CMSC 420 Project–Spring, 2003
Version 2.03

The BIG 420 project. *

Last Modified April 6, 2003

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 keep track of named dots (of various sizes and colors) and analyze and update those dots via a kd tree. Should be fun ;)

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 PM1 quadtree, and the TA's favorite the B+ tree.

1.1 Part 1: SkipList and KD Tree

This section describes the first portion of the project, comprising development of a command line interpreter, a basic data dictionary and the kd tree.

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

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

*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.

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/download.html

The online version of the documentation is at

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

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.

1.1.3 SkipList.java

Another first for this semester is a requirement that some of your data structure code implement a standard interface. 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 sortedmap interface specified at:

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

You may of course have any number of other public functions which you find useful for your project. [edit: I won't change this so late, but I recommend that you *do not* add extra public functions so that your class stays interchangeable with TreeMap]

As the TA implements this interface with own skiplist he may reduce this requirement by allowing you to skip some of the interface requirements. [edit: see the news group for functions which you may skip] If that happens I still think it would be a very beneficial exercise to implement the entire interface.

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/

1.1.4 KD Tree- order 2

Unlike with the skiplist you are free to implement the kd tree any way you like with no particular interface. I will though recommend the use of java's point classes (point2d) which will allow you to easily make use of java's geometric algorithms later on. I would also suggest that you generally try to follow the java Collection interface. Standards are good things ;)

Examining the command decoder, pay particular attention to the MAP_ALL() function, which represents to the most common historical use of a kd tree. The primary purpose of the kd tree is to perform 'orthogonal range search' (ie. the rectangle search in the command decoder), which with a balanced tree has guaranteed

O(sqrt(n)+k) run time, where k is the number of elements actually found. You'll note that you can count the number of points without actually visiting them all in O(sqrt(n)) time.

1.1.5 Part 1 Command Specification

You will build a command decoder with a small set of commands; 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 CLEAR_ALL() is

```
*****
 ==> CLEAR_ALL()
All structures are cleared.
```

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.

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 SkipList. (If you don't complete a working skiplist you should use a TreeMap here instead to get some credit).

If a dot with the same name already exists print an error. If a dock 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. If the dot does not exist print an error. If the given dot has already been added to the kd tree map print an error and do not remove it from the dictionary. Delete from the dictionary should be O(logn), checking for existence in the kd tree will be dependent on the structure of the tree.

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

<error>:=<DNE>|<AA>
<AA>:=Error: Dot <name> has already been mapped and cannot be deleted.
```

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.

```
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>
```

MAP_DOT(name) inserts the specified dock into kd tree map. If the dot does not exist in the dictionary print an error message. The kd tree is expected to be constructed in standard fashion, which means that your tree should be exactly the same as the TA's (so long as the map_all function is never called).

```
Output summary:
<output>:=<success>|<error>
<success>:=Dot <name> has been added to the map.<nl>

<error>:=<DNE>|<AI>
<AI>:=Error: The specified dot has already been added to the map.
```

MAP_ALL() This function will build an ideal (log n depth) kd tree from all the dots currently in the dictionary. In practice kd trees are often built to analyze data that already exists, rather than being constructed point by point. A kd tree of order 2 built this way has guaranteed theta(logn) depth and O(sqrt(n)) rectangle query search time. The first part of this operation should be to remove any dots which are currently in the kd tree. The run time of this function should be O(nlogn) expected.

```
Output summary:
<output>:=<success>|<error>
<success>:=All dots have been added to the map.<nl>

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

COUNT_RECTANGLE(lx,ly, ux,uy) prints the number of dots on the map whose coordinates are within the closed rectangle determined by the four points: (lx,ly),(lx,uy),(ux,ly),(ux,uy). Your program should be as efficient as possible, in particular it is not necessary to visit every point in order to get a correct count. If the kd tree is built using the MAP_ALL command then this should run in O(sqrt(n)) time.

```
Output summary:
<output>:=<success>
<success>:=Total count: <int>.<nl>
```

COLOR_RECTANGLE(lx,ly,ux,uy,color) This function locates all dots in the kd tree map whose coordinates are contained within the closed region determined by the four points: (lx,ly),(lx,uy),(ux,ly),(ux,uy) and sets their color to the one specified. Success can be determined with the DRAW_MAP or LIST_DOTS commands

```
Output summary:
<output>:=<success>
<success>:=Update complete.
```

PRINT_KD_TREE() prints the output of traversing the kd tree in *preorder*. If the MAP_ALL function as not been called then your output should match the TA's exactly. The output format is identical to that of LIST_DOTS, except for the order that the dots are printed and the error message for an empty tree.

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

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

DRAW_MAP() Draws all the points of the map using the java canvas class. (If you're good with java graphics you are not required to use the canvas class, but your output should be similar). Your program should stop running until the graphics window is closed(this is the default behavior of the canvas class).

```
Output summary:
<output>:=<success>
<success>:=Drawing complete.
```

DRAW_KD_TREE() Same as the DRAW_MAP function, except that the partitions of the kd tree are also displayed. Examples will be provided.

```
Output summary:
<output>:=<success>
<success>:=Drawing complete.
```

1.2 Part 2: PM1 quadtree and ShortestPath

1.2.1 Adjacency List- updated

The first [smaller] part of this project will be to implement and use an 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
```

After much deliberation, for this project it has been decided that you will implement this structure as a Skiplist of Skiplists. Code re-use is good stuff, and a skiplist is functionally(as far as writing your program) no worse than any of the java API classes that are available. If you don't like your own SkipList you may inherit someone elses from project 1. 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 are guaranteed that all paths in this project will be bidirectional. This means that you need only store each edge once instead of twice, giving you half the memory usage and half the access time but making your graph less general. This optimization is up to you(it's not much to implement).

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 put 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 as with the KDTree 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 skiplist is an O(1) expected time operation(note that it would be log(E) in a balanced binary tree), so each round takes O(V) 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(V+ElgE), which since E is bounded by V^2, is O(ElgV). And that's the required running time of your search;)

Yes, your adjacency list is to be a skiplist of skiplists, and you should go ahead and use your skiplist for the priority queue (you may either use the O(logn) operation to remove the head of the list, or you may implement a 'pop' or 'shift' operation that runs in O(1) expected/amortized time. Why expected? Well, you have update that tower for that node- and the tower may be up to about O(logn) height. Recall though that the *average* height of a node is only 2.

You might want to understand this stuff- you might have occasion to want to explain it to someone someday,or something... ;)

1.2.2 PM1 Quadtree

The previous version of this specification gave a 1x1 limit on cell partitioning. In retrospect this seems rather silly, as it does nothing to decrease the complexity of the project while basically crippling your implementation. Not that you'll ever want to use a quadtree in real life, but if you did you probably would not want an arbitrary 1x1 limit. Instead, the INIT_QUADTREE function will be modified to take 3 arguments from the user- the size of the tree, the maximum recursion depth before aborting an insert, and the minimum distance between two lines at which point they should be considered to be intersecting. The last two parameters are mostly redundant, but I am including both explicitly so that you can chose which one to use.

The purpose of the third parameter is to allow efficient intersection detection. Ideally, if an insert is going to fail it would be better to discover this *before* you start to modify the current tree.

One acceptable way to do this is with a "trial run" before your actual insert. Basically, you use your insert algorithm to visit all the leaf nodes that the new segment will be inserted into, and then you check for intersection with the elements in those leaves. If you find an intersection (or two segments which are close enough together to be considered intersecting) then you abort. Otherwise you can procede with an unchecked insert since you know that no intersections will occur. There are tradeoffs between this and the "maximum depth rule". This method will add a certain overhead (extra intersection checking) to legal inserts. This would be especially bad if the user had specialized data that he knew a priori (ahead of time) would never contain interesections. If intersections are expected with some frequency, however, this extra check may present a significant overall speedup. You might consider a hybrid approach, where you only explicity check for intersections after a certain number of splits, or at a certain depth in the tree. Or you might not ;)

As a side note, in the spirit of being generic, I recommend that you try to build your tree so that it can handle floating point coordinates as well as integer coordinates (even though Dots will always be integers).

I am mandating that we all build our trees in a standardized way, so that the structure appears identical. Your tree and nodes will have to follow the following standard rules:

1. No two points can fall in the same node
2. A point cannot be contained in the same node as a segment, unless that point is an endpoint of that segment.
3. No two segments can fall in the same node, unless they intersect at a shared endpoint *and* that endpoint is contained in the *same* node.
4. If a point falls on the boundary between more than one node, then it must be added to every node that it intersects (a point may be contained in up to four leaf nodes).
5. If a single point of a segment falls on the boundary of more than one node then it must be added to all of those nodes as above. (A single segment can of course appear in a *lot* of nodes).
6. The tree must at all times be minimal- that is, there must never be a smaller tree which follows all of the above rules and still contains all of the same data. This should only be a complicated issue during deletion.

In no case should you try to attempt to solve any pm1 problems with brute force. All that is generally required is logical pruning- ie. "If the segment does not overlap my southeast child, then don't try adding it to that child!"

And oh yes, unlike in the book, *our* skiplists have (0,0) at the southwest corner and *not* the northwest corner(the inverted coordinate system used by your monitor and most graphics apps).

Since there are a lot of complexities in how you build your quadtree there will be less stringent speed requirements then there were in P1. Still, try to make it as efficient as you can! There will still be benchmarks, and many kudos at least for the fastest implementations(maybe extra credit, but I have no say in that). Don't brute force, please? :)

That should be all you need to know to build a quadtree!

1.2.3 Part 2 Command Specification

You will build a command decoder with a small set of commands; you will expand it later to accommodate commands required for future parts.

The following is a list of commands you should support for part 2 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 a hypothtical *wink wink* CLEAR_ALL() command would be:

```
*****
 ==> CLEAR_ALL()
All structures are cleared.
```

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, it means a floating point decimal number printed with exactly three digits after the decimal place (including trailing zeros as necessary). To do this in java check out the NumberFormat class.

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.

INIT_QUADTREE(size, maxdepth, isect_dist) Sets a number of parameters for the quadtree.

size gives the upper bounds for the quadtree. After initialization the quadtree should hold points/lines that fall between [0,0] at the southwest/lower left hand corner, and $[2^{size}, 2^{size}]$ in the northeast upper right hand corner. If size is less than 2 or more than 30 print an error.

maxdepth gives the maximum recursion depth in the tree before an intersection error should be reported. A maxdepth of 0 indicates that the root would not be allowed to split- ie. the tree could hold a single edge (A,A). A maxdepth of 1 would allow the tree to be split once into four quadrants. The value given will never be 0 or 1, I only state them explicitly so that we all agree on indexing ;)

isect_dist specifies the minimum distance between two segments before the are considered intersecting. This number will be an integer as with size, however it may be negative. The actual minimum distance should be 2^{isect_dist}. A value of 0 would indicate that two lines one unit apart should be treated as intersecting. For concreteness this number should be inclusive, so that in the case of 0, (1,1) and (1,2) would be considered as intersecting. This detail is mostly irrelevant, since in floating point arithmetic 'inclusive' and 'exclusive' have very little meaning.

This command will always precede the first command which requires the quadtree. This command is only valid the first time it is successfully called (out of range is not a successful call). If an attempt is made to reinitialize the tree, print an error.

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

```
<success>:=Quadtree initialized.<nl>

<error>:=<OOR>|<INIT>
<OOR>:=Error: size out of range.<nl>
<INIT>:=Error: The Quadtree has already been initialized.<nl>
```

CREATE_DOT(name, x, y, radius, color) UNCHANGED 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 SkipList.

If a dot with the same name already exists print an error. If a dock 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. If the dot does not exist print an error. If the given dot has already been added to the Adjacency List (via CREATE_SEGMENT()) print an error and do not remove it from the dictionary. If it has already been added to the PM1 (via MAP_SEGMENT()) print an error. Delete from the skiplist should be O(logn). Some slowdown may occur from checks to the adjacency list and PM1 if either is non-empty. This command will likely be used to verify that you can search for points in those structures ;)

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

<error>:=<DNE>|<AA>|<ZZ>
<AA>:=Error: Dot <name> has already been added to the Adjacency List.
<ZZ>:=Error: Dot <name> has already been added to the Quadtree.
```

LIST_DOTS() UNCHANGED 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.

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

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

COLOR_DOT(name,color) UNCHANGED Changes the color of the dot with the specified name to the color given. The change should be reflected in the map, however only the skiplist should 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 between the dots named to the Adjacency List. If one or both dots does not exist print the ¡DNE¿ for the first argument not found. If the segment already exists print an error. This command is not related to the PM1.

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

<AI>:=Error: The specified segment already exists.
```

DELETE_PATH(name1, name2) Deletes a segment from the Adjacency list. If either endpoint does not exist, or the segment does not exist print an error. This command is not related to the PM1.

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

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

SHORTEST_PATH(name1,name2) Prints the shortest path from the first dot to the second dot based on the segments 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.

```
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.
```

MAP_SEGMENT(name1,name2) Adds a segment to the PM1.It is possible that name1==name2, this should not be an error. The PM1 should be able to know whether any given point has a path to itself or not. There is one further error to detect. If the segment intersects a segment already in the tree(except for shared endpoints), print an error and leave the PM1 unchanged. You may use this error even if the segment intersected is the same as the one you are trying to insert. Further discussion on interesection detection rules will appear on the newsgroup and later versions of the spec.

```
Output summary:
<output>:=<success>|<error>
<success>:=Mapped segment (name1,name2).<nl>
<error>:=<DNE>|<ID>

<ID>:=Error: Intersection detected.
```

UNMAP_SEGMENT(name1,name2) Deletes a segment from the PM1. If the points do not exist, or the segment does not exist, print an error. If after the operation an endpoint would be left with no adjacent segments then it should also be completely removed from the tree. The tree should be collapsed so that it is minimal- which basically means that the tree should look as if the segment had never been in the tree to begin with. (Some collapsing will need to be done). Remember that if (A,A) is explicitly added to the tree via MAP_SEGMENT, then (A,A) must be unmapped before A is removed.

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

<error>:=<DNE>|<SNF>
<SNF>:=Error: The specified segment was not found on the map.
```

NEAREST_SEG_TO_POINT(x,y) Finds the Segment in the PM1 closest to the specified point, along with the distance to the segment. If multiple segments are the same distance from the point pick your favorite. If the quadtree is empty print an error. Print the endpoints of the segment in asciibetical order.

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

<success>:= Nearest segment: <segment>. Distance: <double>.<nl>
<segment>:= (<name1>,<name2>)
<error>:= Tree is empty.<nl>
```

COLOR_SEGMENTS(lx,ly,ux,uy,color) A twist on the old version. This function will first locate all segments in PM1 which *OVERLAP* the specified inclusive rectangular region(determined by (lx,ly),(lx,uy),(ux,ly),(ux,uy Then it will change the color of the endpoints of those segments to the color specified. The segment endpoints will not necessarily be within the rectangle. Print the number of unique segments matched. Success can also be determined with the DRAW_MAP or LIST_DOTS commands

```
Output summary:
<output>:=<success>
<success>:=Update complete.  Found <int> segments.
```

PRINT_QUADTREE() Prints out the PM1 Quadtree map. Be sure to read the section on the PM1 to understand what its structure must be. When printing you must print in the following order: Northwest, Northeast, Southwest, Southeast. If you reach a leaf with a dot in it you should print the dot's name, followed by all the dots it is adjacent to in some order. If a black leaf does not contain an endpoint dot, print the segment that passes through it. A tree with only one isolated point created by MAP_SEGMENT(A,A) should have output:

```
A:A
```

IE., a one element tree should look like a leaf, even if not implemented that way.

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

<success>:= <pmtree><nl>
<pmtree>:=<black_node><nl>|<white_node><nl>|<nl><grey_node>
```

```
<grey_node>:= NW <pmtree> NE <pmtree> SW <pmtree> SE <pmtree>
<black_node>:=<name>:<namelist>|<segment><nl>
<white_node>:=
<namelist>:= <name><namelist>|<name>
<segment>:= (<name1>,<name2>)

<error>:= Tree is empty.<nl>
```

DRAW_MAP() Draws all the points and segments of the pm1 map using the java canvas class. The points should be displayed as in part1, the lines can be plain solid black lines. (If you're good with java graphics you are not required to use the canvas class, but your output should be similar). Your program should stop running until the graphics window is closed(this is the default behavior of the canvas class).

```
Output summary:
<output>:=<success>
<success>:=Drawing complete.
```

DRAW_QUADTREE() Draws the internal partitions of the quadtree, as well as the segments inside it. Do not draw the pretty colorful dots, as they will just get in the way of debugging.

```
Output summary:
<output>:=<success>
<success>:=Drawing complete.
```

1.3 Part 3: B+ and PM1 based Animation

1.3.1 On B+ Trees and Java

Before beginning this project, required reading is page 343 in your book, which begins section 10.5 "B-Trees". I now assume you've read that page;)

The thing that makes B Trees of any variety so special is the internal node, which is sized to be about one page size (or disk block) of *contiguous* memory. This way if a page fault *does* occur when accessing a node you are guaranteed to be able to perform several searches in a row without causing any more faults.

For those not familiar with virtual memory, it may be helpful to think in terms of a database. The index for a large database may easily exceed the physical memory of the machine running the database. For this reason a database manager can only keep a fraction of the index in memory at any given time. To fascilitate this, the manager works with a block size, say 4kbytes, where any 4k region of logical memory is either entirely in physical memory, or entirely swapped out to disk. If an index search needs to access a node that is not in memory, the manager must choose an existing block in physical memory to swap to disk, then copy the new block in its place. Then the search can proceed. This operation is *extremely* slow, so you would like to do it as little as possible. With a B+ Tree index, every time one of these faults occurs an entire node gets put into physical memory and all searches within that node can occur without fear of another fault. In a structure like a BST there are no such guarantee. There is no guarantee of memory locality even between a single node and its children, so every single comparison can conceivably cause another fault. In the worst case *thrashing* can occur, where, for instance, the tree gets so deep with nodes located in random portions of memory that by the time a leaf is reached the root of the tree has been pushed out of memory. In this worst case scenerio almost every comparison would require another fault, easily making simple accesses tens of thousands of times slower.

Here are some numbers I pulled off the web, the relevant comparison is "hit time" versus "miss penalty":

```
from: http://www.cp.eng.chula.ac.th/faculty/pjw/teaching/ca/vm.htm

Typical range of parameters for virtual memory
```

```
      block (page) size          512-8192 bytes
      hit time                   1-10 clock cycles
      miss penalty               100,000-600,000 clocks
      (access time)              (100,000-500,000 clocks)
      (transfer time)            (10,000-100,000 clocks)
      miss rate                  0.00001% - 0.001%
\begin{end}


      All of that said, can you see the problem with a java B+ Tree?  What's an internal
      node supposed to look like?

\begin{verbatim}
    class Node
      {
      Object key[];
      Node children[];
      int filled;
      Node(int size)
        {
        filled=0;
        key = new Object[size];
        children = new Node[size];
        }
      }
```

I hope it's obvious that the above implementation will not at all do what a B+ Tree node is supposed to do. Sure, you can build a structure that looks like a B+ Tree (and you will), but a node has absolutely no internal memory locality. The keys, in particular, come and go at essentially random times in the lifetime of the structure. There's no guarantee that two consectutive keys aren't physically located on opposite sides of the heap.

So, what to do. An option is to have an array of characters/bytes/ints to contain the keys- but 1. this limits the B+ tree to using only key types that can be represented as chars/bytes/ints/etc, and 2., there are no functions in java the let you do comparisons on fields of base types. There's no character based string library, for instance. So you would have to either write a character based comparison library specific to a single key type, or instantiate a new key object for every single comparison. And that would just be obsenely slow.

Those who miss C++ will know that the problem of a generic B+ tree can be very cleanly solved with templates. But oh well. The algorithms will all be the same, and if you ever want to use your own C/C++ B+ trees for any real applications it should not be a difficult rewrite.

Anyway, I've come up with I think a fair compromise for java, based on the idea that objects allocated consecutively are *likely* to have good memory locality. You will still have an array of keys, but the key type must be 'assignable'. That is, the contents of one key must be able to be copied into another key without changing the memory in that key. Now you'll have:

```
    class Node
      {
      BKey key[];
      Node children[];
      int filled;
      Node(int size)
        {
        filled=0;
        key = new Object[size];
```

```
    children = new Node[size];
    for(int i=0;i<size;i++)
      key[i]=key.newKey();
    }
  }

class BKey
  {
  static BKey newKey();
  void assign(BKey other);
  }
```

Notice the line `key[i]=key.newKey();}` This is necessary since the B+ Tree won't know what the object type actually is (another advantage of templates). The newKey function will have to build an entire BKey object whose memory will never change. Then, when the B+ tree is actually in use whenever a key is added to a node the assign function is used to overwrite the memory of an existing key array entry. With this trick we should at least be able to get *some* use of memory locality out of java.

I've decidided that you will actually implement *two completely endependant B+ tree classes.* One will be based on just pointer reassignment for a node's keys. The other will use the trick above to try and preserve memory locality. My advice would be to write the code one way, and then before submitting do the few modifications needed to get the other way working. You could easily have a common base class for the two B+ trees- any functions that don't require reassigning internal node keys would not be affected. In this way we can see what the performance difference is when trying very large data sets. Should be fun eh.

FYI I will have to have some code posted to specify your B+ tree and the key type. I'm not set on the actual implementation. We may have a class like 'Comparator' called 'Assigner' or something like that, along with a factory class for building empty keys. As I said, my advice is that you first implement the B+ without such considerations- just using naive pointer assignment inside your nodes, and then fix it all up when you're done;)

And yes, your B+ tree *will* implement just as much of the sortedmap interface as your Skiplists had to.

1.3.2 On your PM1

The bulk of your project 3 grade (60+% I am guessing) will revolve around building a B+ tree. The side project is to use your PM1 do to something interesting. In this case we'll use it to try to do some animation.

This is the idea: I will create some world by creating a bunch of segments- I can fill it with anything I can make out of non-intersecting line segments. Think something like 10,000 lines in a 1048576 x 1048576 world. Then I'll ask for some windowed flyby view- for instance, "slide from (0,0) to (1000,1000) in 1 unit increments with a view window that is 50 x 50". Your job will then be to use the PM1 to help select to segments to draw in each frame. What's fun about this is that you have some freedom with your implementation. The simplest approach would just be, for each frame, to do a full rectangle query from the PM1. But you will find that each frame mostly overlaps the previous one, so this would involve a lot of redundant work. You might just query the newly exposed portions of the screen for each frame, and do a linear search of your old lines to decide which are no longer on the screen. You might also do some lookahead with the PM1, so that you don't actually even have to query it for each frame. The PM1 is certainly not free to access, and if you can narrow yourself to a small set of segments a linear search through those segments could easily be faster than a PM1 access.

Hopefully meesh will appreciate this as the return of "We're not going to tell you the fastest way to do it, try to be creative." :)

In project three we'll actually use the separate adjacency list to plot paths. This way I could, for instance, draw a large maze in the PM1, and then have the adjacency list store the correct path through the maze and ask you to follow that. There's truly no limit to what we can do!

And finally, there will be two 'modes' to this PM1 application. The first will be a "to screen" application, while the second will print each frame as text output. The latter will make things easier for me in grading ;)

1.3.3 Misc Notes for P3

You do not need to wholescale replace your SkipLists with a B+ tree. There is no need to change your adjacency list. You should use the B+ for your data dictionary and your coordinate checking structure. However, as with the Skiplist the primary benchmark testing will be done by using your B+ tree class with my own code. I happen to think that the naive pointer assignment based B+ should be faster in general than the skiplist, and I hope that for large datasets we can get the assignment based B+ to outperform the skiplist. The easiest and slowest way to do this would be to do grading on a low memory non-cluster machine, but that's probably more work than I'm prepared for ;) I don't think we'll really be able to see the gain unless some of our memory gets paged to disk, and with the amount of memory the cluster machines have available I am not sure that is feasible.

1.3.4 Part 3 Command Specification

You will build a command decoder with a small set of commands; you will expand it later to accommodate commands required for future parts.

The following is a list of commands you should support for part 2 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 a hypothtical *wink wink* CLEAR_ALL() command would be:

```
*****
 ==> CLEAR_ALL()
All structures are cleared.
```

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><seglist>
<seglist>:<segment><nl>|<segment><nl><seglist>
<segment>:= (<name1>,<name2>)
```

Whenever a ¡double¿ appears, it means a floating point decimal number printed with exactly three digits after the decimal place (including trailing zeros as necessary). To do this in java check out the NumberFormat class.

This ¡int¿ in a frame is the 0 based frame number for teh current drawing command. For instance, if you are drawing the 300th frame of ANIMATE_PATH, then the int should be 299. In the ¡segment¿ the names should be in increasing alphabetical order. The segments of the segment list. when printed, should be printed in increasing alphabetical order as well based on the alphabetically lowest endpoint of each segment. Break ties with the second endpoint. For instance:

```
Frame 7:
(a,b)
(a,c)
(a,d)
(b,d)
(c,d)
```

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.

SET_BPTREE_ORDER(btree_order) will indicate the order of the B+ tree used in the data set. It will *always* be the first command. Don't bother checking to see if some OTHER command is the first. We won't do that. However, you should detect that the B+ tree order has already been set if the command appears again.

The order will never be less than 3. You should check for this condition to avoid crashing horribly on a bad input, but I won't add a specific error for this part.

Note that the following rules apply to B+ tree nodes:

Internal: must always contain between floor$((btree_order-1)/2)$ and $btree_order-1$ keys, with exactly one more child than the number of keys at all times. (this implies between ceiling$(btree_order/2)$ and *btree_order* children per node, inclusive).

Leaf: must always contain between ceiling$((btree_order-1)/2)$ and $btree_order-1$ keys, inclusive. Must not contain *btree_order* keys! This is to force consistency between our projects so that I have some chance of grading. Remember the root is an exception, in that it never has a lower bound on the number of keys it contains.

Also, whenever a value is equal to a key it must go to that key's RIGHT child. This is mandatory, meaning no credit will be given for the B+ tree if this rule is not observed.

Even if you do not implement the b+ tree you must still implement this function! Default to printing the correct `<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>:= Order set to <btree_order>.<nl>
<error>:=Error: B+ tree already initialized.<nl>
```

RANGE_DOTS(name1, name2) Lists all dots with names between name1 and name2. If name1 ¡ name2 the dots must be listed in increasing strcmp order (endpoints included, neither name1 nor name2 need actually be in the dictionary). If name1 ¿ name2 The docks must be listed in *reverse* strcmp order. Show of that B+ range search ;)

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

<success>:=<dotlist>
<error>:= No matching dots found.<nl>
```

PRINT_BPTREE() requires you to list the B+ in a breadth first search order. If you used links between internal nodes this will be easier, BFS is more complicated. Every level of the tree is enclosed in braces {}, every node is enclosed in parenthesis, every key within a node is separated by commas. Each level of the tree should appear on its own line and in order. A sample tree of order 3 is printed below.

```
{(bar)}
{(DOT3),(foo)}
{(DOT1,DOT2),(DOT3),(bar),(foo)}
```

Note the leaf DOT3 is to the RIGHT of the key DOT3:)

Even at the leaves print only the key (the dotname). If the tree is empty, print "Tree is empty." Your tree is not expected to match mine exactly. Your grade will be based on your tree displaying the properties described above in the SET_BPTREE_ORDER command.

For our order m tree: The leaves contain between 1 and m-1 keys. They may not have m keys. Internal node internal nodes must have between ceiling(m/2) and m children. There must be one fewer guides than children (no 'extra' key on the far left should be printed, even if you used one in your implementation). Your tree, of course, must also contain the correct data at the leaves!

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

<success>:=<b+rows><nl>
<b+rows>:=<b+row><nl><b+rows>|<b+row>
<b+row>:={<nodes>}
<nodes>:=<node>,<nodes>|<node>
<node>:=(<keys>)
<keys>:=<key>,<keys>|<key>
<key>:=<dotname>

<error>:= Tree is empty.<nl>
```

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 primarily 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 animating the path along a line, the line should always be at the center of the view window, 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 says how far to move each frame in the current direction of travel. If you are following a particular line segment and the next step will cause you to pass the end of that segment then you should have a frame exactly at that endpoint (don't try to advance some fractional step into the next line segment you might be drawing). Some discourse on the use of parametric representations of lines will probably be necessary if we stick with this.

```
Output summary:
<output>:=<success>
<success>:=Drawing mode changed to <mode>.<nl>
```

ANIMATE_HORIZONTAL_PATH(x,y,dist) This is meant as a first step/parial credit measure for people who don't completely figure out drawing non-orthagonal (horizontal or vertical) paths. The first frame should be centered at (x,y), and then you should procede right for the distance specified by dist incrementing each frame by the step specified in SET_DRAW_MODE.

There is no such thing as failure (I promise (x,y) will be in bounds). If the current mode is BOTH or TEXT then each frame should also have a textual version printed according to the BNF and explanatory side comments ;)

```
Output summary:
<output>:=<optional textoutput><nl><success>
```

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

ANIMATE_SHORTEST_PATH(name1, name2) Animates the shortest path from the first dot to the second dot based on the segments 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 animate the shortest path. The first frame should be centered at name1, then each frame should be formed by moving the step specified by SET_DRAW_MODE in the direction of the current line segment. If you are following a particular line segment and the next step will cause you to pass the end of that segment then you should have a frame exactly at that endpoint (don't try to advance some fractional step into the next line segment you might be drawing).

```
Output summary:
<output>:=<optional textoutput><nl><success>
<success>:=Animation Complete.<nl>
<error>:= <DNE>|<NP>
<NP>:= Error: No path exists.
<optional textoutput>:=<framelist>
<success>:=Animation Complete.<nl>
```

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 B+ Tree 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 B+ Tree.

If a dot with the same name already exists print an error. If a dock 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. If the dot does not exist print an error. If the given dot has already been added to the Adjacency List (via CREATE_SEGMENT()) print an error and do not remove it from the dictionary. If it has already been added to the PM1 (via MAP_SEGMENT()) print an error. Delete from the B+ should be O(logn). Some slowdown may occur from checks to the adjacency list and PM1 if either is non-empty.

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

<error>:=<DNE>|<AA>|<ZZ>
<AA>:=Error: Dot <name> has already been added to the Adjacency List.
<ZZ>:=Error: Dot <name> has already been added to the Quadtree.
```

INIT_QUADTREE(size, maxdepth, isect_dist) UNCHANGED Sets a number of parameters for the quadtree.

size gives the upper bounds for the quadtree. After initialization the quadtree should hold points/lines that fall between [0,0] at the southwest/lower left hand corner, and $[2^{size}, 2^{size}]$ in the northeast upper right hand corner. If size is less than 2 or more than 30 print an error.

maxdepth gives the maximum recursion depth in the tree before an intersection error should be reported. A maxdepth of 0 indicates that the root would not be allowed to split- ie. the tree could hold a single edge (A,A). A maxdepth of 1 would allow the tree to be split once into four quadrants. The value given will never be 0 or 1, I only state them explicitly so that we all agree on indexing ;)

isect_dist specifies the minimum distance between two segments before the are considered intersecting. This number will be an integer as with size, however it may be negative. The actual minimum distance should be 2^{isect_dist}. A value of 0 would indicate that two lines one unit apart should be treated as intersecting. For concreteness this number should be inclusive, so that in the case of 0, (1,1) and (1,2) would be considered as intersecting. This detail is mostly irrelevant, since in floating point arithmetic 'inclusive' and 'exclusive' have very little meaning.

This command will always precede the first command which requires the quadtree. This command is only valid the first time it is successfully called (out of range is not a successful call). If an attempt is made to reinitialize the tree, print an error.

```
Output summary:
<output>:=<success>|<error>
<success>:=Quadtree initialized.<nl>

<error>:=<OOR>|<INIT>
<OOR>:=Error: size out of range.<nl>
<INIT>:=Error: The Quadtree has already been initialized.<nl>
```

LIST_DOTS() UNCHANGED 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.

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

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

COLOR_DOT(name,color) UNCHANGED Changes the color of the dot with the specified name to the color given. The change should be reflected in the map, however only the B+ Tree should 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 between the dots named to the Adjacency List. If one or both dots does not exist print the ¡DNE¿ for the first argument not found. If the segment already exists print an error. This command is not related to the PM1.

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

<AI>:=Error: The specified segment already exists.
```

DELETE_PATH(name1, name2) Deletes a segment from the Adjacency list. If either endpoint does not exist, or the segment does not exist print an error. This command is not related to the PM1.

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

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

SHORTEST_PATH(name1,name2) Prints the shortest path from the first dot to the second dot based on the segments 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.

```
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.
```

MAP_SEGMENT(name1,name2) Adds a segment to the PM1.It is possible that name1==name2, this should not be an error. The PM1 should be able to know whether any given point has a path to itself or not. There is one further error to detect. If the segment intersects a segment already in the tree(except for shared endpoints), print an error and leave the PM1 unchanged. You may use this error even if the segment intersected is the same as the one you are trying to insert. Further discussion on interesection detection rules will appear on the newsgroup and later versions of the spec.

```
Output summary:
<output>:=<success>|<error>
<success>:=Mapped segment (name1,name2).<nl>
<error>:=<DNE>|<ID>

<ID>:=Error: Intersection detected.
```

UNMAP_SEGMENT(name1,name2) No longer tested Deletes a segment from the PM1. If the points do not exist, or the segment does not exist, print an error. If after the operation an endpoint would be left with no adjacent segments then it should also be completely removed from the tree. The tree should be collapsed so that it is minimal- which basically means that the tree should look as if the segment had never been in the tree to begin with. (Some collapsing will need to be done). Remember that if (A,A) is explicitly added to the tree via MAP_SEGMENT, then (A,A) must be unmapped before A is removed.

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

<error>:=<DNE>|<SNF>
<SNF>:=Error: The specified segment was not found on the map.
```

NEAREST_SEG_TO_POINT(x,y) No longer tested Finds the Segment in the PM1 closest to the specified point, along with the distance to the segment. If multiple segments are the same distance from the point pick your favorite. If the quadtree is empty print an error. Print the endpoints of the segment in asciibetical order.

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

<success>:= Nearest segment: <segment>. Distance: <double>.<nl>
<segment>:= (<name1>,<name2>)
<error>:= Tree is empty.<nl>
```

COLOR_SEGMENTS(lx,ly,ux,uy,color) UNCHANGED A twist on the old version. This function will first locate all segments in PM1 which *OVERLAP* the specified inclusive rectangular region(determined by (lx,ly),(lx,uy),(ux,ly),(ux,uy)). Then it will change the color of the endpoints of those segments to the color specified. The segment endpoints will not necessarily be within the rectangle. Print the number of unique segments matched. Success can also be determined with the DRAW_MAP or LIST_DOTS commands

```
Output summary:
<output>:=<success>
<success>:=Update complete.  Found <int> segments.
```

PRINT_QUADTREE()UNCHANGED Prints out the PM1 Quadtree map. Be sure to read the section on the PM1 to understand what its structure must be. When printing you must print in the following order: Northwest, Northeast, Southwest, Southeast. If you reach a leaf with a dot in it you should print the dot's name, followed by all the dots it is adjacent to in some order. If a black leaf does not contain an endpoint dot, print the segment that passes through it. A tree with only one isolated point created by MAP_SEGMENT(A,A) should have output:

```
A:A
```

IE., a one element tree should look like a leaf, even if not implemented that way.

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

<success>:= <pmtree><nl>
<pmtree>:=<black_node><nl>|<white_node><nl>|<nl><grey_node>
<grey_node>:= NW <pmtree> NE <pmtree> SW <pmtree> SE <pmtree>
<black_node>:=<name>:<namelist>|<segment><nl>
<white_node>:=
<namelist>:= <name><namelist>|<name>
<segment>:= (<name1>,<name2>)

<error>:= Tree is empty.<nl>
```

DRAW_MAP()UNCHANGED Draws all the points and segments of the pm1 map using the java canvas class. The points should be displayed as in part1, the lines can be plain solid black lines. (If you're good with java graphics you are not required to use the canvas class, but your output should be similar). Your program should stop running until the graphics window is closed(this is the default behavior of the canvas class).

```
Output summary:
<output>:=<success>
<success>:=Drawing complete.
```

DRAW_QUADTREE()UNCHANGED Draws the internal partitions of the quadtree, as well as the segments inside it. Do not draw the pretty colorful dots, as they will just get in the way of debugging.

```
Output summary:
<output>:=<success>
<success>:=Drawing complete.
```

1.4 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 1 week 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.

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!

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.

```
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
```

1.5 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.

New to this semester some of your data structures may be included with the TAs own testing code to test their efficiency and correctness. In addition, there may be some subjective grading based on your drawing functions.

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 a k-d tree test. So, if you can't get the skiplist to work you may wish to replace it with a functional TreeMap so you can get credit for the rest of the project. (This holds for other structures as well later on).

1.6 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.

1.7 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.

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