**Design Idea Worksheet – Print out and fill out one of these for each design idea.**

**Your name: Scott Vincent in collaboration with Adam Brush**

**Design # (each of your designs should have a number): 1**

**This is a (circle one): --New design**

**--Variation, based on design # \_\_\_\_\_\_\_\_ (If it is a variation, you only need**

**to fill in the parts that differ from the basic design)**

**Data structure on which design is based:**

*Unordered Array*

**Describe how the data will be organized:**

*The data, i.e. Each ‘Entry’ object, will be arranged as it is read out of the .csv file. There will be no sorting mechanism involved. Duplicate values will be eliminated in the pre-build import.*

**Augmentations**: None of the data structures we have studied fit this problem perfectly, all of them have performance problems if used as-is. In this space describe any changes or additions you are making to the basic data structure to improve performance.

*I thought about ways to sort the data but it makes little sense because I don’t know what to order the array by, x or y- coordinate. Neither of the two makes a more efficient algorithm for the reasons we discussed in class.*

**Performance**: For each of the main methods of the Starbucks.h, describe in detail the algorithm you would use, and then give the asymptotic running time that would result.

**build**

**Algorithm:***create new array of size n;*

*loop through array from the pointer address which is passed in;*

*check and see if the location is a duplicate, if so don’t add it.  
copy objects into new array;*

**Running time: θ(N) : a linear build**

**getNearest**

**Algorithm:**

*Create default shortDistance;*

*Create an variable\* of type entry to hold the addy of the current shortDistance location;  
loop through array;{  
calculate distance of each object;  
compare to default shortDistance: if < save new shortDistance;*

*Update Entry\* var which belongs to this shortDistance;*

*}  
return object which belongs to shortDistance*

**Running time: θ(N)**

**Accuracy**: How accurate do you expect your solution to be? 1.0 is perfect, 2.0 means off by a factor of 2, and so on. Explain your reasoning.

*I expect this to be 1.0. Since I loop through all the values I will find the shortest distance.*

**Difficulty:** On a scale of 1 to 10, with 1 being “very easy” to 10 being “impossible,” how hard will it be for you to successfully implement this solution in the time allotted (1 week)?

*I think this will be a 2 or 3 with the difficulty coming from the reading of the .csv file.*

**Design # (each of your designs should have a number): 2**

**This is a (circle one): --New design**

**--Variation, based on design # \_\_\_\_\_\_\_\_ (If it is a variation, you only need**

**to fill in the parts that differ from the basic design)**

**Data structure on which design is based:**

*Linked List*

**Describe how the data will be organized:**

*Each node will contain the location x,y coordinates. We will keep the data in a linear order with prev & next pointers. Duplicate values will be eliminated in the pre-build import.*

**Augmentations**: None of the data structures we have studied fit this problem perfectly, all of them have performance problems if used as-is. In this space describe any changes or additions you are making to the basic data structure to improve performance.

*There’s not much to augment. This would be a better idea than an array if we were loading values directly from a .csv since we wouldn’t have the out-of-memory issues possible with our unsorted array solution.*

**Performance**: For each of the main methods of the Starbucks.h, describe in detail the algorithm you would use, and then give the asymptotic running time that would result.

**build**

**Algorithm:***create new list (starting with Sentinel);  
create a new node;*

*loop through array from the pointer address which is passed in;*

*check and see if the location is a duplicate, if so don’t add it.  
create a new node for each location object;*

**Running time: θ(N)**

**getNearest**

**Algorithm:**

*Create default shortDistance;*

*Create an variable\* of type entry to hold the addy of the current shortDistance location;  
traverse linked list;{  
calculate distance of each object;  
compare to default shortDistance: if < save new shortDistance;*

*Update Entry\* var which belongs to this shortDistance;*

*}  
return object which belongs to shortDistance*

**Running time: θ(N)**

**Accuracy**: How accurate do you expect your solution to be? 1.0 is perfect, 2.0 means off by a factor of 2, and so on. Explain your reasoning.

*I expect this to be 1.0. Since I loop through all the values I will find the shortest distance.*

**Difficulty:** On a scale of 1 to 10, with 1 being “very easy” to 10 being “impossible,” how hard will it be for you to successfully implement this solution in the time allotted (1 week)?

*I think this will be a 2 or 3 with the difficulty, again, coming from doing the upfront work, the reading of the .csv file.*

**Design # (each of your designs should have a number): 3**

**This is a (circle one): --New design**

**--Variation, based on design # \_\_\_\_\_\_\_\_ (If it is a variation, you only need**

**to fill in the parts that differ from the basic design)**

**Data structure on which design is based:**

*Array Grid*

|  |  |  |
| --- | --- | --- |
| A3 | B3 | C3 |
| A2 | B2 | C2 |
| A1 | B1 | C1 |

**Describe how the data will be organized:**

*The array will be semi-sorted by the boundary of the cells. A NxN matrix will be constructed and location objects will be placed inside the cell which contain the object’s (x,y) coordinate.*

*Duplicate values will be eliminated in the pre-build import.*

**Augmentations**: None of the data structures we have studied fit this problem perfectly, all of them have performance problems if used as-is. In this space describe any changes or additions you are making to the basic data structure to improve performance.

*The making of the grid should be fairly straight forward. Using the shortDistance we could check the surrounding cells but I think that would get kind of tricky. Worst case scenario you would check three adjoining cells but it might be easier to circle around the target cell to the eight adjoining cells and check the distances. You could do A LOT of math checking to see if your shortDistance < (queryPoint – wallEdge) to see which adjoining cells you could omit from your search.*

**Performance**: For each of the main methods of the Starbucks.h, describe in detail the algorithm you would use, and then give the asymptotic running time that would result.

**build**

**Algorithm:**

*Divide space into a 2-d array;*

*Keep track of cell walls;*

*Loop over locationObject array;{  
 check coordinates;*

*if the location is a duplicate don’t add it;*

*place locationObject into a matching cell;*

**Running time: Cell# \* (N) = θ(N)**

**getNearest**

**Algorithm:**

*Check which cell your queryPoint belongs to;  
 run getDistance routine to find the shortDistance;*

*Check to see if shortDistance is less than the distance from queryPoint to the walls of the cell,*

*if ‘true’ then it is the shortDistance;*

*else  
check to see which walls are closer and run getDistance routine on all the locationObjects in that cell;*

*return locationObect;*

**Running time:** *Your worst case would be O(N) because if for some reason all the location entities were located in just a single cell and that’s where your queryPoint was located then you’d have to check the distance to each point be no better than an unsorted array.*

**Accuracy**: How accurate do you expect your solution to be? 1.0 is perfect, 2.0 means off by a factor of 2, and so on. Explain your reasoning.

*It would be 1.0 If we check the shortDistance () value against all the other surrounding cells*

**Difficulty:** On a scale of 1 to 10, with 1 being “very easy” to 10 being “impossible,” how hard will it be for you to successfully implement this solution in the time allotted (1 week)?

*7. This will require A LOT of code, debugging and testing to (1) find our starting cell, (2) get it’s shortDistance, (3) check wall boundaries, (4) decide which surrounding cells to check, (5) Update shortDistance if necessary (6) return the correct shortDistance. A sure fire strategy for problems!*

**Design # (each of your designs should have a number): 4**

**This is a (circle one): --New design**

**--Variation, based on design # \_\_\_\_\_\_\_\_ (If it is a variation, you only need**

**to fill in the parts that differ from the basic design)**

**Data structure on which design is based:**

*Divide and Conquer Strategy: Binary Tree*

**Describe how the data will be organized:**

*The data will go in as we learned in class. We randomize the array in the setup function then call the build function using the median object’s coordinates to make a binary tree.  
  
Another possibility is once we have our imported array we sort it around a pivot point based on a median (ex. Pivot=.5) to attain a optimally balanced tree in our build function. But, I suggested this question in class and we discussed how a randomized input would be approximately as good on account of the lg(n) running time and wouldn’t incur the pre-sorting cost.*

*Duplicate values will be eliminated in the pre-build import.*

**Augmentations**: None of the data structures we have studied fit this problem perfectly, all of them have performance problems if used as-is. In this space describe any changes or additions you are making to the basic data structure to improve performance.

*We would have to adjust our binary tree into K-D Tree for our two-dimensional grid. Each tree is of max two nodes (none, a left and/or a right) keeping with the binary tree paradigm. We are performing an insert of elements based on whether the input level is an x or y level with small values going on the left and larger values going on the right.*

**Performance**: For each of the main methods of the Starbucks.h, describe in detail the algorithm you would use, and then give the asymptotic running time that would result.

**Pre-build: randomize Entry objects;**

**build**

**Algorithm:** *Based on insert*

*Check to see if we are at the root node (base case);  
If level is even:  
 calculate median x-value;  
 if x < median; insert left else insert right;*

*Else If level is odd:*

*calculate median x-value;  
 if y < median; insert left else insert right;*

*add a layer of depth;*

**Running time: θ(N)**

**getNearest**

**Algorithm:** *Based on search*

*Check: if node == null; return null*

*Check: if we are at the root node (base case); return node;*

*Set root node;*

*{*

*// write a recursive function which returns the entry which is closest*

*Entry\* if best\_left getNearest();*

*Entry\* if best\_right getNearest();*

*Entry\* currentEntry = node -> data*

*Move down a level;*

*}*

*Return Entry\**

**Running time: Ω( lg (N))**

**Accuracy**: How accurate do you expect your solution to be? 1.0 is perfect, 2.0 means off by a factor of 2, and so on. Explain your reasoning.

*I reckon it’ll be a 1.0 if I can get the tree working correctly. But if I blorf the tree then it could easily be a factor of 2 or more.*

**Difficulty:** On a scale of 1 to 10, with 1 being “very easy” to 10 being “impossible,” how hard will it be for you to successfully implement this solution in the time allotted (1 week)?

*Probably a 6-7 with my understanding of trees. What we learned in class seems reasonable enough if we just trust in recursion.*

**Design Idea Summary –**

**Summary of results from Design Idea Worksheets:**

Fill in one column of the table for each of your designs. (No, you don’t have to have 15 designs. I just wanted to make sure you had plenty of space)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **UA** | **LL** | **Grid** | **KDBT** |  |  |
| **Design #** | **1** | **2** | **3** | **4** |  |  |
| **build**  **running time** | **θ(N)** | **θ(N)** | **θ(N)** | **θ(N)** |  |  |
| **getNearest**  **running time** | **θ(N)** | **θ(N)** | **O(N)** | **Ω( lg (N))** |  |  |
| **Estimated accuracy (1.0 is perfect)** | **1.0** | **1.0** | **1.0** | **1.0-2.0+** |  |  |
| **Estimated Difficulty** | **3** | **3** | **7** | **6** |  |  |

**Which design will you use:** (Note – If you do NOT use this design for phase 2, you will lose 10% on phase 2)

*Design 1*

**Explain why you chose this design:**

*Due to its simplicity. If I can get it to work then I’ll try my hand at the tree structure.  
Right now I’ve spent an inordinate amount of time on unsuccessfully trying to load the contents of a file into a vector so my time for actually playing with and learning about trees may be limited. If I had gotten my unsorted array figured out by now, as I had hoped, then I would certainly pick the KD Tree.*