CMSC 420 Term Project–Fall, 2002
Version 4.02

A Crazy Tourist Traffic Planning Experience *

Last Modified December 8, 2002

1 The Problem:

The years is 3077. You are head of the traffic management on the tourist planet Temas. Why the heck are you here? - rumor has it that this is the planet on which the Alien Sametists first landed. Several major tourist attractions, such as skeletons of the huge space ships whose forms suggest that they were built by non-humans, support these rumors. Recently discovered are numerous fossilized tendrils of sematoplasm-interconnects between the Hanan-0-Cells once used by Sametists to harvest the planet.

The gravity laws were repealed several decades ago, and people are no longer limited to surface travel; thus, there are many flight docks on and above the planet's surface where tourists can park their rented flyers and go sightseeing. Your job is to handle traffic management on the planet as an increasing number of tourists visit every year. You are responsible for approving straight-line routes between various docks, as well as identifying new sites of interest. You are also responsible for helping tourists plan their trips- helping to find routes that give the best fly-by views of various off-route attractions.

In order to streamline your work you have decided to develop advanced software to help with these tasks.

The assignment is to be done in four segments, with the parts specified below. Part 1 asks you to construct a B+ tree to contain flight docks, to set up an adjacency list to store approved paths between the flight docks, and to build command interpreter to be used in this and later parts. You will also create a k-d tree of order 3 to index the docks based on coordinate. In part 2 you will add the ability to remove docks from the B+ tree. You will also begin testing the PR quadtree 2 dimensional spatial data structure to map the surface level of the planet, which has the highest concentration of docks.

For Part 3 you will replace the PR quadtree with PM1 quadtree in order to analyze your flights along with the docks. For part four you will use an array of PM1 quadtrees to do a simplified map of flights in 3 dimensional space. Somewhere in there you will use your graph to implement some algorithms like shortest path and minimum spanning tree. More on future parts later.

1.1 Part 1: Indexing docks and basic searches. Due Saturday October 5th at 10pm

This section describes the first portion of the project, comprising development of a command line interpreter, the basic data dictionary used to index the docks, the adjacency list used to store approved paths between docks, and the the k-d tree used to perform searches to aid you in planning.

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

You are all blessed with a professor with a PhD 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

*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.

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.
```

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_DOCK and creaTe_DOCK should both be interpreted as the CREATE_DOCK command). Spaces and blank lines should be parsed out and ignored.

My advice would be to spend some time thinking out a plan to write a parser which does not require a lot of copy and paste code. You might make parsing a command + list of arguments a single function, for instance, a la scanf. If something goes wrong, throw an exception! (This is what your TA did, and it is doable in java. No claims that this is the *best* way of doing it). If you find that you really love parsing you might check into programs like lex/jlex which are somewhat covered in the compiler class cmsc430. I wouldn't use them for this project though.

1.1.2 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. Each command spans exactly one line; commands will be uppercase, and reasonably sloppy syntax is to be supported (spaces and empty lines are allowed).

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.

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(here or for any future parts- this covers internal node size only, in part 3 or 4 you will have to allow for an independant leaf size, which may be as small as 1) . 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!

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>
```

CLEAR_ALL() initializes all data structures used in your program. This is always the first command in the test data, but can also appear anywhere after the first command, and should delete all information from the dictionary(B+ tree), and map(k-d tree and adjacency list).

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

<success>:=All structures cleared.<nl>
```

CREATE_DOCK(dock_name, x, y, z) creates a flight dock at the three-dimensional point (x, y, z) and assigns name dock_name to it. Names will be limited to 10 characters in length and will be alphanumeric or '_'; x and y coordinates will be in range [0, 1024). The z coordinate will be in the range [0,64). You should store the dock internally in an asciibetically ordered B+ tree. Dock names are case sensitive. You should not check for coordinate conflict with an existing dock in this function, however you will in the INSERT_DOCK function.

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

<dock>:= <dockname> at (<int>,<int>,<int>)

<error>:=<nameErr>
<nameErr>:=Error: Dock <dockname> already exists.<nl>
```

INSERT_DOCK(dockname) inserts the specified dock into k-d tree map. The dock to be inserted should have been created earlier using CREATE_DOCK command. If the dock does not exist print an error message. The k-d tree is expected to be constructed in standard fashion, which means that your tree should be exactly the same as the TA's.

```
Output summary:
<output>:=<success>|<error>
<success>:=Dock <dockname> has been inserted.<nl>

<error>:=<NE>|<AI>
<NE>:=Error: Dock <dockname> does not exist.
<AI>:=Error: Dock <dockname> has already been inserted.
```

CREATE_FLIGHT(dockname1, dockname2) approves flight between two flight docks. Both dock names should be valid (ie. created beforehand with CREATE_DOCK command); otherwise, output an error message. Flights are bi-directional. (In terms of underlying graph this command creates an undirected edge; the length of this edge is just Euclidean distance between given docks). Print a confirmation message on success. Note that flights are to be stored as adjacency lists. Flights can only be approved between docks inserted into the k-d tree map. If either dock has not been inserted print an error. The TA will not attempt to create flight from a dock to itself, but you should handle this case gracefully.

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

<success>:= Flight approved.<nl>

<error>:= <NE>|<AE>|<NI>
<NE>:= Error: Dock <whichdock> does not exist.<nl>
<NI>:= Error: Dock <whichdock> has not been inserted.<nl>
<whichdock>:= <dock_name1> | <dock_name2>
<AE>:= The specified flight has already been approved.<nl>
```

A note on <NE> and <NI>- the first dock that fails(based on the order the docks are given to the function) should be the one printed. The <NE> error should supercede the <NI> error. At most one error should ever be printed.

LIST_DOCKS() Lists the dock names and their coordinates in increasing alphabetical (strcmp) order of the dock names. This function will be used as a measure of success for the CREATE_DOCK function.

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

<success>:=<docklist>
<docklist>:=<dock><nl><docklist>|<dock><nl>
<dock>:= <dockname> at (<int>,<int>,<int>)

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

LIST_FLIGHTS() lists all approved flights along with their lengths. The docks should be listed in increasing strcmp order, with the docks linked to each individual dock listed in strcmp order afterwards. For instance, if approved flights are (D1, 0,0,0)-¿ (D2, 0,0,1), and (D1, 0,0,0)-¿ (D3, 0,1,0), then the output would appear as:

```
D1: D2(1.000) D3(1.000)
D2: D1(1.000)
D3: D1(1.000)

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

<success>:= <adj-list>
<adj-list>:=<adj-row><adj-list>|<adj-row>
<adj-row>:=<dockname>:  <adjEdges><nl>
```

```
<adjEdges>:= <adjEdge>(<double>) <adjEdges> | <adjEdge>(<double>)
<adjEdge>:= <dockname>
<error>:= No flights have been approved.<nl>
```

CUBE_DOCK(lx,ly,lz, ux,uy,uz) identifies and prints the docks in the map whose location on the planet is within the closed cube determined by set of eight points: (lx,ly,lz),(lx,uy,lz),(ux,ly,lz),(ux,uy,lz), (lx,ly,uz),(lx,uy,uz),(ux,ly,uz),(ux,uy,uz) Your program should be as efficient as possible- your project may be tested on *very* large inputs that should be able to finish in a reasonable amount of time(based on trials with the TA's project). The docks *must* be printed in pre-order (self, left child, right child) with respect to the k-d tree(which should be identical to the TA's). The output format is similar to that of the LIST_DOCKS command.

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

<success>:=<docklist>
<docklist>:=<dock><nl><docklist>|<dock><nl>
<dock>:= <dockname> at (<int>,<int>,<int>)

<error>:= No docks found within the specified region.<nl>
```

PRINT_KD_TREE() prints the output of traversing the k-d tree in *preorder*. Your output is expected to match the TA's exactly. The output format is identical to that of LIST_DOCKS, except for the order that the docks are printed and the error message for an empty tree.

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

<success>:=<docklist>
<docklist>:=<dock><nl><docklist>|<dock><nl>
<dock>:= <dockname> at (<int>,<int>,<int>)

<error>:= Tree is empty.<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)}
{(DOCK3),(foo)}
{(DOCK1,DOCK2),(DOCK3),(bar),(foo)}
```

Note the leaf DOCK3 is to the RIGHT of the key DOCK3:)

Even at the leaves print only the key (the dockname). 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>:=<dockname>

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

1.2 Part 2: Structure upgrades: B+ delete and PR. Due Saturday October 26th at 10pm

After several months of using and tweaking your new software you've decided you need an upgrade. For planning purposes you often add docks to the dictionary you decide against actually building on or above the planet's surface. The extra unused docks are cluttering your database, so you will add the ability do delete these unused records from the B+ tree.

You've also decided to experiment with a new PR Quadtree structure to try and improve performance for range searches on the planet. Since the PR Quadtree is only a 2-d structure you've decided to test it only on the docks on the planets surface. If successful you may expand it to handle the higher level docks at a later time.

1.2.1 Part 2 Command Specification

Functionally, for this project you will be replacing the k-d tree of part 1 with a pr quadtree. CUBE_DOCK will be replaced with RECTANGLE_DOCK since you are only working in two dimensions. You are also adding the ability to delete entries in the B+ tree.

Finally, we are making you implement some drawing to help you debug your quadtrees. See DRAW_MAP at the bottom.

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.

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!

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>
```

CLEAR_ALL() initializes all data structures used in your program. This is always the second command in the test data, but can also appear anywhere after the first command, and should delete all information from the dictionary(B+ tree), and map(PR Quadtree / adjacency list).

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

<success>:=All structures cleared.<nl>
```

CREATE_DOCK(dock_name, x, y, z) creates a flight dock at the three-dimensional point (x, y, z) and assigns name dock_name to it. Names will be limited to 10 characters in length and will be alphanumeric or '_'; x and y coordinates will be in range [0, 1024). The z coordinate will be in the range [0,64). You should store the dock internally in an asciibetically ordered B+ tree. Dock names are case sensitive. You should not check for coordinate conflict with an existing dock in this function, however you will in the INSERT_DOCK function.

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

<dock>:= <dockname> at (<int>,<int>,<int>)

<error>:=<nameErr>
<nameErr>:=Error: Dock <dockname> already exists.<nl>
```

INSERT_DOCK(dockname) inserts the specified dock into PR Quadtree map. The dock to be inserted should have been created earlier using CREATE_DOCK command. If the dock does not exist print an error message. The PR Quadtree is expected to be constructed in standard fashion, which means that your tree should be exactly the same as the TA's. If the z coordinate of the specified dock is not 0 then print an error (¡NZ¿).

```
Output summary:
<output>:=<success>|<error>
<success>:=Dock <dockname> has been inserted.<nl>

<error>:=<NE>|<AI>|<DC>|<NZ>
<NE>:=Error: Dock <dockname> does not exist.
<AI>:=Error: Dock <dockname> has already been inserted.
<DC>:=Error: Dock <dockname> already exists at the specified coordinates.
<NZ>:=Error: Beta version cannot map dock above ground level.
```

REMOVE_DOCK(dockname) deletes a dock from the B+ tree dictionary. If the dock has already been inserted then you should print an error message and leave the B+ tree unchanged. If the dock does not exist, print an error message. Otherwise print a message to say the dock has been successfully deleted.

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

```
<success>:=Deleted dock <dockname>.
<error>:=<DNE>|<AI>
<DNE>:= Error: Dock <dockname> does not exist.<nl>
<AI>:= Error: Dock <dockname> has already been inserted.<nl>
```

CREATE_FLIGHT(dockname1, dockname2) approves flight between two flight docks. Both dock names should be valid (ie. created beforehand with CREATE_DOCK command); otherwise, output an error message. Flights are bi-directional. (In terms of underlying graph this command creates an undirected edge; the length of this edge is just Euclidean distance between given docks). Print a confirmation message on success. Note that flights are to be stored as adjacency lists. Flights can only be approved between docks inserted into the PR Quadtree map. If a dock has not been inserted print an error. If an attempt is made to create a flight between a dock and itself, (ie. dockname1==dockname2) report an error.

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

<success>:= Flight approved.<nl>

<error>:= <MU>|<NE>|<AE>|<NI>
<MU>:= Error: Docknames must be distinct.<nl>
<NE>:= Error: Dock <whichdock> does not exist.<nl>
<NI>:= Error: Dock <whichdock> has not been inserted.<nl>
<whichDock>:= <dock_name1> | <dock_name2>
<AE>:= The specified flight has already been approved.<nl>
```

A note on <NE> and <NI>- the first dock that fails(based on the order the docks are given to the function) should be the one printed. The <NE> error should supercede the <NI> error. At most one error should ever be printed.

LIST_DOCKS() Lists the dock names and their coordinates in increasing alphabetical (strcmp) order of the dock names. This function will be used as a measure of success for the CREATE_DOCK function.

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

<success>:=<docklist>
<docklist>:=<dock><nl><docklist>|<dock><nl>
<dock>:= <dockname> at (<int>,<int>,<int>)

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

LIST_FLIGHTS() lists all approved flights along with their lengths. The docks should be listed in increasing strcmp order, with the docks linked to each individual dock listed in strcmp order afterwards. For instance, if approved flights are (D1, 0,0,0)-¿ (D2, 0,0,1), and (D1, 0,0,0)-¿ (D3, 0,1,0), then the output would appear as:

```
D1: D2(1.000) D3(1.000)
D2: D1(1.000)
D3: D1(1.000)
```

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

<success>:= <adj-list>
<adj-list>:=<adj-row><adj-list>|<adj-row>
<adj-row>:=<dockname>:  <adjEdges><nl>
<adjEdges>:= <adjEdge>(<double>) <adjEdges> | <adjEdge>(<double>)
<adjEdge>:= <dockname>
<error>:= No flights have been approved.<nl>
```

RECTANGLE_DOCK(lx,ly, ux,uy) identifies and prints the docks in the map whose location on the planet is within the closed rectangle determined by set of four points: (lx,ly,0),(lx,uy,0),(ux,ly,0),(ux,uy,0), Your program should be as efficient as possible- your project may be tested on *very* large inputs that should be able to finish in a reasonable amount of time(based on trials with the TA's project). The docks *must* be printed in pre-order (NW NE SW SE) with respect to the PR Quadtree (which should be identical to the TA's). The output format is similar to that of the LIST_DOCKS command.

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

<success>:=<docklist>
<docklist>:=<dock><nl><docklist>|<dock><nl>
<dock>:= <dockname> at (<int>,<int>,<int>)

<error>:= No docks found within the specified region.<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)}
{(DOCK3),(foo)}
{(DOCK1,DOCK2),(DOCK3),(bar),(foo)}
```

Note the leaf DOCK3 is to the RIGHT of the key DOCK3:)

Even at the leaves print only the key (the dockname). 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>:=<dockname>

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

PRINT_PRTREE() prints the PR Quadtree in order (NW NE SW SE). Before going in any direction you should print out the direction you are about to go. Only one direction should appear on a line. You should also indent 2 spaces for each level of depth in the tree. The dock name itself should be printed on the same line as the last direction printed. For instance, a tree containing (A,0,1000,0), and (B,1000,0,0) would print as:

```
NW A at (0,1000,0)
NE
SW
SE B at (1000,0,0)
```

If the dock (C,500,600,0) were added, the output would look like:

```
NW
  NW A at (0,1000,0)
  NE
  SW
  SE C at (500,600,0)
NE
SW
SE B at (1000,0,0)
```

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

<success>:= <prtree><nl>
<prtree>:=<black_node>|<white_node>| NW <prtree> NE <prtree> SW <prtree> SE <prtree>
<black_node>:=<dock><nl>
<dock>:= <dockname> at (<int>,<int>,<int>)
<white_node>:= <nl>

<error>:= No matching docks found.<nl>
```

DRAW_MAP(outfile) Draws the quadtree partitions, the points in the quadtree, and the graph of approved flights(which appear on ground level- z==0). (superimposed on the quadtree). You have two options for implementing this:

1. If you are using java you can use the canvas class on the web page (or any other java drawing utils that you prefer, but I hear that the canvas class is quite nice). If you choose this method you can ignore the outfile parameter.

2. If you are using c++ (or java with a few modifications) you can make use of the printquad/showquad also on the web page. Also quite easy to use. My recommendation is to the psdraw.h file I provided to output drawing commands to a file. When grading I will then manually run printquad program to produce a postscript file. *outfile* should be the name of the file you produce. If you wish to produce a postscript file directly, its name should be '*outfile*.ps'. Like the java project, if you can display the image directly to an x-term window using the showquad package or any other means you may do so.

Please use common sense in rendering your picture to a size that fits on an average size screen(1024x768 max). It is understood the small quadrants will be difficult to see in the picture, this is ok. We will just be eyeballing pictures, as long as your output looks correct we will be happy.

DRAW_MAP() is primarily for *you*. As stated, drawing is very easy to do, The goal is that by forcing you to draw your output you will have an easier time debugging, especially later projects.

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

1.3 Part 3: PM1 Quadtree- A Timeless Classic. Due Saturday November 23rd at 10pm

Upper management is impressed with your new tools but wants more. They would like to see a synergy between point-region quadtree and the collection of flights. Using the PR Quadtree you can already search for docks near an arbitrary geographic coordinate(well, you didn't in P2, but you could have!), wouldn't it be great to do the same for flights?

Management has heard wonderful things about a 'PM1 quadtree' and thinks you should integrate one into your traffic management software. So you will. The PM1 allows for the storage of information about line segments(flights), but since you will still need to keep records on the location of isolated docks you will also add the ability to store individual points in the PM1. This will allow you to more efficiently process queries about flight locations on the map. (for instance, "What flight comes closest to point (x,y)") It will still be necessary to maintain a separate graph structure, which allows efficient processing of queries such as the shortest path between two points, and calculating a minimum spanning tree(and many many others).

1.3.1 Part 3 Command Specification

The definitions below will use the following standard BNF definitions

```
<docklist>:=<dock><nl><docklist>|<dock><nl>
<dock>:= <dockname> at (<int>,<int>,<int>) of type <docktype>
<docktype>:=REMOTE | STANDARD
<flight> := <dockname> <dockname>
<DNE>:= Error: Dock <dockname> does not exist.<nl>
```

When printing a flight, the first dockname must always be alphabetically less than the second. In addition, 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).

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. If multiple dock arguements cause the same error, then the dock given first should be the one for which an error is 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.

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!

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>
```

CLEAR_ALL() initializes all data structures used in your program. This is always the second command in the test data, but can also appear anywhere after the first command, and should delete all information from the dictionary(B+ tree), and map(PM1 quadtree / adjacency list).

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

<success>:=All structures cleared.<nl>
```

CREATE_DOCK(dock_name, x, y, z) creates a flight dock at the three-dimensional point (x, y, z) and assigns name dock_name to it. Names will be limited to 10 characters in length and will be alphanumeric or '_'; x and y coordinates will be in range [0, 1024]. The z coordinate will be in the range [0,64]. Note that the boundaries are closed for the PM1. You should store the dock internally in an asciibetically ordered B+ tree. Dock names are case sensitive. You should not check for coordinate conflict with an existing dock in this function, however you will in the CREATE_FLIGHT function.

For project 3 docks may have a specific type dependant on the first letter of their name (case insenstative). So far these will include only Remote Docks (which will begin with the letter 'R' or 'r') and Standard Docks (those beginning with any other valid character). Remote Docks will have the special property that flights cannot be created to them or between them.

In project 4 more types may be added if convenient[for me :)]

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

<error>:=<BAD_COORD>|<AE>
<AE>:=Error: Dock <dockname> already exists.<nl>
<BAD_COORD>:= Error: Coordinates out of range.<nl>
```

REMOVE_DOCK(dockname) deletes a dock from the B+ tree dictionary. If the dock has already been inserted in the PM1 via CREATE_FLIGHT or INSERT_REMOTE_DOCK then you should print an error message and leave the B+ tree unchanged. If the dock does not exist, print an error message. Otherwise print a message to say the dock has been successfully deleted.

My putting this statement here does not mean that this wasn't a requirement for P2, but to make this clear B+ delete must run in O(logn) time. (As must insert)

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

<success>:=Deleted dock <dockname>.
<error>:=<DNE>|<AI>
<AI>:= Error: Dock <dockname> has already been inserted.<nl>
```

LIST_DOCKS(type) If type is ALL, lists all dock. Otherwise lists only docks of the corresponding type (STANDARD or REMOTE). If the type specified is not one of the above 3 print an error. Output should always be in asciibetically increasing order. Note that the BNF specifies all infor for the dock should be printed, not just the names.

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

<success>:=<docklist>
<error>:= <BT>|<NM>
<BT>:= Error: The specified type is invalid.
<NM>:= No matching docks found.<nl>
```

LIST_FLIGHTS() lists all approved flights along with their lengths. The docks should be listed in increasing strcmp order, with the docks linked to each individual dock listed in strcmp order afterwards. For instance, if approved flights are (D1, 0,0,0)-¿ (D2, 0,0,1), and (D1, 0,0,0)-¿ (D3, 0,1,0), then the output would appear as:

```
D1: D2(1.000) D3(1.000)
D2: D1(1.000)
D3: D1(1.000)

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

<success>:= <adj-list>
<adj-list>:=<adj-row><adj-list>|<adj-row>
<adj-row>:=<dockname>:  <adjEdges><nl>
<adjEdges>:= <adjEdge>(<double>) <adjEdges> | <adjEdge>(<double>)
<adjEdge>:= <dockname>
<error>:= No flights have been approved.<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)}
{(DOCK3),(foo)}
{(DOCK1,DOCK2),(DOCK3),(bar),(foo)}
```

Note the leaf DOCK3 is to the RIGHT of the key DOCK3:)

Even at the leaves print only the key (the dockname). 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>:=<dockname>

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

PRINT_PMTREE() Prints out the PM1 Quadtree map. The PM Tree should have the origin [0,0] at the lower left hand corner(SW Corner), with [1024,1024] at the upper right corner (NE corner). When printing you must print in the following order: Northwest, Northeast, Southwest, Southeast.

If you reach a leaf with a dock in it you do not need to worry about printing the list of flights associated with that leaf.

Our outputs should be identical regardless of implementation. A tree with one element should make the root appear to be a leaf- ie. Only the REMOTE dock would be printed out. (The only way a the root can be a leaf is is a REMOTE dock is added to an empty tree).

```
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>:=<dock><nl>| <flight><nl>
<white_node>:=

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

RANGE_DOCKS(dockname1, dockname2) Lists all docks with names between dock_name1 and dock_name2. If dockname1 ¡ dockname2 the docks must be listed in increasing strcmp order (endpoints included, neither dockname1 nor dockname2 need actually be in the dictionary). If dockname1 ¿ dockname2 The docks must be listed in *reverse* strcmp order.

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

<success>:=<docklist>
<error>:= No matching docks found.<nl>
```

SHORTEST_PATH(dockname1, dockname2) Prints the shortest path from dockname1 to dockname2. If either name is not in the dictionary print an error. Likewise if either has not been added to the PM1 print an error. Finally, if there is no path between the two docks then print an error. Otherwise print out the shortest path from dockname1 to dockname2, followed by the total length of the path. If either dock is a remote dock you can quickly jump to the "No path exists" error.

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

<success>:= <dockname1> -> <moredocks> <nl>Total length:<double>.<nl>
<moredocks>:= <dockname> -> <moredocks>| <dockname2>
<error>:= <DNE>|<NF>|<NP>
<NF>:= Error: Dock <dockname> has not been added to the map.
<NP>:= Error: No path exists.
```

INSERT_REMOTE_DOCK(dock_name) Adds a REMOTE dock as an isolated point to the PM1. You may think of this as an edge of zero length if you like. Adding the dock will have no effect on the adjacency list. If the dock does not exist, or is not a REMOTE dock, then print an error. STANDARD docks may only be added via the CREATE_FLIGHT command. If the dock has already been inserted, print an error. If a different dock has already been inserted with the same coordinates print an error with the name of the conflicting dock. This is still a 2-d problem, so if the z cooradinate of the dock is not zero, print an error.

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

<success>:= Dock <dockname> has been inserted.<nl>

<error>:= <NR>|<DNE>|<NZ>|<AI>|<DC>
<NR>:= Error: Dock <dockname> is not a REMOTE dock.<nl>
<AI>:= Error: REMOTE dock <dockname> has already been inserted.<nl>
<DC>:= Error: Dock <other_dockname> already exists at the specified coordinates.
<NZ>:= Error: Beta version cannot map dock above ground level.
```

CREATE_FLIGHT(dockname1, dockname2) approves flight between two flight docks. Both dock names should be valid (ie. created beforehand with CREATE_DOCK command); otherwise, output an error message. Flights are bi-directional. (In terms of underlying graph this command creates an undirected edge; the length of this edge is just Euclidean distance between given docks). Print a confirmation message on success. The flights must be stored in two structures, the adjacency list and the PM1. There is a restriction that flights cannot be approved to, from, or between REMOTE docks.

If an attempt is made to create a flight between a dock and itself, (ie. dockname1==dockname2) report an error. This is still a 2-d problem, so if the z cooradinate of the dock is not zero, print an error.

Right now there is no support for intersecting edges in the PM1, so if the flight intersects an existing flight print an error. In this case the PM1 should not be modified. The error should include the flight that is intersected. If multiple flights are intersected you may chose one arbitrarily. For simplicity the TA will not add a flight which intersects a REMOTE point, but you should still handle this gracefully for your own peace of mind. Note that if an endpoint of the flight is at the same coordinates a different dock which is already in the PM1, then this will be handled as an intersection error.

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

```
<success>:= Flight approved.<nl>

<error>:= <DS>|<RD>|<DNE>|<ID>|<AA>|<NZ>
<DS>:= Error: Docknames must be distinct.<nl>
<RD>:= Error: Flights cannot be created to REMOTE docks.<nl>
<ID>:= Error: Intersection detected with flight: <dockname1> -> <dockname2>
<AA>:= The specified flight has already been approved.<nl>
<NZ>:= Error: Beta version cannot map dock above ground level.
```

DRAW_MAP(outfile) Draws the quadtree partitions, the points in the quadtree, and the graph of approved flights(which appear on ground level- z==0). (superimposed on the quadtree). You have two options for implementing this:

1. If you are using java you can use the canvas class on the web page (or any other java drawing utils that you prefer, but I hear that the canvas class is quite nice). If you choose this method you can ignore the outfile parameter.

2. If you are using c++ (or java with a few modifications) you can make use of the printquad/showquad also on the web page. Also quite easy to use. My recommendation is to the psdraw.h file I provided to output drawing commands to a file. When grading I will then manually run printquad program to produce a postscript file. *outfile* should be the name of the file you produce. If you wish to produce a postscript file directly, its name should be '*outfile*.ps'. Like the java project, if you can display the image directly to an x-term window using the showquad package or any other means you may do so.

Please use common sense in rendering your picture to a size that fits on an average size screen(1024x768 max). It is understood the small quadrants will be difficult to see in the picture, this is ok. We will just be eyeballing pictures, as long as your output looks correct we will be happy.

DRAW_MAP() is primarily for *you*. As stated, drawing is very easy to do, The goal is that by forcing you to draw your output you will have an easier time debugging, especially later projects.

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

1.4 Part 4: Skiplist, PM1 delete Due Saturday December 14th at 10pm

For this project you will make use of the 3rd coordinate for the first time since project 1. In order to map this third coordinate you will implement a skiplist of pm1 quadtrees. To test the skiplist the upper bound requirement on the z coordinate has been removed changed to max_signed_32bit_int (too lazy to figure out latex right now for the caret). If the upper bound is actually an issue don't worry, all the numbers used will certainly be below 100,000, but there is no reason why your program should not handle any integer value.

As an example of what I am describing, consider the set of commands:

```
CREATE_DOCK(A,4,6,2000)
CREATE_DOCK(B,20,1000,2000)
CREATE_FLIGHT(A,B)
```

Your first action will be to query the skiplist to see if there is already a PM1 quadtree with a height value of 2000. If there is, you will insert the flight into that tree. If there isn't, then you will initialize a new PM1 with that flight, and add it to the skiplist with the key 2000. The skiplist will of course be sorted by the height. If a PM1 ever becomes completely empty because all the docks inside of it are deleted, then that is the time that the entire PM1 level should be deleted from the skiplist.

For skiplist testing make sure that you at least can handle creating a PM1 with a single isolated dock, and deleting a single isolated dock from such a PM1. I will use this ability to try and check your skiplist functionality separately from the other parts of the project.

The other major aspect of this project is the implementation of PM1 delete. Moreso than with project3 I will try and make sure there are a fair amount of points for the 'basic' cases. Be aware if you have not thought about it that PM1 delete is more complicated than just collapsing subtrees because of the division rules involved. There is IO from last semester I will put up demonstrating this with postscript pictures (I am not sure I will have the IO perfectly tuned to the requirements of the current spec, but you'll get the idea).

As far as grading there will be 130 points available but the grade will only be based out of 100. For this project that will mean that you can skip PM1 delete or the nearest flight/dock commands and still get 100 points (which means neither will be worth more than 30 points). You may still do more to get extra credit. You may find it advantageous whatever you decide to get the basic cases of PM1 delete to work. You can roughly assume that the following groups will all be counted about equally:

1. `Skiplist add`
2. `Skiplist delete (you'll need at least PM1 delete of a single point to work)`
3. `Nearest flight/nearest dock`
4. `PM1 delete`

I did say 'about', so don't expect quite 30/30/30/30. There will also be some tests for intersection detection, which were unfortunately omitted from project3s testing. Note that *There will not be any testing of B+ delete!* If you never got it quite debugged, let it be. If you've done good work on it since the end of p3 and the posting of this spec because you weren't aware of this, talk to meesh or the TA[me]. This may have already been mentioned in class. If so, cool.

Ah, and the tests aimed at testing PM1 delete will only deal with z coordinates of 0 as as with p2 and p3, so you need not have a working skiplist to get PM1 delete credit. Since there are no more graph algorithms being run, you do not need to support the adjacency list if it is not convenient to do so (I won't call list_flights, that will be about the only change).

1.4.1 Part 4 Command Specification

The definitions below will use the following standard BNF definitions. In addition, the echoing rule at the beginning of command spec 1 still holds;)

```
<docklist>:=<dock><nl><docklist>|<dock><nl>
<dock>:= <dockname> at (<int>,<int>,<int>) of type <docktype>
<docktype>:=REMOTE | STANDARD
<flight> := <dockname> <dockname>
<DNE>:= Error: Dock <dockname> does not exist.<nl>
```

When printing a flight, the first dockname must always be alphabetically less than the second. In addition, 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).

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. If multiple dock arguements cause the same error, then the dock given first should be the one for which an error is 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.

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!

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>
```

CLEAR_ALL() initializes all data structures used in your program. This is always the second command in the test data, but can also appear anywhere after the first command, and should delete all information from the dictionary(B+ tree), and map(PM1 quadtrees /skiplist).

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

<success>:=All structures cleared.<nl>
```

CREATE_DOCK(dock_name, x, y, z) creates a flight dock at the three-dimensional point (x, y, z) and assigns name dock_name to it. Names will be limited to 10 characters in length and will be alphanumeric or '_'; x and y coordinates will be in range [0, 1024]. The z coordinate will be in the range [0,max_32bit_signed_int]. Note that the boundaries are closed for the PM1(although CREATE_DOCK does not interact with any structures besides the B+ tree). You should store the dock internally in an asciibetically ordered B+ tree. Dock names are case sensitive. You should not check for coordinate conflict with an existing dock in this function, however you will in the CREATE_FLIGHT function.

For project 3 docks may have a specific type dependant on the first letter of their name (case insenstative). So far these will include only Remote Docks (which will begin with the letter 'R' or 'r') and Standard Docks (those beginning with any other valid character). Remote Docks will have the special property that flights cannot be created to them or between them.

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

<error>:=<BAD_COORD>|<AE>
<AE>:=Error: Dock <dockname> already exists.<nl>
<BAD_COORD>:= Error: Coordinates out of range.<nl>
```

REMOVE_DOCK(dockname) NOT TESTED IN PROJECT 4. Deletes a dock from the B+ tree dictionary. If the dock has already been inserted in the PM1 via CREATE_FLIGHT or INSERT_REMOTE_DOCK then you should print an error message and leave the B+ tree unchanged. If the dock does not exist, print an error message. Otherwise print a message to say the dock has been successfully deleted.

My putting this statement here does not mean that this wasn't a requirement for P2, but to make this clear B+ delete must run in O(logn) time. (As must insert)

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

<success>:=Deleted dock <dockname>.
<error>:=<DNE>|<AI>
<AI>:= Error: Dock <dockname> has already been inserted.<nl>
```

LIST_DOCKS(type) If type is ALL, lists all dock. Otherwise lists only docks of the corresponding type (STANDARD or REMOTE). If the type specified is not one of the above 3 print an error. Output should always be in asciibetically increasing order. Note that the BNF specifies all infor for the dock should be printed, not just the names.

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

<success>:=<docklist>
<error>:= <BT>|<NM>
<BT>:= Error: The specified type is invalid.
<NM>:= No matching docks found.<nl>
```

LIST_FLIGHTS() lists all approved flights along with their lengths. The docks should be listed in increasing strcmp order, with the docks linked to each individual dock listed in strcmp order afterwards. For instance, if approved flights are (D1, 0,0,0)-¿ (D2, 0,0,1), and (D1, 0,0,0)-¿ (D3, 0,1,0), then the output would appear as:

```
D1: D2(1.000) D3(1.000)
D2: D1(1.000)
D3: D1(1.000)

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

<success>:= <adj-list>
<adj-list>:=<adj-row><adj-list>|<adj-row>
<adj-row>:=<dockname>:  <adjEdges><nl>
<adjEdges>:= <adjEdge>(<double>) <adjEdges> | <adjEdge>(<double>)
<adjEdge>:= <dockname>
<error>:= No flights have been approved.<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)}
{(DOCK3),(foo)}
{(DOCK1,DOCK2),(DOCK3),(bar),(foo)}
```

Note the leaf DOCK3 is to the RIGHT of the key DOCK3:)

Even at the leaves print only the key (the dockname). 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>:=<dockname>

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

PRINT_PMTREE(LEVEL) Prints out the PM1 Quadtree map for the z coordinate of LEVEL. If the level does not exist print the standard 'empty' error. The PM Tree should have the origin [0,0] at the lower left hand corner(SW Corner), with [1024,1024] at the upper right corner (NE corner). When printing you must print in the following order: Northwest, Northeast, Southwest, Southeast.

If you reach a leaf with a dock in it you do not need to worry about printing the list of flights associated with that leaf.

Our outputs should be identical regardless of implementation. A tree with one element should make the root appear to be a leaf- ie. Only the REMOTE dock would be printed out. (The only way a the root can be a leaf is is a REMOTE dock is added to an empty tree).

It's up there at the top, but remember that the docks in a fligth must be printed in increasing alphabetical order.

```
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>:=<dock><nl>| <flight><nl>
<white_node>:=

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

RANGE_DOCKS(dockname1, dockname2) Lists all docks with names between dock_name1 and dock_name2. If dockname1 ¡ dockname2 the docks must be listed in increasing strcmp order (endpoints included, neither dockname1 nor dockname2 need actually be in the dictionary). If dockname1 ¿ dockname2 The docks must be listed in *reverse* strcmp order.

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

<success>:=<docklist>
<error>:= No matching docks found.<nl>
```

SHORTEST_PATH(dockname1, dockname2) DONE, FINITO, NOT TESTED Prints the shortest path from dockname1 to dockname2. If either name is not in the dictionary print an error. Likewise if either has not been added to the PM1 print an error. Finally, if there is no path between the two docks then print an error. Otherwise print out the shortest path from dockname1 to dockname2, followed by the total length of the path. If either dock is a remote dock you can quickly jump to the "No path exists" error.

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

<success>:= <dockname1> -> <moredocks> <nl>Total length:<double>.<nl>
<moredocks>:= <dockname> -> <moredocks>| <dockname2>
<error>:= <DNE>|<NF>|<NP>
<NF>:= Error: Dock <dockname> has not been added to the map.
<NP>:= Error: No path exists.
```

INSERT_REMOTE_DOCK(dock_name) Adds a REMOTE dock as an isolated point to the PM1 correspoding to the z coordinate of the dock with name dock_name. You may think of this as an edge of zero length if you like. If the dock does not exist, or is not a REMOTE dock, then print an error. STANDARD docks may only be added via the CREATE_FLIGHT command. If the dock has already been inserted, print an error. If a different dock has already been inserted with the same coordinates print an error with the name of the conflicting dock.

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

<success>:= Dock <dockname> has been inserted.<nl>

<error>:= <NR>|<DNE>|<AI>|<DC>
<NR>:= Error: Dock <dockname> is not a REMOTE dock.<nl>
<AI>:= Error: REMOTE dock <dockname> has already been inserted.<nl>
<DC>:= Error: Dock <other_dockname> already exists at the specified coordinates.
```

REMOVE_REMOTE_DOCK(dock_name) Removes a REMOTE dock from the appropriate PM1with name dock_name. If the dock does not exist, is not a remote dock, or is not in the PM1, then print an error. STANDARD docks may only be removed via the DELETE_FLIGHT command. If a PM1 becomes empty because of this command, then that PM1 should be removed from the skiplist (This is very important if you want skiplist delete credit, regardless of PM1 delete).

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

<success>:= Dock removed.<nl>

<error>:= <NR>|<DNE>|<NAI>|<DC>
<NR>:=  Error: Dock <dockname> is not a REMOTE dock.<nl>
<NAI>:= Error: Dock has not yet been inserted.<nl>
```

CREATE_FLIGHT(dockname1, dockname2) approves flight between two flight docks. Both dock names should be valid (ie. created beforehand with CREATE_DOCK command); otherwise, output an error message. Flights are bi-directional. (In terms of underlying graph this command creates an undirected edge; the length of this edge is just Euclidean distance between given docks). Print a confirmation message on success. The flights must be stored in the PM1. (adjacency list is still an option) There is a restriction that flights cannot be approved to, from, or between REMOTE docks. The docks will be inserted in the PM1 associated with the z coordinate of both docks (the docks must have the same z coordinate, if not print an error).

If an attempt is made to create a flight between a dock and itself, (ie. dockname1==dockname2) report an error.

Right now there is no support for intersecting edges in the PM1, so if the flight intersects an existing flight print an error. In this case the PM1 should not be modified. The error should include the flight that is intersected. If multiple flights are intersected you may chose one arbitrarily. For simplicity the TA will not add a flight which intersects a REMOTE point, but you should still handle this gracefully for your own peace of mind. Note that if an endpoint of the flight is at the same coordinates a different dock which is already in the PM1, then this will be handled as an intersection error.

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

<success>:= Flight approved.<nl>

<error>:= <DS>|<RD>|<DNE>|<LD>|<ID>|<AA>
<DS>:= Error: Docknames must be distinct.<nl>
<RD>:= Error: Flights cannot be created to REMOTE docks.<nl>
<LD>:= Error: Docks must have the same z coordinate.
<ID>:= Error: Intersection detected with flight: <flight>.
<AA>:= The specified flight has already been approved.<nl>
```

Note that the flight is the flight intersected, and follows the same alphabetical ordering rules as flights everywhere else.

DELETE_FLIGHT(dockname1, dockname2) removes a flight from the PM1. If a non-remote dock in a PM1 is left with no adjacent flights after this command then the dock should also be removed from the PM1. If the relevant PM1 becomes empty then it should also be removed from the skiplist.

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

<success>:= Flight deleted.<nl>

<error>:= <DNE>|<NFE>
<NFE>:= Error: The specified flight does not exist.<nl>
```

NEAREST_FLIGHT(x,y,z) Finds the flight closest to the coordinate x,y,z. REMOTE docks must be ignored. For simplicity any flight found must have the same z coordinate as the one given (you only need to work in one PM1). The only possible error is that no flights exist at that z coordinate. On success print the nearest flight along with the shortest distance between the flight and the point (x,y,z).

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

<success>:= Nearest flight: <flight>. Distance: <double>.<nl>
<error>:= Error: No flights exist on this level.
```

NEAREST_DOCK(dockname1, dockname2) Finds the REMOTE_DOCK closest to the Flight specified. non-REMOTE docks must be ignored. For simplicity any dock found must have the same z coordinate as the one given (you only need to work in one PM1). There are two possible errors: 1. the flight does not exist. 2.No remote docks exist at that z coordinate. On success print the nearest flight along with the shortest distance between the flight and the point (x,y,z).

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

<success>:= Nearest dock: <dockname>. Distance: <double>.<nl>
<error>:= <NF>|<NRE>
<NF>:= Error: The specified flight does not exist.
<NRE>:= Error: No REMOTE docks exist on this level.
```

PRINT_SKIPLIST() prints out the nodes of the skiplist in order. Each line will contain they key of the node (the height of its corresponding PM1). So, if I had PM1s at z coordinates/height of 3 5 and 7 the output would just be

```
3
5
7

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

<success>:=<skipnodes>
<skipnodes>:=<skipnode><nl><skipnodes>|<skipnode><nl>
<skipnode>:= <int>

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

DRAW_MAP(LEVEL, outfile) will not be tested, but has a suggested change Draws the quadtree partitions, the points in the quadtree, and the graph of approved flights for the level specified. (superimposed on the quadtree). You have two options for implementing this:

1. If you are using java you can use the canvas class on the web page (or any other java drawing utils that you prefer, but I hear that the canvas class is quite nice). If you choose this method you can ignore the outfile parameter.

2. If you are using c++ (or java with a few modifications) you can make use of the printquad/showquad also on the web page. Also quite easy to use. My recommendation is to the psdraw.h file I provided to output drawing commands to a file. When grading I will then manually run printquad program to produce a postscript file. $outfile$ should be the name of the file you produce. If you wish to produce a postscript file directly, its name should be '$outfile$.ps'. Like the java project, if you can display the image directly to an x-term window using the showquad package or any other means you may do so.

Please use common sense in rendering your picture to a size that fits on an average size screen(1024x768 max). It is understood the small quadrants will be difficult to see in the picture, this is ok. We will just be eyeballing pictures, as long as your output looks correct we will be happy.

DRAW_MAP() is primarily for *you*. As stated, drawing is very easy to do, The goal is that by forcing you to draw your output you will have an easier time debugging, especially later projects.

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

1.5 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):

```
~mhf20001/lbin/submit # part#.tar.gz
```

In all cases '#' represents the number of the project part you are submitting(1,2,3 or 4).

You must include the following with every submission: All necessary source files (.h, .cpp, .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. A file called 'project#', where # is the part you are submitting- 1,2,3 or 4. This file must contain the line needed to run your project after it is comiles. For instance-

```
proj1
```

if you used c++, or

```
java proj1
```

if you used java.

If you leave out the README or project# file your project will fail to submit! You must also include a makefile, so that by typing 'make' I can compile your project. A very basic sample makefile might be:

```
all:
javac *.java
```

or

```
all:
g++ *.cpp -o proj3
```

Here is a moderately more complicated makefile that you might use as a template:

```
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
```

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, so especially if you used g++ make sure to *not* include .o files , core, or a compiled executable. In java you should dump the .class files, but they really aren't that big...

And finally- if you had to add lines to your path in order to compile, remember that I won't have those lines in my path when I am grading! So make sure to include full paths and such in your makefile. Java people- if you are using jikes you might consider submitting a makefile which uses javac.

If there is popular demand for me to add something to my path, post to the newsgroup and I will.

1.6 Grading

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 things, such as the B+ tree printout, cannot 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 by a more complicated grading program. In the case of the B+ tree, 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).

Typically each test file will be worth 10 points, and you will be eligable 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- for instance, a test to check your k-d tree will not print the b+ tree. 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 B+ tree to work you may wish to replace it with a functional BST or similar structure so yuo can pass the rest of the project. (This holds for other structures as well).

Every early submission will overwrite the previous early submission, every ontime submission will overwrite any previous ontime submissions, 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 the 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.

1.7 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 enjoyes 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.

1.8 Integrity Policy

Your work is expected to be your own or to be labeled with its source, whether book or human or web page. Discussion of all parts of the project is permitted and encouraged, including diagrams and flow charts. However, pseudocode writing together is discouraged because it's too close to writing the code together for anyone to be able to tell the difference.

Since the projects are interrelated, and double jeopardy is not 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. Adoption of a BST is permitted for any part of the project, provided that all other rules are observed.

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.