CMSC 420 Term Project–Spring, 2002
Version 4.0
Constructing Hanan University Too (HUT) on Planet Temas-Too*
Last Modified May 6, 2002

1 The Problem:

The nefarious alien Sametists have spent the last few centuries looting the known universe and spreading their message that it is good to know how to design spatial data structures, and not just how to use them. For most planets, the Sametists choose to “merrily strip-mine”1 the planet’s abundant natural resources. It takes lots of raw materials to fuel their plan to colonize the galaxy and to promote the use of spatial data structures universally.

After suffering heavy casualties when a Sametizing center on Temas-ton2 exploded just before the portal to Temas-Ty-Went opened, the Sametists developed bioMechs to make planets habitable for the Sametists without undue loss of life. Thus relieved of any dangerous job assignments, the Sametists returned to pondering the implications of their closure assumptions upon the dimensions and growth rates of their spatial data structures without fearing for their lives.

At any rate, the Sametists employ a process called Hanan-o-Forming, that uses families of biomachines, from nanomachines capable of modifying a genotype, up to MegaBeasts, who dig, reinforce, and excrete the massive lined tunnels that honeycomb each planet infested with Sametists, and provide various service and maintenance functions throughout every level of the habitat. Once the planet has been Hanan-o-Formed, it can be settled and yet another planet of indefatigable Sametists can be trained. Planet Temas-Too is the latest planet to have been infested, and, to educate the teeming fledgling sametists, a 2nd university system, Hanan University Too (HUT).

This project will guide you in erecting Hanan University Too (HUT) using the Hanan-o-Forming Process (HOP). The HOP devours the natural resources of the planet and extrudes the standardized environment that provides the appropriate living and learning atmosphere for Sametists. Note that the HOP lays out and develops the intricate cell–web structures for which MegaBeast-based habitats are renowned, while minimizing the costs associated with the HOP. Legend has it that they follow the ancient rules of Pascal, but no one has been able to verify that suspicion.

The atmosphere of Temas-Too is too toxic for the Sametist’s nictating membraneous respiratory structures, which provide the host cells for many of BioMech’s Hanan-o-bots, or h’obots as they are called. So, a special environment must be established before the MegaBeasts can be unleashed beneath the planet’s surface. Once the MegaBeast-ready sites have been completed, the MegaBeasts and their companion E-bots are unleashed to construct learning sites and the supporting interconnection network that will make up HUT.

Motivation:

The ulterior motive of this project is to force you to explore the process of combining data abstractions based on mixing and matching specific properties or attributes of the real-life or made-up processes to produce composite data-structures that facilitate providing the required service to the customer. That is,

* Participation in this project may prove HAZARDOUS to your health. Unfortunately, failure to participate early and often will definitely have an adverse effect upon your GPA. Take my advice. Start now, because you’re already behind. If you don’t believe me, ask someone who took this course last semester.

1 Tony Kornheiser referring to Robin Givens’ conduct during her short-lived marriage to Mike Tyson.

2 A world plundered by the Alien Sametists during a previous project.
what tradeoffs are necessary to satisfy potentially conflicting constraints of time, cost, and space? Note that non-functional requirements\footnote{This is a little sarcasm on my part: Requirements that do not specify a function (action) of a controlled object, such as a battleship or a life-critical system component, are often referred to as non-functional requirements, even though they may require additional component functions.} such as the need for real-time performance, demands that the code satisfying the process constraints be efficient, as well as correct, because failure is both dangerous and costly. So, additional constraints on the problem may become necessary as we begin to understand the issues associated with developing a HOP HUT. Furthermore, the lessons learned in one portion of the project will definitely help you in later parts.

**Project Overview**

The assignment is to be done in four (4) segments, with the parts (and due dates) specified below. A command interpreter, to be used in all parts of the project, must be constructed to facilitate design and testing of your project. The command decoder will be modified throughout the course; so, you might find yourself upgrading this more often than you think :-).

Part 1 asks you to implement a data dictionary (DD), ordered by cell name, using a Binary Search Tree (BST). The DD will contain the information gathered regarding potential cell locations, organized by name or CID (cell identifier). Then, you will implement a PR quadtree to represent the spatial relationships among the cells chosen for implantation of MegaBeasts throughout the planet. The long term goal is to expand this PR quadtree into a map of all existing knowledge sites and critical inter-site pathways.

In part 2, you will implement the DD using a B+ tree, and will use an adjacency list in congress with the PR quadtree from part 1 to plan and track the generation the biometric web that permeates planet’s crust and supports the Mega-beast processes. This web will form the foundation of the paths to be traversed by Sametists as they move from knowledge site to knowledge site, picking up the diverse concepts that are the mark of the true Sametist. The PR quadtree-based map from part 1 will be expanded to include the inter-cite web, or E-bot net, and to detect, points of intersection within the E-bot net. You will also develop the ability to draw the HUT map on an X-based machine so as to support debugging of the updated map structure in part 3.

In part 3, deletion will be added to the B+tree, and the PR quadtree-based map will be expanded to become a PM1 quadtree which will represent spatial relationships among cells and will capture the planned E-bot net necessary to link the knowledge cites and to indicate directional choices along the E-bot net. In part 4, deletion in the PM1 quadtree is introduced, to further capture points, lines, and planes appropriate for the HUT in a single structure, and to allow planning of the final HUT campus using heuristic algorithms to deal with complex mapping constraints. Each part of the project will also include several applications essential to making this venture profitable.

In each part, points are allotted for program documentation, conformance to requirements and efficiency. Furthermore, a separate BNF specification for the output will be posted for each part, along with sample input and output files. Note that student assistance in generating test data files is always appreciated.

Make sure that you follow all instructions specified in this document, the webpage, the newsgroup and in the class. If you don’t implement an instruction, make sure you follow the directions and fail passively, because we won’t grade past a core dump. And, you only get one complimentary core dump per semester.

The most current version of this document and the corresponding bnf will be posted on the webpage at all times. We will not be able to incorporate ALL modifications or clarifications to the specifications. Thus, the project specification consists of the most recent version of the document on the webpage, plus any modifications posted after the last update of the main version of the specification. So, keep track of postings by the ta or by me that might improve your understanding of the project. Remember, always make sure that your project complies with the most recent postings before submitting. In general, we will freeze the specification at least 1 week before the project due date.

### 1.1 Part 1: Developing Hanan-0-Cells: (Due: February,16-18) Max. points: 100

This section describes the first portion of the project, comprising the basic data dictionary (DD points), the PR tree (PR points), and the applications (AP points) needed to situate the MegaBeast mother cells
beneath the planet’s surface. The breakdown of points is posted below.

1.1.1 Part 1 Command Specification

You will also build a command decoder with a minimal set of commands; you will expand it appropriately 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). Documentation and conformance to the rules is worth 10 points for part 1. The clear all function is worth 5 points. The data dictionary development (DD points) is worth 15 points. There are 30 points for developing the map (PR points). And, 40 points are allotted to the rectangle search (AP points).

The following is a list of commands you should support and a description of the output you should give for each one. Note that for all functions, you should print "*****\n" followed by an " =\n" 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 you are not required to reformat the orginal command (fixing spacing, for instance) in any way.

- **CLEAR\_ALL()** [5 points] initializes all data structures used in your program. This is always the first command, but can also appear anywhere after the first command, and should then cause deletion of all currently defined cells and routes. Print “All structures cleared.” after execution, even if this is the first command in the data file.

- **CREATE\_CELL(cell\_name, x, y)** [DD points] adds the name of the cell along with its coordinate position into the data dictionary. The data dictionary should be implemented as a binary search tree (BST) with keys in the natural ASCII order (ie, strcmp() order- henceforth ascibetical order) of the cell names. In the case of duplicate keys, we assume that “less than” values are to be inserted into the left subtree and ‘greater than or equal to’ values are inserted into the right subtree. Cell names will be composed of up to 6 characters that are _ or alphanumeric. Coordinates will be in the range [0, 1024).

  *The mapping between cell names and coordinates is one-to-one, meaning that you can’t have two instances of the same coordinates inserted having different cell names and vice versa.*

  While creating a cell that exists already is an error, at this stage we are only trying to catch duplicate names. That is, while two cells with different names are forbidden to have the same coordinates there is no intention to put this specific error in the test data until Part 3, and you are not required to catch that error NOW for reasons that should be obvious if you try to think of an efficient way to check for that condition.

On success you should print:

```
Created cell <cell\_name>.
```

The following are possible error messages:

- **Error:** Cell <cellname> already exists.

- **LIST\_CELLS()** [DD points] Lists the cell names and their coordinates in increasing alphabetical (strcmp) order of the cellnames. This function will be used as a measure of success for the CREATE\_CELL function.

  If there are no created cells simply print "None." Otherwise output the cells one on each line in increasing ascibetical order in the format

3
<cellname> at (<x-coordinate>,<y-coordinate>)
...
<cellname> at (<x-coordinate>,<y-coordinate>)

followed by

End of list.

Note, contrary to the original description and my original samples, ”End of list.” is not printed if there is no list printed. Just print ”None.”

- INSERT_CELL(cell_name) [PR points] inserts the specified cell into PR quadtree. The cell to be inserted should have been created earlier using CREATE_CELL command.

On success you should print:

Cell <cellname> has been inserted.

The following are possible error messages:

Error: Cell <cellname> is not the name of a created cell.
Error: Cell <cellname> is already inserted.

Note that for part one no two Cells will have the same coordinates so you are not required to check this condition.

- PRINT_PRTREE() [PR points] prints the output of traversing the PR tree in preorder. You will be required to print out the path you take for your benefit and ours. For the most clear explanation see the samples that are posted. Basically, before examining any child you should output the direction you are about to go. When you hit a leaf node print either the Cell (in the form ”<cellname> at (<x-coord>,<y-coord>)”, or the word EMPTY for an empty leaf. ie. A tree with 1 Cell (Foo,20,73) would print as:

    Inserted Cells:
    Foo at (20,73)

since there are no directions to look (the root is a leaf). If you added (Bar, 800,800) you would end up with 4 children from the root and print the following:

    Inserted Cells:
    NW EMPTY
    NE Bar at (800,800)
    SW Foo at (20,73)
    SE EMPTY

and if you added one more point (Hello,520,520) you would print:

    Inserted Cells:
    NW EMPTY
    NE
    NW EMPTY
    NE Bar at (800,800)
    SW Hello at (520,520)
    SE EMPTY
    SW Foo at (20,73)
    SE EMPTY

4
If the tree is empty you should print:

**Tree is empty.**

(Note this is a change from earlier versions. Don’t say "Inserted Cells:" if there aren’t any)

- **RECTANGLE_CELL(c_x, c_y, x_l, y_l)** [AP points] Identifies and prints the names of all cells in the PR quadtree whose location on the planet is within the closed rectangle determined by set of four points:

\[
\{(c_x + x_l, c_y + y_l), (c_x + x_l, c_y - y_l), (c_x - x_l, c_y + y_l), (c_x - x_l, c_y - y_l)\}
\]

You must use the PR quadtree efficiently (as described below) to prune your search in order to get full credit [You will need to match the TA’s code exactly].

With this simple rectangle there is an exact amount of pruning that can be done—namely, you need only follow a child if the rectangle overlaps the child’s region at least one point. You will be required to print in the same format as in PRINT_PRTREE, except for the following:

1. When you reach an an empty leaf you will not print EMPTY.
2. When you reach a leaf with a point that is outside the rectangle, do not print the cell name.
3. Do not print directions that you did need to search.

After printing the search path you took, you are to list the cells in NW-NE-SW-SE order in a list similar to the output of LIST CELLS(). If you only accomplish this second part you will no more than one half of the possible points for this function. For example, in the tree above the output to **RECTANGLE_CELL(800,700,20,120)** would look like:

**NE**

- **NE Bar at (800, 800)**

**SE**

**Cells within rectangle:**

- **BAR at (800, 800)**

Explanation: Starting from the root, the NorthWest quadrant from \(x = [0, 512] y = [512, 1024]\) does not overlap the rectangle \(x = [780, 780] y = [580, 820]\), so you don’t go NW, etcetera... Think of the output as representing all the quadrants that a point in the rectangle might have been in... Longer examples are in the sample online.

If there are no points in the given rectangle, then after printing the search path you should print only:

**No cells within the specified region.**

This is also a change from earlier specification.

Note: Although you have all doubtlessly implemented your quadtrees in different ways, the PRINT_PRTREE and RECTANGLE CELLS commands should behave as if they were working with the theoretical model of a quadtree—ie. each node is either a leaf or has four children. Just because you use a NULL pointer to represent an empty leaf, for instance, does not mean that you can skip printing out that direction in your traversal. For this reason your output must match the TA’s exactly for these functions to be considered correct. You will be given test files to help ensure that you are formatting correctly.

Note2: You may adopt anyone’s BST, so long as it is properly attributed in your documentation. This includes your own BST from 214 (!). The BST need not be balanced; however, no deduction will be
made for a correctly working balanced BST. If you are truly ambitious, you can build the B+‐tree early. The functions in part one will should give the same result for a BST, a B+‐tree, or even an AVL tree, because we aren’t checking the dynamic (run time) code size, or the execution time. However, there is no need to do so. You are required to write your own command decoder and PR, but the PR quadtree can be built by extending an existing BST ADT, for example. Or, you can build it from scratch.

1.2 Part 2: EnviroBot Net and Nutrient Webs, Due: Mar 16-18, 2002, Max. points: 150

Hanan-O-Forming is based on the process of doing the hardest task in each stage of a project first. And, this task is chosen only after a careful assessment of the simplicity and independence of the remaining tasks. That is, the complex parts are worked on in parallel with the assembly of simpler tasks, thus maximizing the probability of executable code, while constantly improving the number of tasks in the testing stage. That way, project integration and testing is merely a matter assembling parts. Modification and process improvement becomes a matter of upgrading a given module. Just a suggestion.

In order to continue the development of the h'bots required to H’form the planet, EnviroBots are injected at an initial MacroCell cell and multiply quickly, extruding the nutrient meshes that infiltrate every layer of the planets’ composition. The EnviroBots cooperate, first connecting the chosen MacroCells through the EnviroBot (EBOT) net and then mingling their individual nets to create the Nutrient Web, the series of concentric shells of living, biometallic webbing that are the unmistakable sign of a Samaist enclave. Note that places where the EBOT net crosses are called PseudoCells, because they can be used just like an initial MacroCell. Identification of PseudoCells and insertion into the data dictionary and map is done to ensure an accurate view of the EBOT net.

Your job is to program the EnviroBots to do their job.

**Self-Assessment:** If you find yourself redesigning your entire mental model of the project framework every time we integrate a new function into the current project phase, you are not “getting” the project. You should begin to recognize a “standard” for this project—certain terms and output styles will persist, and not change much after part 2. You are supposed to have to rethink things, but not the most basic—the parts of the project that i allow you to adopt for part 3, regardless of your grade on part 1 and part 2 are the minimum that you should be able to handle by spring break. That is, if you didn’t get parts one and two to work, you should ask for code and start to work asap.

That is, think through the entire project to identify potential difficulties in satisfying each deliverable function or testable requirement identified during this CAREFUL reading of the specification.

1.2.1 Part 2 Command Specification

Part 2 of the project requires the implementation of a B+‐tree for the data dictionary; use of the PR quadtree and an adjacency list in an application; and the addition of the required commands to the command decoder.

Note that for full credit on part 2, the functions listed below must replace any with the same command name, as we will be adding parameters and changing the underlying structures from those used in part 1.

As before, each command spans exactly one line; commands will be uppercase, and reasonably sloppy syntax is to be supported (spaces and empty lines are allowed). Documentation and conformance is worth DOC points. Don’t forget to check the BNF in section as well.

- **SET_BPTREE.ORDER**(btree order) [DD points] will indicate
the size of the B+‐tree used in the data set. It will always be the first command in the data set. 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.

Your output should look like

**Error:** B+ tree already initialized. The current order is:<current-order>
• CLEAR_ALL() [DD points] 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 then cause deletion of all currently defined cells and routes. This also resets j to 0. Print "All structures cleared."

after execution, even if this is the second command in the data file.

• CREATE_CELL(cell_name, x, y) [DD points] adds the name of the cell along with its coordinate position into the data dictionary. The data dictionary should be implemented as a B+ tree with keys in the natural ASCII order (strcmp()) order) of the cell names. Cell names will be composed of 6 characters that are _ or alphanumeric. Coordinates will be in the range [0, 1024]. Print a confirmation message if this command is processed successfully. There used to be an assumption about the max number of cells which could be added. You may make no such assumption.

As before, creating a cell that exists already is an error. However, there is no need to detect that two cells with different names have identical coordinates, and, there is no intention to put this specific error in the test data until Part 3.

• LIST CELLS() [DD points] Output identical to part 1. Lists the cell names and their coordinates in increasing alphabetical (strcmp) order of the cellnames. This function will be used as a measure of success for the CREATE_CELL function. Print "None" if no cells have been created. For part 2 Cells should now be printed with 3 digits past the decimal. This means most of your points will end in .000 See sample IO for details.

• PRINT_BPTREE() [DD points] Lists the B+ in a breadth-first search order. If you used links between internal nodes this will be trivial, BFS will be a little more complicated. Check the web page for pseudo code (soon).

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)}
{((CELL3),(foo))}
{((CELL1,CELL2),(CELL3),(bar),(foo))}

Even at the leaves print only the key (the cell name). 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 following properties (this does deviate somewhat from shaffer, you are required do what what is listed below):

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 guide 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!

• INSERT_CELL(cell_name) [MAP points] inserts the specified cell into PR quadtree just as the coordinates are listed when they are inserted in the CREATE_CELL. An attempt to insert a cell already present in the PR quadtree should result in an error message (same error as part1).

Note: At the same time that a cell is inserted in the PR quadtree, the EnviroBot (EBOT) net connecting existing cells is extended to include the additional cell. Details of EBOT net development are included here for your convenience. When a cell is inserted in the PR quadtree successfully, the cell must be added to the EBOT net by finding the cheapest edge between the existing cell and all other cells already in the net.

The edge cost is computed according to the following algorithm for inserting the j-th cell into the path.
If \( j = 1 \) the cost is 0. For \( j > 1 \), the edge cost between cells A and B having coordinates \( x_A, y_A \) and \( x_B, y_B \), respectively, is

\[
\text{cost}_{AB} = \left( (x_A - x_B)^2 + (y_A - y_B)^2 + (\max(r_A, r_B))^j \right)^rac{1}{j}
\]

where \( r_A = \text{sqrt}(x_A^2 + y_A^2) \) and \( r_B \) is defined similarly.

So, if cell B is being inserted in the PR quadtree and there are already \( j - 1 \) cells in the service path, an edge will be inserted from cell B to the existing cell A for which \( \text{cost}_{AB} \) is minimal. And, yes, that means the number of cells already in the EBOT net also contributes to the cost. However, once an edge has been inserted in the EBOT net, no attempt should be made to recompute the cost.

Furthermore, as the cost increases, it may be more reasonable to keep the cost in two pieces—the portion that dominates, with the \( j \)-th power tacked in, and the portion from the difference in \( x \) and \( y \) coordinates...this will keep the need to use double precision to store humongous integers from introducing unneeded errors. Do what you think is right.

We're this a REAL application, you might suggest that the developers consider using a disconnected EBOT net once the number of cells makes the cost prohibitively large—on the other hand, we really don't know the units of the cost function, now do we, so these values might be what the customer expects.

Full credit will be given for brute force (meaning always compute all the distances to all cells in the net). However, up to 10 points extra credit may be given if your algorithm need not always check all known cells to find the closest—and sometimes, reasoning based on the size of a given portion of the cost function can allow you to prune the PR quadtree.

If you do choose this option, make sure you describe your algorithm within your code, and explain your algorithm and why it is better than brute force in the README.

p.s. The cost function doesn’t have any particular rationale, but ya gotta live with it ;)

**Important: Last Minute Clarification of Ebot Net cost function**

Since the last term of the cost function blows up quickly, the client has agreed to accept the following modification. Floating point may be used as an approximation to the cost and double precision gives best results. The BNF now reflects this change, with `<float>` replacing `<int>` as needed.

Furthermore, suppose you are adding cellname D to the EBOT NET. When comparing cost \( c_A \), the cost of a potential edge connecting A to D, to cost \( c_B \), the cost of a potential edge between \( B \) and \( D \), you find that the following is true:

\[
\frac{|c_A - c_B|}{\max(c_A, c_B)} < 10^{-5}
\]

Then, you may continue as if the cost to connect \( D \) to either \( A \) or \( B \) were identical, and use `strcmp` order of the cell name to discern which of \( A \) or \( B \) gets the connection. Please note that if you compute the function as written, you will have a loss of precision when \( c_A \) and \( c_B \) differ by only a few low order digits.

So, you might want to consider the following alternative if you need to preserve the actual difference for some other reason. This method can be evaluated without the loss of precision caused by subtracting two nearly equal floating point numbers.

If \( c_A < c_B \), then the cost of the edge to A is treated as matching that to B whenever

\[
c_B(1 - 10^{-5}) < c_A
\]

Similarly, if \( c_B \leq c_A \), then if

\[
c_A(1 - 10^{-5}) < c_B
\]

the edges should be treated as having equal cost.

(Note that the previous discussion assumes that \( c_A \) and \( c_B \) are non-negative.)
• PRINT_PRRTREE() [MAP points] prints the output of traversing the PR quadtree in preorder, just like in part 1.

• PRINT_EBOT_NET() [MAP points] prints the adjacency list corresponding to the minimal network eaten through the planet’s interior by the EnviroBots in preparing the MacroCells for implantation of the MI's. This represents a minimal network connecting all the MacroCells activated thus far, as described above under the heading INSERT_CELL.

Requirements for demonstrating which Cells you visited have been removed, as in order for me to verify that you haven’t fudged your results I would have to examine your code anyway. Extra credit for cutting down the total number of comparisons will therefor be based on the description of your algorithm in your README.

You will also never explicitly print the cost of the EBOT-NET (seems odd doesn’t it, you should think about why I have done this). Simply print what the network is. Your network will be expected to match exactly unless pseudo cells are involved.

The cells should be listed in stremp order, with the nodes linked to each individual cell listed in stremp order afterwards. (Ok, not too helpful) Try:

CELL1: CELL2 CELL 3
CELL2: CELL1
CELL3: CELL1 CELL4
CELL4: CELL3

• RADIUS_CELL(cell_name, radius) [AP points] identifies and prints all cells within the closed circle of the specified radius around the specified cell. You are expected to use the PR quadtree efficiently to guide your search process. For full credit, your output should also include the path that you follow during the search, as defined in part 1 for RECTANGLE_CELL. Programs that do not prune the search tree efficiently will be penalized up to half credit. Note that you are not required to find the best such algorithm; merely one that prunes the PR quadtree in some reasonable way. It is acceptable to use your original rectangle function to prune. As before, you must make sure that your tree behaves like that theoretical model- that is, even if you use NULL pointers for children you should still print out the relevant directions, or you will receive a giant 0 points. For instance on the following tree:

NW
NE Foo at (5.000,6.000)
SW Bar at (1023.000,1023.000)
SE

the output of RADIUS_CELL(Foo,1000,1000) should be

NW
NE Foo at (5.000,6.000)
SW
SE

even though NW and SE may actually be NULL pointers. Please don’t mess this up, I’ll have no sympathy if you do.

item CREATE_PSEUDO.Cells( x_coord, x_range, y_coord, y_range) [AP points] Ahh pseudo cells. Perhaps it is best to explain why these exist. 1. To get you to implement line intersection 2. To get you do do subdivision in your quadtree beyond the unit level.

These are warm-ups for project three, and not directly related to p2. That said, read on...

Identify pseudocells within rectangle determined by x_coord plus or minus x_range, and y_coord plus or minus y_range. Next, name the pseudo sites " +PSxxx" where xxx is the number of pseudo cells
that have already been created +1, i.e, the first pseudo cell should be called +PS001, the second pseudo cell should be +PS002 etc. Of course, I am limited in my ability to force which order your cells are added in, so outputs may not necessarily match. The "+" will ensure that the user cannot enter such a name as a Cell name ahead of time, since + is not a valid character for the CREATE_CELL() command. Then, insert the site information into the B+tree and insert the site into the quadtree. If while inserting into the quadtree you find that for some point p already in the tree, \(-p.x.x--;0001\) and \(-p.y.y--;0001\) you must disregard the point. It should then have no effect on the PRTree or the BPTree or other values. Contrary to what I(brian) posted, Pseudo cells have absolutely no effect on the EBOT_NET. PRINT_EBOT_NET should not display Pseudo Cells at all. The avoid a slew of error checking and make our lives easier, after CREATE_PSEUDO CELLS is called for the first time, INSERT_CELL() *will not be called again* unless CLEAR_ALL() is called first. Any other function may still be called, including another call to CREATE_PSEUDO CELLS().

As this command is processed you should print out the pseudo cells you create. See the sample i/o for proper formatting, but essentially it will look like either:

```
*****
  => CREATE_PSEUDO CELLS(...)
New pseudo cells:
+PS001 at (......)
.
.
.or

*****
  => CREATE_PSEUDO CELLS(...)
No new pseudo cells discovered in the specified region.
```

- **DRAW_QTREE_MAP()** [MAP points] Pending. If this isn’t updated by friday night you are not responsible. Check back.

Original description to give you an idea (you will definately be drawing in part3, so it won’t hurt to play with even if this isn’t updated):

Draws the quadtree partitions, as well as cells in the PR tree and Ebot net edges included in the quadtree on an X-terminal screen. If this instruction appears after execution of a CREATE_PSEUDO CELLS() command, then the pseudo cells should be included in the drawing. Similarly, the entire ebot net should be shown if more than one cell has been inserted in the map.

Note that you will need either the C++ utilities on our 420 webpage, or SWING in Java. Please note that this ability will be very helpful for part 3. And, this function description will need to be updated, because the TA’s have to decide how to grade this piece.

### 1.2.2 Part 2 BNF

Please post queries, corrections, and kudos to the newsgroup (csd.cmsc420).

**PLEASE LOOK TO THE SAMPLES FOR CORRECT FORMATTING.** If there is a descrepancy between the two, go with the sample data.

Notation: If the symbol on the left-hand-side is `<XXX-cmd>`, that means that upon reading the XXX command from standard input you will write the right-hand side of the rule to standard output. The command names are a bit shortened-- their full form is as given in section...-- and you are should echo them command to output with a newline afterwards. Note that there may be extra requirements for the output (e.g.,
alphabetical order by cell name) which are not shown here, but appear elsewhere in the specification, or will be clarified on the newsgroup. For example, some parts of the spec ask you to print a list of cell names; for these,

\[
<cell> := <cell-name>
\]

Other parts ask you to print the cell with its coordinates, giving

\[
<cell> := <cell-name> at (<int>, <int>)
\]

And, the pr quadtree search path (in radius_cell) demands only the coordinates:

\[
<cell> := (<int>, <int>)
\]

So, pay attention. You should know the specification inside and out by the end of the semester or, at least, "I" will.

\[
<clear-cmd> := <clear-cmd>
\]

\[
All \ structures \ cleared.<nl>
\]

\[
<create-cmd> := <create-cmd>
\]

\[
Created \ cell \ <cell-name>.
\]

\[
| <create-cmd>
\]

\[
Error: \ Cell \ <cell-name> \ already \ exists.<nl>
\]

\[
<list-cmd> := <list-cmd>
\]

\[
<cell-list>
\]

\[
<cell-list> := <cell><nl>
\]

\[
<some-cells>
\]

\[
End \ of \ list.<nl>
\]

\[
| None.<nl>
\]

\[
End \ of \ list. <nl>
\]

\[
<some-cells> := <cell><nl><some-cells> |
\]

\[
<print-bptree-cmd>:=<print-bptree-cmd>
\]

\[
bptree:<nl>
\]

\[
None.<nl>
\]

\[
End \ of \ list.<nl>
\]

\[
| <print-skip-cmd>
\]

\[
bptree:<nl>
\]

\[
bptree-nodes
\]

\[
End \ of \ list.<nl>
\]

\[
bptree-nodes>:=<bptree-node><nl><bptree-nodes> |<bptree-node><nl>
\]

\[
<more \ bptree \ stuff \ to \ appear. \ don't \ believe \ this>
\]
<insert-cmd> ::= <insert-cmd>
Cell <cell-name> has been inserted.<nl>
  | <insert-cmd>
Cell <cell-name> inserted and connected to <nl>
cell <cell-name> (cost: <float>).  
  | <insert-cmd>
Error: Cell <cell-name> is not the name of a created cell.<nl>
  | <insert-cmd>
Error: Cell <cell-name> is already in the PR quadtree.<nl>

<print-kdtree-cmd> ::= <print-kdtree-cmd>
  Inserted cells:<nl>
  <cell-list>

<print-net-cmd> ::= <print-net-cmd>
  Search path:<nl>
  <cell-list>
  Cells in net with cost <float>:  
  <adjacency-list>

<adjacency-list> ::= <adjacency><nl>
  <some-adjacencies>
  End of list.<nl>
  | None.<nl>
  End of list.<nl>

<some-adjacencies> ::= <adjacency><nl><some-adjacencies> |

<adjacency> ::= <cell-name> <a strcmp order cell names delimited by
ONE SPACE and NO NEWLINES>

<radius-cmd> ::= <radius-cmd>
  Search path:<nl>
  <cell-list>
  Cells within sphere:<nl>
  <cell-list>
1.3 Part 3: Developing the Local Hanan-o-bot Web Due: April 20-22, 02002,
Max. points: 150

The Sametist University equivalent of classrooms, offices, dorms, athletic fields, libraries, and the passages
connection them are constructed by MegaBeast colonies that communicate by extruding and entwining their
many, many tendrils. Throughout their life cycle, MegaBeast colonies depend on an intricate web composed
of the sensory structures of literally millions of H’bots of all sizes, shapes, and function. This part will extend
the B+ tree developed previously, to indicate different types of H’bots in different cells by encoding the type
in the cellname.

The positions of the H’bots throughout the planet, and their connective
tendrils will be stored in a point-region (PM1) quadtree. Then, distinguished members of the H’bot
web net will become conjoined with tendrils of sametoplasm. This web net will be constructed by following
the shortest paths between selected MegaBeasts. To make life easier in part 4, you will also be required to
integrate a quad-tree drawing package into your code.

This portion of the project will model the construction of the Hanan-o-bot support web. There are
three classes of Hanan-o-bots involved in the bioengineering of the ideal MegaBeast habitat. Unfortunately,
h’bots of any two different classes tend to combine with nasty results. So, the facilities need to be physically
isolated based on the kinds of Hanan-o-bots being processed. There are three types of Hanan-o-bots, isolated
by lining the each inter-type interfaces with inert materials that prevent interactions among these volatile
materials. The levels are named alpha, gamma, and sigma, as detailed in the command specification below.

Part 3 Overview Specification Modifications

Note: this is more than just what was in the bare bones specification posted on the newsgroup.

1. Data Dictionary work

(a) We are updating “set BPtree order” command, with a command having two arguments: the
bptree order, and the bptree key limit (L). L will dictate the maximum number of keys (pieces
of data) in a leaf node.

(b) **B+ tree search rules:**
No credit unless your B+ tree implements a multiway search trie with “less than (<)” a given
key referring to the values of the appropriate left B+ subtree child, and “greater than or equal
to (≥)” referring to the values of its corresponding right B+ subtree child.

(c) **B+ tree coordinate check.** Duplicate coordinates must be detected regardless of name. The
easiest way to do this is to re-use your pr-quadtree. You could also write a 2-d tree from your bst in
part 1. In either case, we don’t care how you do deletion in the coordinate-checking structure—just
so long as it is correct.

(d) **B+ tree leaf contents:** We are modifying the number of data elements in B+ tree leaves to be
between [ceiling of (L/2), L], where L is given by the SET_BP_TREE_ORDER command. This
means that a leaf node may contain more or less points than internal nodes in the tree. Motivation:
remember that the goal of the B+ tree is to make every node an ideal size (the size of a page of
memory, for instance). Perhaps 20 keys could fit in a single page of memory, but if a physical
copy of the data is stored in the leaf, and the data is twice as large as the key, then maybe only
10 records (actual pieces of data) can be stored per leaf.

(e) **B+ Tree Delete:** Expand B+tree to include deletion. Actual deletion is required and mandatory.
Lazy delete will receive zero credit. Neither “mark and sweep” or “reinsert any/all existing nodes”
is acceptable. To receive and credit for having a working bptree at all, you must implement this
function.
delte_cell(cell_name) is the expected syntax

(f) **Cell type.** Add the ability to recognize/store a ”type” for each cell name with type encoded as
the first few characters of the cellname. You may choose to determine the cell type when you
create the data dictionary and use an extra entry that says specifically what the type is. Or,
you can examine the name every time you need to determine the type of a cell. One way throws
space at it. The other saves space by always checking the name of the cell when the type must
be discerned. The list cells function will be modified to take a type argument, again, essentially
as described in the currently posted version.

(g) Add the ability to do range searches in the data dictionary. You should be able to list all cells with
names between cell
name1 and cell
name2 in strcmp order (endpoints included, neither cell
name1 nor cell
name2 need actually be in the dictionary)

RANGE CELLS(cell
name1, cell
name2) is expected syntax

2. Spatial Data structure—MAP work Recall that the term q-edge refers to the portion of a segment that
is contained in a given partition (black node)

The first difference between the PR and the PM1 is that all boundaries are closed. Thus, a vertex
inserted on the corner of a partition must be inserted in all the partitions sharing that corner point.

When you receive a command INSERT_TENDRIL(cell1, cell2) you insert the segment with endpoints
cell1 and cell2 into the PM1 quadtree.

You are to implement a PM1 quadtree for the map, as presented in class where a black node holds
one of the following: exactly one vertex and any q-edges attached to it, or exactly one q-edge and
no vertices or an isolated point (produced by inserting a tendril with the cell1=cell2). and samet
mentions that this can be done in his book, as well. note that only a certain ”type” of cell can be an
isolated point but, we’ll discuss that later.

When a set resolution command is encountered, the value of the resolution is set to high or low,
depending on the parameter. Remember, by default, the start up is in low resolution, and if high
resolution is not implemented, there will be a “fail safe” response given that will keep your program
from crashing on submit.

Note: Only low resolution will be tested to submit. If you are unable to implement high resolution you
may as well ignore the command altogether, as you will fail the associated tests. Please no infinite
loops.

The default PM1 quadtree mode is “low resolution”, which refers to how the case where one tendril
crosses a tendril already inserted in the PM1 quadtree. In low resolution mode, you return a message
indicating that the tendril could not be inserted, and continue with normal processing. That is, if you
are inserting a tendril (edge) and in doing so discover that it intersects a tendril already inserted present
in the quadtree at a point where no cell exists (not a vertex), you will return a message indicating this
problem and will not insert that edge.

High resolution mode treats edge intersections differently. When an edge being inserted crosses and
existing q-edge, we replace the crossing point with a pseudocell. This will be the ”high resolution”
quadtree that will be worth more than the simpler case. You have to identify the pseudocell, insert
name in data dictionary, and update any other structures involving the formerly two segments that
now are four segments, all of which meet at the pseudo-cell. any stored adjacency lists need to be
update too. The rules for creating pseudocells will be the same as for project 2 (and will be reviewed
below).

Expect to have to print out the PM1 quadtree somehow to indicate what you have stored in each leaf
as well as the tree’s internal structure.

Note that drawing the PM1 will help you debug.

1.3.1 Part 3 Command Specification

As before, each command spans exactly one line; commands will be uppercase, and reasonably sloppy syntax
is to be supported (spaces and empty lines are allowed). For those who don’t guess <nl> in the BNFs below
is ‘newline’, ie ‘\n’
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 then cause deletion of all currently defined cells and routes. Print “All structures cleared.” after execution, even if this is the second command in the data file.

Output summary:
<output>::<success>

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

CLEAR_PM() initializes or re-initializes (ie. empties) a PM1 quadtree data structure. Note that the PM1 quadtree should also be re-initialized if the command CLEAR_ALL is encountered. Print a message to indicate success.

Output summary:
<output>::<success>

<success>::=Quadtree cleared.<nl>

SET_BPTREE_ORDER(btree_order,leaf_order) will indicate the size 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 leaf_order can be as small as 1, btree_order will be at least 3.

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(leaf_order/2) and leaf_order keys, inclusive.

Remember the root is an exception, in that it never has a lower bound on the number of keys it contains. Its upper bound is determined by whether it is a leaf or internal node.

And, as noted in the overview, whenever a value is equal to a key it should go to that key’s RIGHT child. This is mandatory in part 3, meaning no credit will be given for the quadtree if this rule is not observed.

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

<error>::= Internal order set to <btree_order>. Leaf order set to <leaf_order>.<nl>

CREATE_CELL([cell_name,x,y]) adds the name of the cell along with its coordinate position into the data dictionary. The data dictionary should be a $B^+$-tree as described above. Coordinates will be in the range [0, 1024). Print a confirmation message if this command is processed successfully. You may assume that no more than 1,048,576 cells [Generous, eh? Sincerely, Brian] will ever be inserted in the data dictionary.

As before, creating a cell that exists already is an error. However, because the number of Sametists accessing the database has increased, the likelihood of input errors has increased. So, the input to the dictionary must be checked to make sure that (X, Y) coordinates are not replicated. That is, distinct cell names must imply distinct (X, Y) values, and brute force search across the leaves isn’t fast enough. We may test you on large inputs with reasonable CPU time constraints; 10000 comparisons will take significantly longer than log(10000) comparisons, this is easily measurable.
And, you are welcomed to store the coordinates in a 2-d tree or pr quadtree. I don’t care which. We aren’t looking for the absolute fastest method, but CREATE_CELL should, may must, run asymptotically in logarithmic time.

Please note that there are several distinct Hanan-o-bot types: Alpha, Gamma, Sigma, and Mu. The type of Hanan-o-bot will be encoded in the first letter of the cell name. Type Alpha Hanan-o-bots begin with the letters A–F inclusive; type Gamma Hanan-o-bots begin with the letters G–J inclusive; type Sigma cells begin with the letter S–V; and type Mu cells begin with the letters M–R. Type Psi cells are the pseudocells, which begin with ‘+’. Other cell-types might make use of the remaining letters; so, please don’t assume that letters not included are not used. That is, we can use any first letter we want for a cell name; so, you are not required to examine the first letter of the cell name when creating the entry in the data dictionary.

Other cell names can be stored in the dictionary as well as in the PM1 quadtree. So, if you receive a cell name with a first letter other than A–F, G–J, M–R, or S–V, it is of unknown type and is treated as if it is in class Omega. While cells of type Omega should be listed when appropriate, as in LIST.Cells(OMEGA), no other commands will take Omega cellnames as arguments at this time.

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

<success>::=Created cell <cellname> of type <celltype>.<nl>
<celltype>::=ALPHA | GAMMA | SIGMA | MU | PSI | OMEGA

<error>::=<nameErr> | <coordErr>

<nameErr>::=Error: Cell <cellname> already exists.<nl>
<coordErr>::=Error: A cell already exists at these coordinates.<nl>

Since it’s late, I promise the two errors will never occur at the same time (a cell that already exists \*and\* is at the same position as another cell) but for part 4 you may wish to adopt the standard that the nameErr supersedes the coordErr. Not important....

**DELETE_CELL(cell_name)** deletes a cell from the B+ tree dictionary. Your delete must run in logarithmic time, and lazy delete is impossible to use (since your tree will be printed using PRINT_BPTREE() and lazy delete will break rules inside your leaves). Please don’t count on cheating the grading program—it will be updated for part 3 to repair known vulnerabilities. Note that you should also delete the coordinates from whatever structure you are using to facilitate checking for duplicates; however, you may choose your own deletion technique.

If the cell does not exist, print an error message. If the cell you are trying to delete has already been inserted in the PM1 quadtree you should print an error message, since you are not yet required to write PM1 delete. Otherwise print a message to say the cell has been successfully deleted.

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

<success>::=Deleted cell <cellname>.<nl>
<error>::=<DNE> | <AI>
<DNE>::= Error: Cell <cellname> does not exist.<nl>
<AI>::= Error: Cell <cellname> has already been inserted.<nl>
**LIST CELLS** (*cell_type*) lists the cell names of the cell type specified, and their coordinates in increasing alphabetical (strcmp) order of the cell names. The parameter *cell_type* is any one of ALPHA, GAMMA, MU, SIGMA, PSI, or OMEGA. The parameter OMEGA means that all cells, regardless of type, should be listed. This function will be used as a measure of success for the CREATE CELL function.

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

<success>:=<celllist>
<celllist>:=<cell><celllist>|<cell>
<cell>:=<cellname> at <coord> of type <celltype><nl>
<coord>:= (<double-3 digits after decimal>,<double-3 digits after decimal>)
<celltype>:=ALPHA | GAMMA | SIGMA | MU | PSI | OMEGA

<error>:= None.<nl>

Yes, there is some redundancy in having you print the celltype when they should all be the *same* type, but this should be easier since you will only need one common print function. This is also practical for LIST CELLS(OMEGA). Pseudocells should be printed when the PSI specifier is received.

**RANGE CELLS** (*cell_name1*,*cell_name2*) expects you to list all cells with names between *cell_name1* and *cell_name2* in strcmp order (endpoints included, neither *cell_name1* nor *cell_name2* need actually be in the dictionary) List all valid cells regardless of type. If no cells are found, print "No matching cells found."

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

<success>:=<celllist>
<celllist>:=<cell><celllist>|<cell>
<cell>:=<cellname> at <coord> of type <celltype><nl>
<coord>:= (<double-3 digits after decimal>,<double-3 digits after decimal>)

<error>:= No matching cells 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 trivial, BFS will be a little 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 (3,2) is printed below. Note that it is not a valid order (3,3) tree, since leaves would need at least 2 keys each.

{(*(bar))
{(*(CELL3),(foo))
{(*(CELL1),CELL2),(CELL3),(bar),(foo)}

Note the leaf CELL3 is to the RIGHT of the key CELL3.)

Even at the leaves print only the key (the cellname). 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.

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

17
SET_RESOLUTION(mode). THE REQUIREMENT FOR THIS FUNCTION HAS BEEN POSTPONED TILL PART FOUR. The main reason is that the most practical way to implement high res is to use PM1 delete, which you are not supposed to be required to write yet. As a reward to those who do implement high res, a small amount of extra credit will be given. The test will be very basic to get this extra credit (no more than two segments will intersect at the same time). I am not changing the description of insertTendril, but realize that you must always be in low-res mode :) Description: mode will be either "HIGH" or "LOW" and you do not need to check for invalid modes. Your program should begin in LOW mode by default. Print a message to indicate success "Resolution set to <mode> mode". For more information on the use of mode, see the overview and INSERT_TENDRIL below.

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

<success> ::= Resolution set to <mode> mode.<nl>

INSERT_TENDRIL(cell_name1, cell_name2) inserts an edge with endpoints cell_name1 and cell_name2 into the PM1 quadtree which has its quadrants closed on all sides. Both the specified cells should be valid cell names. These routes make up the M\'LAC and the MegaBeast cell population.

Cells to be inserted should have been created earlier using CREATE_CELL command. If either cell is not present in the data dictionary, output an error message. If the [ed: TENDRIL] exists in the quadtree already, output an error message. Print confirmation message if the command is processed successfully.

If the command is called for an existing route, output an error message. When this function is called with GAS cells (type GAMMA, ALPHA, or SIGMA), the cell names must be distinct. When called for MegaBeast cells (type MU), you MUST do a special error condition check. MegaBeast cells are isolated cells and hence cannot have routes to other cells. If one cell is a MegaBeast cell, then the other cell must match the original cell (i.e. cell_name1 = cell_name2); otherwise print an error message. This means that MegaBeast cells, corresponding to isolated points, are inserted in the PM1 quadtree as a route with zero length. Also, this function cannot be called with pseudocells as arguments. That is, no cell type Psi can be provided as a vertex of an edge in the test data.

If the cells provided as arguments have not been inserted in the PM1 quadtree, insert them as well as the route. Print a message to indicate success. PRINT_PMTREE() will be used to measure the success of this operation when splits are encountered.

A note on errors—If in high resolution mode (described below) a route/tendril is split, no attempt will be made when grading to re-insert the same route. This is somewhat artificial, but may simplify things a little bit.

How to handle intersection: While inserting edges into the PM1 Quadtree, there exists a possibility that two edges intersect at some point other than endpoints; if this happens, normally the recursive
quadtree subdivision will loop infinitely (because there is no way to separate routes into different
pixels). You should detect this situation by, controlling the recursion depth or current pixel size, or
by checking for distance/intersection when reaching a leaf node. You [ed: MUST] assume that the
minimum separation of unique objects is 0.00001; any vertices or q-edges closer than that are treated
as identical or intersecting, respectively.

If you are in low-resolution mode and detect an unwanted intersection, print an error message saying
that the edge cannot be inserted and discard any changes made to the quadtree.

If you are in high resolution mode there are two possibilities. First possibility: one or more edges
intersect the new edge at exactly one point each. In this case you must create a new pseudo cell at
each point of intersection and update all the edges that are involved. For instance if there is a tendril
from A to B, and you add an intersecting tendril from C to D, you will create a new pseudo cell P and
change all (A,B) edges to (A,P) and (P,B) edges, and add edges (C,P) and (P,D). This may involve
creating edges from one new pseudo cell to another (If a new tendril intersects two existing tendrils, for
instance). If you write delete from the PM1 in this project you can use it to make this easier. There
are other ways to do the update, however. You will have to write PM1 delete for project 4, so if you
choose this path your time will not be wasted.

Second possibility: One tendril overlaps an existing tendril at more than one point (the tendrils are
collinear). Your TA promises this case will not occur in project 3. It may show up in project 4 however.
Pseudo cells should follow the naming scheme +PSxxx where xxx is a unique number of your choosing.

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

<success>:= Tendril inserted.<nl>

<error>:= <INVALID>|<AE>|<MU>|<NW>

<INVALID>:= Error: <whichCell> is not the name of a created cell.<nl>
<whichCell>:= <cell_name1> | <cell_name2>

<AE>:= Error: This tendril already exists.<nl>

<MU>:= Error: Nu cell mismatch.<nl>

A note on INVALID: the first cell that does not exist is the only one which should be printed. The
<MU> error occurs when one cell is of type MU(A megabeast cell) and cell_name1 != cell_name2 (see
CREATE_CELL).

LIST_TENDRILS() [SS] points] lists all existing edges in the PM1 quadtree along with their lengths.
The routes should be listed in ASCII order in the adjacency list format of PRINT_BOT_NET in
part2, with an extra term for length (described below). Megabeast cells should not be listed at all.
Print a message if no matching tendrils are found.

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

<success>:= <adj-list>
<adj-list>:=<adj-row><adj-list>|<adj-row>
<adj-row>:=<cellname>:<adjEdges><nl>
<adjEdges>:=<adjEdge><adjEdges> | <adjEdge>
<adjEdge>:=<cellname>({double-3digits after decimal})

<error>:= No matching tendrils found.<nl>
PRINT_PMTREE() prints the (preorder) PM1 in a format similar to that of projects 1 and 2 PRINT_PRTREE. If [ed: AN ISOLATED Q EDGE] is encountered print "Edge: |cell1| |cell2|" where cell1 and cell2 are the endpoints of the edge. cell1 and cell2 must be printed in ascibetical(strcmp) order. Notice I have removed the word 'EMPTY' from the BNF as well as "inserted cells:" and there is no list of cells at the end. Your output should match mine exactly on this function, btw.

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

<success>:= <pmtree><nl>
<pmtree>:=<black_node>|<white_node>| NW <pmtree> NE <pmtree> SW <pmtree> SE <pmtree>
<black_node>:=<cell>|<isolated_q_edge>
<cell>:=<cellname> at <coord> of type <celltype><nl>
<isolated_q_edge>:= Edge: <cellname> <cellname><nl>
<white_node>:= <nl>

<error>:= No matching tendrils found.<nl>

For isolated q-edges, print the endpoints in increasing ascibetical order(i.e. Edge: A B).

NEAREST_CELL(cell_name) finds the nearest GAS cell to the given MegaBeast cell in terms of Euclidean distance(must be in PM1 Map!). Print an error message if cell_name is not a type Mu cell. If successful, output should include the GAS cell name and its type. If two or more GAS cells are equidistant from the given MegaBeast cell, chose the one that is earlier in strcmp order. Your search should be efficient. As a measure of efficiency you should print out the paths that you are following in your PM1 quadtree (as you did in Part 1). Programs that do brute-force search through the entire tree will be penalized.

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

<searchpath>:=Search Path:<nl><searchStuff>
<searchStuff>:=See defn of print_pmtree, print only directions your algorithm actually goes. Treat q-edges as white notes for printing output.

<success>:=<nearest GAS cell><cell>. <nl>Distance: <double- 3 digits after decimal>. 
<cell>:=<cellname> at <coord> of type <celltype><nl>

<error>:= <searchpath><noCells>|<DNE>|<NM>
<noCells>:=Error: No GAS cells found.<nl>
<DNE>:= Error: Cell <cellname> does not exist.<nl>
<NM>:= Error: Cell <cellname> is not a Megabeast cell.<nl>

Note that the notation says the search path is printed only for the |noCells| error, not the other errors (it is also printed on success).

RADIUS CELLS(cell_name, radius) [AP points] identifies and prints all GAS cells within the specified radius of a given cell. You are expected to use the PM1 quadtree efficiently to guide your search process; since this function will be worth very little at this point I will take your word for it. The cellname given must be in the dictionary but need not be in the PM1. All cells printed in the result must be in the PM1 map.

Output summary:
<output>:=<success>|<error>
<success>:= Cells within radius:<nl><celllist>
<celllist>:=<cell><celllist>|<cell>
<cell>:=<cellname> at <coord> of type <celltype><nl>
<coord>:= (3 digits after decimal, <double- 3 digits after decimal>)
<celltype>:=ALPHA | GAMMA | SIGMA | MU | OMEGA

<error>:= <noCells> |<DNE>
<noCells>:=No cells within the specified region.<nl>
<DNE>:= Error: Cell <cellname> does not exist.<nl>

PRINT_MST() prints all the edges of the 'maximally connected minimum spanning tree' (I just made that up, Brian) of all inserted tendrils (excluding Megabeast/Mu cells). A definition of this structure would be "The minimum weight graph such that for all cells A,B- if there was a path from A to B in the original graph, then there is also a path from A to B in the MAXMST". The weight of an edge is just its length. MST is an abuse of the term here, but since it has been in the spec for a while I am not changing the function name.

I said before: By maximally connected I mean that if cell A can reach cell B via the tendrils in the PM1, then A must also be able to reach B via the tendrils in the MST. Output should be in the format of LIST_TENDRILS, with only edges in the MST listed. Also print the total cost of the MAX MST. The only error condition is that no tendrils exist.

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

<success>:= <adj-list><nl>Total length of MST is <double- 3digits after decimal>.<nl>
<adj-list>:=<adj-row><adj-list>|<adj-row>
<adj-row>:=<cellname>:: adjEdges<nl>
<adjEdges>:= <adjEdge> <adjEdges> | <adjEdge>
<adjEdge>:= <cellname><double- 3digits after decimal>)

<error>:= Error: No tendrils have been inserted.<nl>

DRAW_QTREE_MAP(outfile, scale) (AP points) Draws the quadtree partitions, as well as cells and edges included in the quadtree on an X-terminal screen using the resources on the class webpage. If you are using the showquad package then outfile should be the name of your output file to be used with showquad (we will run showquad after your program finishes). If you draw the map while the program is running (using the java canvas class, for instance) just ignore the outfile parameter. The meaning of scale will appear shortly.

Edit: The scale factor provided will always be 1. Please use common sense in rendering your picture to a size that fits on an average size screen. 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).

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

<success>:= Drawing complete.<nl>
<error>:= Not implemented.<nl>
1.4 Part 4: Developing the MegaBeast Local Area Cellnet M’LAC

Once the support nets have infested the planet and produced the nested concentric spheres of biomechanical nutrient webs needed to support developing MegaBeast cells, the actual structures used to educate fledgling Sametists can be constructed. Immature MegaBeasts are implanted in otherwise isolated cells to produce the Sametist equivalents of classrooms, administration buildings, office suites, athletic facilities and laboratories. Cells of types alpha, gamma and sigma will be connected to produce the backbone of the MegaBeast local area cellnet (M’LAC), which is then entwined with tendris extruded by the maturing MegaBeast cells, thus producing the M’LAC. The M’LAC permits the MegaBeasts to communicate with each other and with the Hanan-o-bot protobrain already installed at the planet’s core.

After inserting both MegaBeast cells and M’LAC edges in the quadtree, you will determine the closest tendris and the closest gamma, alpha, or sigma (GAS) cells to the MegaBeast. And, you will identify all 3-sided and 4-sided polygons having at least two vertices within a specified radius of each MegaBeast cell. The vertices of the polygon are cells of type alpha, gamma or sigma, and the edges of the polygons are existing routes between such cells.

1.4.1 Part 4 Command Specification

In addition to new commands, part four will support all commands from project3. The old commands are relisted here, with any changes/fixes that need to be made.

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

New Commands:

DELETE_TENDRIL(cell_name1, cell_name2) Deletes the tendril specified. Any pseudo cells on the tendril should be removed from the PM1 and from the B+tree, and respective intersected edges should be restored. For simplicity no two tendris will intersect at the same pseudo-cell. After the tendril is deleted the PM1 should look like it would if that tendril had never been inserted.

Note that deletion will not be checked using pseudocells as arguments.

This function will be tested in both low resolution mode and high resolution mode to avoid double jeopardy.

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

<success> ::= Tendril deleted.<nl>
<error> ::= <INVALID> | <NE>
<INVALID> ::= Error: <whichCell> is not the name of a created cell.<nl>
<wcell> ::= "cell_name1" | "cell_name2"
<NE> ::= Error: The specified tendril does not exist.<nl>

NEAREST_TENDRIL(cell_name) Finds the non-MU tendril closest to the MU cell specified. you may assume that the cell given is a MU-Cell that already exists in the B+tree. It does not need to be in the PM1. You may also assume that there are tendris in the PM1. So there are no error messages!

Output summary:
<output> ::= <success>
<success> ::= Tendril <cellname1> <cellname2>. Distance: <double- 3 digits after decimal>.<nl>

cellname1 and cellname2 are the endpoints of the tendril. cellname1 | cellname2. The distance is the distance to the nearest point of the tendril.
SHORTEST_PATH(cell_name1, cell_name2) Prints the shortest path from cellname1 to cellname2. You may assume that cellname1 and cellname2 are in the PM1 and that neither are MU cells. The only error is if no path exists between the two cells.

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

<success> := <cellname1> -> <moreCells> <nl>Total length: <double 3 digits after decimal>. <nl> <moreCells> := <cellname> -> <moreCells> | <cellname2>
<error> := Error: No path exists.

Modified Commands:

CREATE_CELL([cell_name, x, y]) adds the name of the cell along with its coordinate position into the data dictionary. The data dictionary should be a B+ tree as described in project3. Coordinates will be in the range [0, 1024]. Unlike in project3, you will no longer double-check that there are no duplicate coordinates. This function must run in O(log n) time.

Please note that there are several distinct Hanan-o-bot types: Alpha, Gamma, Sigma, and Mu. The type of Hanan-o-bot will be encoded in the first letter of the cell name. Type Alpha Hanan-o-bots begin with the letters A–F inclusive; type Gamma Hanan-o-bots begin with the letters G–J inclusive; type Sigma cells begin with the letter S–V; and, type Mu cells begin with the letters M–R. Type Psi cells are the pseudocells, which begin with ‘+’. Other cell-types *might* make use of the remaining letters; so please don’t assume that letters not included are not used. That is, we can use any first letter we want for a cell name; so, you are not required to examine the first letter of the cell name when creating the entry in the data dictionary.

Other cell names can be stored in the dictionary as well as in the PM1 quadtree. So, if you receive a cell name with a first letter other than A–F, G–J, M–R, or S–V, it is of unknown type and is treated as if it is in class Omega. While cells of type Omega should be listed when appropriate, as in LIST CELLS(OMEGA), no other commands will take Omega cellnames as arguments at this time.

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

<success> := Created <cell>, <nl>
<cell> := <cellname> at <coord> of type <celltype>
<coord> := (<double 3 digits after decimal>, <double 3 digits after decimal>)
<celltype> := ALPHA | GAMMA | SIGMA | MU | PSI | OMEGA
<error> := <nameErr>

<nameErr> := Error: Cell <cellname> already exists.<nl>

DELETE_CELL(cell_name) deletes a cell from the B+ tree dictionary. If the cell is already in the PM1, then you must delete all tendrils coming from that cell in the PM1 according to the rules of delete_tendril().

If the cell does not exist, print an error message. Otherwise print a message to say the cell has been successfully deleted.

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

<success> := Deleted cell <cellname>.
<error> := DNE
<DNE> := Error: Cell <cellname> does not exist.<nl>
**RANGE CELLS**(*cell_name1, cell_name2*) expects you to list all cells with names between *cell_name1* and *cell_name2*. If *cell_name1* < *cell_name2* the cells must be listed in increasing strcmp order (endpoints included, neither *cell_name1* nor *cell_name2* need actually be in the dictionary). If *cell_name1* > *cell_name2* the cells must be listed in reverse strcmp order.

List all valid cells regardless of type. If no cells are found, print "No matching cells found."

**Output summary:**
<output>:=<success>|<error>
<success>:=<cellist>
<cellist>:=<cell><cellist>|<cell>
<cell>:=<cellname> at <coord> of type <celltype><nl>
<coord>:= (double-3 digits after decimal),<double-3 digits after decimal>)
<error>:= No matching cells found.<nl>

**SET_RESOLUTION**(mode). Mode will be either "HIGH" or "LOW" and you do not need to check for invalid modes. Your program should begin in LOW mode by default. Print a message to indicate success "Resolution set to <mode> mode". For more information on the use of mode, see the overview and **INSERT_TENDRIL** below.

**Output summary:**
<output>:=<success>
<success>:= Resolution set to <mode> mode.<nl>

**INSERT_TENDRIL**(*cell_name1, cell_name2*) inserts an edge with endpoints *cell_name1* and *cell_name2* into the PM1 quadtree which has its quadrants closed on all sides. Both the specified cells should be valid cell names. These routes make up the **M'LAG** and the MegaBeast cell population.

Cells to be inserted should have been created earlier using CREATE_CELL command. If either cell is not present in the data dictionary, output an error message. If the tendrill exists in the quadtree already, output an error message. Print confirmation message if the command is processed successfully.

If the command is called for an existing route, output an error message. When this function is called with GAS cells (type GAMMA, ALPHALPHA, or SIGMA), the cell names must be distinct. When called for MegaBeast cells (type MU), you must do a special error condition check. MegaBeast cells are isolated cells and hence cannot have routes to other cells. If one cell is a MegaBeast cell, then the other cell must match the original cell (ie. *cell_name1* = *cell_name2*); otherwise print an error message. This means that MegaBeast cells, corresponding to isolated points, are inserted in the PM1 quadtree as a route with zero length. Also, this function cannot be called with pseudocells as arguments. That is, no cell type Psi can be provided as a vertex of an edge in the test data.

If the cells provided as arguments have not been inserted in the PM1 quadtree, insert them as well as the route. Print a message to indicate success. **PRINT_PMTREE**() will be used to measure the success of this operation when splits are encountered.

A note on errors—If in high resolution mode (described below) a route/tendril is split, no attempt will be made when grading to re-insert the same route. This is somewhat artificial, but may simplify things a little bit.

**How to handle intersection:** While inserting edges into the PM1 Quadtree, there exists a possibility that two edges intersect at some point other than endpoints; if this happens, normally the recursive quadtree subdivision will loop infinitely (because there is no way to separate routes into different pixels). You should detect this situation by, controlling the recursion depth or current pixel size, or by checking for distance/intersection when reaching a leaf node. You may assume that the minimum
separation of unique objects is 0.00001; any vertices or q-edges closer than that are treated as identical or intersecting, respectively.

If you are in low-resolution mode and detect an unwanted intersection, print an error message saying that the edge cannot be inserted and discard any changes made to the quadtree.

If you are in high resolution mode there are two possibilities. First possibility: one or more edges intersect the new edge at exactly one point each. In this case you must create a new pseudo cell at each point of intersection and update all the edges that are involved. For instance if there is a tendril from A to B, and you add an intersecting tendril from C to D, you will create a new pseudo cell P and change all (A,B) edges to (A,P) and (P,B) edges, and add edges (C,P) and (P,D). This may involve creating edges from one new pseudo cell to another (If a new tendril intersects two existing tendrils, for instance). To simplify grading, if two tendrils intersect without sharing an endpoint, then they will not intersect at an endpoint.

Second possibility: One tendril overlaps an existing tendril at more than one point (the tendrils are co-linear). Your TA promises this case will not occur. Pseudo cells must follow the naming scheme +P$XXX$ where $XXX$ is a unique number of your choosing.

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

<success> := Tendril inserted.<nl>
<error> := <INVALID>|<AE>|<MU>|<NM>
<INVALID> := Error: <whichCell> is not the name of a created cell.<nl>
<whichCell> := <cell_name1> | <cell_name2>
<AE> := Error: This tendril already exists.<nl>
<MU> := Error: Mu cell mismatch.<nl>
<INT> := Error: Intersection detected.<nl>

A note on <INVALID>: the first cell that does not exist is the only one which should be printed. The <MU> error occurs when one cell is of type MU (A megabeast cell) and cell_name1 != cell_name2 (see CREATE_CELL).

Unchanged Commands:

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 then cause deletion of all currently defined cells and routes. Print “All structures cleared.” after execution, even if this is the second command in the data file.

Output summary:
<output> := <success>

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

CLEAR_PM() initializes or re-initializes(i.e. empties) a PM1 quadtree data structure. Note that the PM1 quadtree should also be re-initialized if the command CLEAR_ALL is encountered. Print a message to indicate success.

Output summary:
<output> := <success>

<success> := Quadtree cleared.<nl>
SET_BPTREE_ORDER(btree_order, leaf_order) will indicate the size 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 leaf order can be as small as 1, btree order will be at least 3.

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(leaf_order $/2$) and leaf_order keys, inclusive.

Remember the root is an exception, in that it never has a lower bound on the number of keys it contains. Its upper bound is determined by whether it is a leaf or internal node.

And, as noted in the overview, whenever a value is equal to a key it should go to that key’s RIGHT child. This is mandatory in part 3, meaning no credit will be given for the quadtree if this rule is not observed.

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

<success>:= Internal order set to <btree_order>. Leaf order set to <leaf_order>.<nl>
<error>:=Error: $B^+$ tree already initialized.<nl>

LIST_CELLS(cell_type) lists the cell names of the cell type specified, and their coordinates in increasing alphabetical (strcmp) order of the cellnames. The parameter cell_type is any one of ALPHA, GAMMA, MU, SIGMA, PSI, or OMEGA. The parameter OMEGA means that all cells, regardless of type, should be listed. This function will be used as a measure of success for the CREATE_CELL function.

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

<success>:=<cellist>
<cellist>:=<cell><cellist>|<cell>
<cell>:=<cellname> at <coord> of type <celltype><nl>
<coord>:= (double- 3 digits after decimal),<double- 3 digits after decimal>)
<celltype>:=ALPHA | GAMMA | SIGMA | MU | PSI | OMEGA

<error>:= None.<nl>

Yes, there is some redundancy in having you print the celltype when they should all be the *same* type, but this should be easier since you will only need one common print function. This is also practical for LIST_CELLS(OMEGA). Pseudocells should be printed when the PSI specifier is received.

PRINT_BPTREE() requires you to list the $B^+$ in a breadth first search order. If you used links between internal nodes this will be trivial, BFS will be a little 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 (3,2) is printed below. Note that it is not a valid order (3,3) tree, since leaves would need at least 2 keys each.

{{bar}}
{{CELL3),(foo}}
{{CELL1,CELL2),(CELL3),(bar),(foo}}
Note the leaf CELL3 is to the RIGHT of the key CELL3;)
Even at the leaves print only the key (the cell name). 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.

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

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

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

LIST_TENDRILS() [SS] points] lists all existing edges in the PM1 quadtree along with their lengths.
The routes should be listed in ASCII order in the adjacency list format of PRINT_BOT_NET in
part2, with an extra term for length (described below). Megabeast cells should not be listed at all.
Print a message if no matching tendrils are found.

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

<success>:=<adj-list>
<adj-list>:=<adj-row><adj-list>|<adj-row>
<adj-row>:=<cellname>:<adjEdges><nl>
<adjEdges>:=<adjEdge><adjEdges>|<adjEdge>
<adjEdge>:=<classname>(<double-3digits after decimal>)

<error>:= No matching tendrils found.<nl>

PRINT_PMTREE() prints the (preorder) PM1 in a format similar to that of projects 1 and 2 PRINT_PRTREE.
If [ed: AN ISOLATED Q EDGE] is encountered print "Edge: |cell1|cell2" where cell1 and cell2 are
the endpoints of the edge. cell1 and cell2 must be printed in asciibetical(strcmp) order. Notice I have
removed the word "EMPTY" from the BNF as well as "inserted cells:" and there is no list of cells at
the end. Your output should match mine exactly on this function, btw.

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

<success>:=<pmtree><nl>
Pmtree>:=<black_node>|<white_node>| NW <pmtree> NE <pmtree> SW <pmtree> SE <pmtree>
<black_node>:=<cell>|<isolated_q_edge>
<cell>:=<classname> at <coord> of type <celltype><nl>
<isolated_q_edge>:= Edge: <classname> <classname><nl>
<white_node>:=<nl>

<error>:= No matching tendrils found.<nl>

For isolated q-edges, print the endpoints in increasing ascibetical order(ie: Edge: A B).
2 Instructions and Policies

2.1 General Information

Your project must execute on the OTI's cluster. Otherwise, it will not be graded. Your program will be compiled and executed by the TA on the OTI's cluster, and grades will be assigned on the basis of this execution. Your executable should be named part# [Note: everything in lowercase] in the makefile, where # must be replaced with the appropriate part number ie., 1 for part 1.

Your program must read its input from standard-in (a file directed to standard-in, in c++, for example, so it will run when we type part1 < input.1)). There will be at least one (but maybe more) newline characters before EOF. Test your code accordingly.

Here is an example of a makefile that would create an executable part1 in c++.

all: part1.cc Bptree.cc Bptree.h
     g++ -o part1 -02 part1.cc B+-tree.cc -lm

Here is an example of a makefile for java

all: part4.java Bptree.java Dijkstra.java
     javac *.java

The first line of the makefile should include all the files involved in your project; the second line is the compilation command itself. There should be TAB symbol after all: in the first line and at the beginning of the second line. To compile your program simply type “make”.

2.2 Test Data

Test data and sample outputs will be posted on the class webpage. The posted format must be followed, as we will be using semi-automatic grading. No attempt is made to generate all potential test cases, and you should not assume that a project that produces outputs matching the posted test data is correct. For example, unless otherwise stated, the test data will never reflect any optimizations or attempts at efficiency. Brute force will be the rule.

Your projects will be graded using private data files that will be released after the projects are graded to permit you to repair your code for the next part of the project.

2.3 Submission Instructions

You are required to submit your work electronically using the submit command. Follow the procedure outlined below to submit your project. Note that # should be replaced with the appropriate part number (ie., for part 1, # should be replaced with 1).

1. mkdir part#
2. Copy all the source code (.c, .cc, .java, makefile, .h) into this directory (part#). Do not copy executable or object files.
3. cd part#
4. tar -cvf part#.tar *
5. gzip part#.tar
6. `mh420001/lin/submit # part#.tar.gz`

28
If you get any errors, report them to your TA immediately.

*The instructor reserves the right to fail any student who does not make a good faith effort on all parts of the project.*

Please note that a project submitted later would overwrite the one submitted earlier. Therefore only the last submitted version will be graded. Project related questions: post your question to the class newsgroup csd.cmsg420 or visit the tas during office hours.

### 2.4 Late Policy

The late policy is that posted on the newsgroup, and below. Note that due dates are posted here for all parts, not just part 3.

**Part 4:** Early with 10% bonus: before 10am Saturday, May 11th.
On Time: no later than 10pm Saturday, May 11th.
Late with 25% penalty: no later than 10pm Sunday, May 12th.
Late with 50% penalty: no later than 10am Monday, May 13th.

**Part 3:**
Early with 10% bonus: before 10am Saturday, April 20th.
On Time: no later than 10pm Saturday, April 20th
Late with 25% penalty: no later than 10pm Sunday, April 21st
Late with 50% penalty: no later than 10am Monday, April 22nd

**Part 2:** Early with 10% bonus: before 10am Saturday, March 16th.
On Time: no later than 10pm Saturday, March 16th
Late with 25% penalty: no later than 10pm Sunday, March 17th
Late with 50% penalty: no later than 10am Monday, March 18th

**Part 1:** Early with 10% bonus: before 10am Saturday, February 16th.
On Time: no later than 10pm Saturday, February 16th
Late with 25% penalty: no later than 10pm Sunday, February 17th
Late with 50% penalty: no later than 10am Monday, February 18th

### 2.5 Grading Policy: NEW

A new grading policy will be in place for parts 2-4. Points will be given for minimal conformance to the specification, meaning, at best, nominal or baseline performance has been achieved. Then, points will be awarded for how well you satisfy the letter of the specification. That is, do you meet the constraints given explicitly in the specification. Finally, at the discretion of the grader, points will be given for work that is outside of the specification. Note that it’s not always good to exceed the specification; unless you know that the functionality is required for a later part of the project, it might be wasted effort. So, don’t try to do overdesign here, especially if the bell or whistle you consider adding cannot be demonstrated to be useful in the context of the project. That is, a perfect grade for parts 2 and 3 can be had even if you receive zero OVER design points. In part 4, that will change. To wit, you will be required to include some feature of your own choosing that exceeds the letter of the specification. More details later on this.

To give you an idea of relative importance of each function in the project, we also include point values for each of the major facets: documentation and specification conformance (DOC points); data dictionary (DD points); map points (MAP points); and application points (AP points).
2.5.1 Part 2 Point Distribution

**BASE:** TBD Points given for minimal or baseline functionality.

**SAT:** TBD Points, given for code satisfying the-letter-of the specification, no more no less.

**OVER:** Exceptional points, awarded at the discretion of the TA, for code that exceeds the specification.

Within each class, a certain percentage will be applied to each facet of the project’s principal data structure.

**DOC points:** TBD points for documentation and conformance to the specification and the rules.

**DD points:** TBD points associated with maintaining the data dictionary.

**MAP points:** TBD points for establishing and drawing the map corresponding to the spatial data structures.

**AP points:** Points associated with whatever applications have been chosen for the structures in this part of the project.

2.6 Integrity Policy

Your work is expected to be your own or to be labelled with its source, whether book or human or webpage. 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. For part 3, a working B+-tree and PR quadtree will be provided upon request. For part 4, a working PR quadtree will be provided upon request.

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.