or 0 points depending on whether you pass for fail that test. There is no partial credit for an individual test. 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.

1.5 Standard Disclaimer: Right to Fail(twice for emphasis!)

As with most programming courses, the instructor reserves the right to fail any student who does not make a good faith effort to complete the project.

If you have problems with completing any given part of the project please talk to Dr. Hugue immediately– do not put it off! While the TA enjoys failing students, Dr. Hugue does not, so please be kind and do the project. A submission that gets only 20 or 30 points is considerably better for you than no submission at all.

1.6 Integrity Policy

From Dr. Hugue:

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 my goal, we have a very liberal code use and reuse policy. First and foremost, use of code produced by anyone who is or has ever taken 420 from me requires email from provider and user to be sent to the instructor.

The instructor 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, or who refuses to adhere to the policies stated herein.

Similarly, if you have a committment that you suspect will cause you to miss a project deadline or exam, it is your job to inform Dr. Hugue by email ASAP. Documentation may be required to support extension requests made less than one week prior to the scheduled event.

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