CMSC 420 Term Project–Spring, 2002
Version 2.3

Constructing Hanan University Too (HUT) on Planet Temas-Too*

Last Modified March 9, 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 membranous 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\textsuperscript{3} 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

\textsuperscript{3}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.
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 " == " and an echo of the command given. For instance, the entire valid output to CLEAR_ALL() is

```plaintext
***** == 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 original 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 ascibiletical 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:

```plaintext
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 ascii-jetical order in the format
<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
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\((c_x, c_y, x_h, y_h)\) [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:

\[
\{(c_x + x_h, c_y), (c_x + x_h, c_y - y_h), (c_x - x_h, c_y - y_h), (c_x - x_h, c_y)\}
\]

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 the in the same format as in **PRINT\_PRTREE**, except for the following:

1. When you reach 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 not 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 created the Nutrient Web, the series of concentric shells of living, biometric webbing that are the unmistakable sign of a Sametist 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 you 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 part2 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 inclosed 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 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 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

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

where \(r_A = \sqrt{x_A^2 + y_A^2}\) and \(r_B\) 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 tucked 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.

*Were 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 callname \( D \) 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.)

- Printing `PRINT_PRTREE()` [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  CELL3
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.

• 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, ie, 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 PRITree or the BPTree or any 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 io for proper formatting, but essentially it will look like either:

```
*****
==> CREATE_PSEUDO CELLS(...)
New pseudo cells:
+PS001 at (..., ...)
...
*****
==> 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.cmsg420).

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 1.2.1— 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>
End of list.<nl>
    | None.<nl>
    End of list. <nl>
```

```<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—DRAFT—Don’t TRUST

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 skiplist developed previously to indicate different types of H’bots in different cells by encoding the type in the cell name. The positions of the H’bots throughout the planet will be stored in a point-region (PR) quadtree. The members of the H’bot web net to conjoined by tendrils of semetoplasm and implemented MegaBeast will be identified based on 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 at different levels in the planet’s crust, with each inter-type interface lined with inert materials that insulate these volatile materials. The levels are named alpha, gamma, and sigma, as detailed in the command specification below.

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). Documentation and conformance to "the rules" is worth 10 points.

- **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.

- **CLEAR_ALL() [DOC 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. Print "All structures cleared." after execution, even if this is the second command in the data file.

- **CREATE_CELL(cell_name, r, x, y) [DD points]** adds the name of the cell along with its distance from the planet’s surface and coordinate position into the data dictionary. The data dictionary should be implemented as a skiplist with keys in the natural ASCII order (ie. `strcmp()` order) of the cell names. Cell names will be composed of 6 characters that are α or alphanumeric. Coordinates will be in the range [0, 1024]. Print a confirmation message if this command is processed successfully. You may assume that no more that 1024 cells 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 skip-list 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.

You should print a list of the cell coordinates examined in attempting to insert a new cell name to demonstrate that you did not use brute force to make this check, as described in section 1.3.2. And, you are welcomed to store the coordinates in a 2- or 3-dimensional k-d tree. I don’t care which. We aren’t looking for the fastest method, merely NOT brute force.

Please note that there are several distinct Hanan-o-bot types: Alpha, Gamma, and Sigma. 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. Other cell-types might make use of the remaining letters; so, please don’t assume that no letter higher than N in `strcmp` will be 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.
• LIST.Cells\(cell.type\) [DD points] Lists the cell names of the Hanan-\(\alpha\)-bot type specified, and their coordinates in increasing alphabetical (strcmp) order of the cellnames. The parameter \(cell.type\) is any one of ALPHA, GAMMA, SIGMA, 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.

• PRINT_SKIPLIST() [DD points] Lists the skiplist as shown in the BNF, giving the cell name, followed by the number of links in its tower. This function will be used as a measure of success for having constructed a skip list.

• INSERT_CELL\(cell.name\) [SS points] inserts the specified cell into the point-region quadtree at positions determined by the \((X,Y)\) coordinates of the cell as stored in the data dictionary.

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

Please note that the coordinate system in use for the project has \([0,0]\) at the lower left corner and \((1024,1024)\) at the upper right corner. Order of the subtrees should be: 1 (NE), 2 (NW), 3 (SW), 4 (SE). Traversals of the PR tree should list the path through the PR tree by number.

• PRINT_QTREE() [SS points] should be a preorder traversal of the point-region quadtree that lists the names of all cells that have been inserted in the tree.

• RECTANGLE.Cells\((x_{\text{coord}}, y_{\text{coord}}, x_{\text{range}}, y_{\text{range}})\) [U points] identifies and prints all cells present in the point region quadtree within the specified closed rectangle. Cells should be printed in increasing strcmp order. You are expected to use the point-region quad tree efficiently to guide your search process. For full credit, you need to print the coordinates of any leaves (black nodes) you examine in the process. Programs that do not prune the search tree efficiently will be penalized up to half credit for this function.

• DRAW_QTREE_MAP() [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. Two forms are acceptable: DRAW_QTREE_MAP(outfile, scale), and DRAW_QTREE_MAP(). We will check for both when grading.

• MAP_FEASIBLE_NET\(cell1,cell2\) [AP points] Creates an edge between distinct cells, cell1 and cell2, in the support network to facilitate the spread of Hanan-\(\alpha\)-bots, where cell1 and cell2 must be of the same type. Furthermore, both cells must have been inserted into the quadtree already. If they are not the same type, or are not both in the quadtree, then do not insert the edge, and print an appropriate error message.

• PRINT_FEASIBLE_NET\(type\) [AP points] Lists feasible paths in adjacency list format.

• BUILD_SUPPORT_NET\(type\) [AP points] Generates an MST from all cells of type either ALPHA, GAMMA, or SIGMA. Please note that if the graph of cells of a give type is not connected, you will need to restart the MST algorithm. The edges making up the MST for each type will be inserted in the quadtree.

That is, the quadtree nodes containing the vertices of each segment of the path should include information that permits the presence of an edge intersecting that partition to be recognized quickly, as well as allowing several edges emanating from the same vertex to be distinguished. The edge cost is given by the Euclidean distance between the vertices.

• PRINT_SUPPORT_NET\(type\) [AP points] Prints minimum spanning tree (MST) cost and net in the adjacency list format, where \(type\) is one of: ALPHA, GAMMA, SIGMA. Any other parameter should be identified with an error message because of the BAD ARGUMENT.
Special Note Regarding the Last Four Functions.

Since modifying the feasible net could make a previously extracted MST incorrect, we adopt the following rules for the test data set.

First and foremost, edges between vertices of the same type cannot intersect. This ensures that the edges used to construct the support net (MST) will not intersect either. You need not check for that condition.

Second, once an MST has been constructed (by executing `BUILD_SUPPORT_NET`), no more edges will be added to the feasible net, and no more MST’s will be requested unless a `CLEAR_ALL()` occurs to reset the data structure. You need not test for this. We must guarantee it in the data we give you.

1.3.2 Part 3 BNF

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

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 symbol `<nl>` means newline, or the friendly “backslash n.” The command names are a bit shortened— their full form is as given in section 1.3.1— 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 (cell-coord)
```

and, the k-d tree search paths (in `radius_cell`, `sector_cell`, and similar functions) demand only the coordinates:

```
<cell> ::= <cell-coord>
```

where

```
<cell-coord> ::= (<int>, <int>)
```

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

```
<clear-cmd> ::= <clear-cmd>
     All structures cleared.<nl>
```

```
<create-cmd> ::= <create-cmd>
Cell coordinates searched:<nl>
<cell-list>
Created cell <cell-name>.<nl>
| <create-cmd>
Error: Cell <cell-name> already exists.<nl>
| <create-cmd>
Cell coordinates searched:<nl>
<cell-list>
Error: Cell <cell-name> duplicates coordinates.<nl>
```
<list-cmd> := <list-cmd>
    Cells of type <cell-type>:<nl>
    <cell-list>

<cell-list> := <some-cells>
    End of list.<nl>
    | None.<nl>
    | End of list.<nl>

<some-cells> := <cell><nl><some-cells> | <cell><nl>

<print-skip-cmd> :=<print-skip-cmd>
    Skiplist:<nl>
    None.<nl>
    End of list.<nl>
    |<print-skip-cmd>
    Skiplist:<nl>
    <skiplist-nodes>
    End of list.<nl>

<skiplist-nodes> :=<skiplist-node><nl><skiplist-nodes> |<skiplist-node><nl>

<skiplist-node> :=<cell> has <tower-height> links to cells: <nl>
    <cell-list>

<tower-height> :=<int>

<insert-cmd> := <insert-cmd>
    Cell <cell-name> has been inserted.<nl>
    | <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 quad tree.<nl>

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

<rectangle-cmd> := <rectangle-cmd>
    Checked cell coordinates:<nl>
    <cell-list>
    Cells within rectangle:<nl>
<cell-list>
<draw-cmd> := <draw-cmd>
  Drawing.<nl>

<insert-fn-cmd> := <insert-fn-cmd>
  Created edge between <cell-name> and <cell-name>.<nl>
  Error: <cell-name> and <cell-name> are not of the same type.<nl>

#print-fn-cmd> := <print-fn-cmd>
  Feasible paths:<nl>
  <adjacency-list>
  Error: Bad argument.<nl>

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

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

<adjacency> := <cell-name>: <list cell names, ONE SPACE between and NO NEWLINES>

<build-sn-cmd> := <build-sn-cmd>
  Building support net.<nl>
  | <build-sn-cmd>
  Error: Bad argument.<nl>

#print-sn-cmd> := <print-sn-cmd>
  <component-list>
  Total # of connected components <int> <nl>
  | <print-sn-cmd>
  Error: No cells.<nl>
  | <print-sn-cmd>
  Error: Bad argument.<nl>

<component-list> := <adjacency-list> <component-list>
  | <adjacency-list><nl>
  Total length is: <total-length><nl>

#total-length> := <floating point number>
2 Instructions and Policies

2.1 General Information

Your project must execute on the OITs cluster. Otherwise, it will not be graded. Your program will be compiled and executed by the TA on the OITs 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 i.e., 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
javadoc *.*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 executables or object files.
3. cd part#
4. tar -cvf part#.tar *
5. gzip part#.tar
6. `mh420001/lbin/submit # part#.tar.gz`
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.cmsc420 or visit the tas during office hours.

2.4 Late Policy

The late policy is that posted on the newsgroup.

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.