Syrian Arab Republic Albaath University Department of Software Engineering and Information Systems		جامعة البعث كلية الهندسة المعلوماتية قسم هندسة البرمجيات ونظم المعلومات

بنى معطيات للبيانات ذات المفاتيح متعددة الأبعاد

Data Structures for multidimensional data:
"a study of multidimensional-key data with application and tests"

Presented by **Hussain HUSSAIN**	**Supervised by** **Dr. Suhel Hammoud** **Dr. Rania Loutfy**

العام الدراسي 2014-2015

Data Structures for multidimensional data

Abstract

Several methods to solve the problem of orthogonal range searching in a dynamic structure are presented. A deep study and testing is presented of the proposed structures to solve the problem (hash tables, red-black trees, quadtrees).

The problem of range searching becomes really important for geographic information analysis, object position tracking, objects collision detection and many other problems. The focus is on the representation of points inside a map, and how we can conclude information about some range in an efficient way.

We came up with some results showing the difference between proposed structures based on different considerations. Multiple points of view were considered to show the specification of each structure.

ملخص المشروع

نستعرض مجموعة من الطرق لحل مسألة البحث في مدى مستطيلي الشكل ضمن فضاء ثنائي البعد باستخدام بنية معطيات ديناميكية. دراسة عميقة وتجريب مكثف لبنى المعطيات المقترحة (جداول التقطيع، الأشجار ذات الأحمر والأسود، الأشجار الرباعية).

مسألة البحث ضمن مدى هي مسألة مهمة بالنسبة لمواضيع تتعلق بتحليل المعلومات الجغرافية، تتبع مواقع الأغراض، كشف تصادم الأغراض وغيرها من التطبيقات. تم التركيز على موضوع تمثيل النقط ضمن خريطة وكيفية استرجاعها وأيضاً كيفية القيام ببحث على نطاق معين منها.

تم في النهاية الحصول على نتائج توضح الفروق بين المعطيات المقترحة بناء على اعتبارات مختلفة. وجهات نظر مختلفة تم أخذها بعين الاعتبار من أجل الحصول على ميزات تلك البنى.

Table of contents

1 Introduction

1.1 Overview of the problem

We need a dynamic data structure that represents a set of points in a 2D-space with some data attached to each point. This data structure should provide the following operations:

- Insertion of a point: given a position (x, y) with data attached to it, insert the data in the corresponding position.
- Finding a point: given a position (x, y), find data inside of this position.
- Range searching: given a range represented by a rectangle, return some statistics about this range (e.g. number of points, average of data, the set of points,…) _in this project we concentrate on this operation especially in the case of returning the set of points inside a given range.

1.2 Goals

The aim is to find a data structure that minimizes time and space complexity for the stated operations considering a certain rate for each operation.

1.3 Methods and tools

- The language in which the structures were implemented is java. Java was chosen because of its simplicity and because of its powerful collections API.
- Netbeans (7.4 and 8.0.2) is the IDE used to implement the structures, this IDE supports java and it is very simple. Moreover, it contains important tools for us.
- Netbeans profiler was used to measure other considerations such as memory and CPU usage.
- The language in which the test generators were implemented is C++.
- The IDE in which C++ code was implemented is Code::Blocks 13.12.
- Windows FC command was used to compare output files to the right output. The result output must match the expected output, to test this we used FC command.
- https://www.draw.io/ (Flow Chart Maker & Online Diagram Software) was used to draw some charts and diagrams especially in the design in chapter 3.
- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html (Red-Black Tree visualizer) was used to make the figure of the red-black tree in section 2.2.
- Microsoft Office (Microsoft Excel and Microsoft Word) was used to make most of the tables and diagrams.

1.4 Methodology

A set of data structures is reviewed to solve this problem. Each of the data structures is studied in 4 main steps:

1- Abstraction and reduction

The data structure, in its abstract form, is reduced to solve our problem. The methods for each operation are also studied in its abstract form and algorithm is provided to perform this method.

2- Study and analysis
   Each operation in the data structure is studied to calculate time and space complexity.
3- Implementation
   The data structure was then implemented using java mostly with the use of Java Collections.
4- Testing and profiling
   The implemented code was then tested on several test cases which are generated following different distributions (see Appendix B).
   The usage of memory and CPU by the program, and more specific: by each method in the program was then studied and shown properly.
   Another measure which is the depth of some tree structures was shown as well.

This methodology is reviewed according to these main chapters:

1- Study and analysis
2- Basic design and implementation
3- Testing
4- Profiling

After that, we show the results and challenges and add some appendices for more clarification.

2 Study and analysis

2.1 Hash tables

2.1.1 Overview

Hash tables are well-known data structures which depends on defining a mapping of the key to get the address of the desired item. In other words, it is a structure that maps *keys* to *values* or to a *bucket of values* using a hash function which is considered to involve as less effort as possible in detecting the desired address.[1]

$$Hash: Key \rightarrow Address$$

We are not interested in the details of the hash tables, hash functions or address collision detection; the only thing we want to consider is that this structure is one of the most efficient structures in address mapping problems.

2.1.2 Study in our problem

In our study, we are going to store points in a hash table with their position (x, y) as the key, and their data as values. We will show the methods for each operations with the analysis.

2.1.2.1 Algorithm

2.1.2.1.1 Insertion

The insertion is simply by adding the data to the bucket of the appropriate address which is calculated using the hash function:

$$Bucket[Hash(Key)].Add(Data)$$

2.1.2.1.2 Search

Very similar to insertion operation, but it is done just by reading the desired data:

$$Return\ Bucket[Hash(Key)].Find(Key)$$

2.1.2.1.3 Range Searching

Unfortunately, this operation represents the bottle neck of this powerful structure because there is no intelligent way to retrieve information related to a range from a structure which already stored them with no respect to their spatial information.

So, we have to iterate on all keys and check whether a specific key falls into the desired range and decide upon it:

```
for each key in Hash Table:
     if key falls within Range:
          add it to result
return result
```

2.1.2.2 Analysis

2.1.2.2.1 Time

- In case of insertion or search, the total time needed is the time of the hashing function plus the time of finding a key in the bucket which is implemented as a linked list. Considering the size of a bucket is constant and the time of the hash function is constant too, the total time needed is thus **constant**.
- Since the hash table structure is considered an unordered associative array which means the keys has nothing to do with their order in the table, we have to test every key in the table whether it falls within the desired range or not; if so we add its value to the result. This operation takes **linear** time because getting each key needs constant time.

2.1.2.2.2 Space

The hash table pre-allocates an array of buckets. The size of this array is fairly enough to hold all data which is predicted to be stored inside the table, so simply we can consider that the size of the table is **linear**.

2.1.3 Generalization

If the key is d-dimensional, represented by a vector $x_1 \quad x_2 \quad \dots \quad x_d$, the implementation remains simple; the index of the bucket becomes the hash of the d-dimensional key. Time complexity for insert and search becomes $O(d)$. Time complexity for range searching remains the same $O(n)$.

Space complexity *,in terms of number of points,* is obviously **linear** .

2.2 Red-Black Trees

2.2.1 Overview

As stated in Chapter 13 in [2], A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. Some constrains in this structure, which are known as the **red-black properties**, guarantee that any path from the root to a leaf is not more than twice as long as another path form the root to a leaf node, which means the tree is approximately balanced. Figure 2-1 a smple of red-black tree shows an example of a red-black tree.

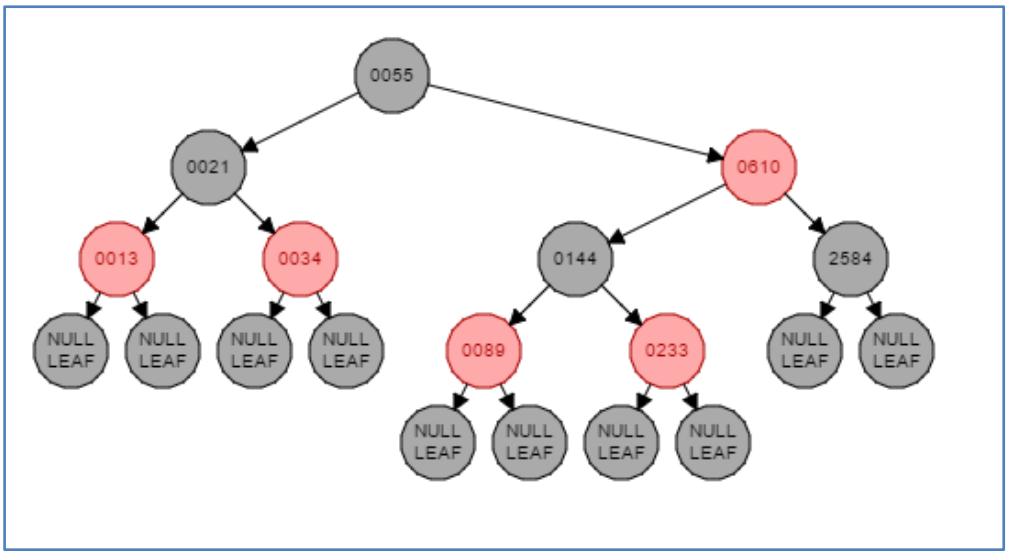


Figure 2-1 a smple of red-black tree

2.2.2 Study in our problem

We store the points in a red-black tree sorted according to their x coordinates and then y coordinates i.e. if two points have the same x values we sort them according to their y values. Let's say that this comparison is defined by the **operator** $<$.

2.2.2.1 Algorithm

2.2.2.1.1 Insertion

The insertion of a node is a little bit complicated, and we are not interested in the details of this process, but reader can see Chapter13 in [2] for details.

Insertion process starts similarly to insertion in regular binary search trees, but one of the properties of the red-black trees may be violated after this insertion, so a rotation process and a re-coloring process will be executed to solve the inconsistency. For simplicity we show the process in its major steps:

$$Find\ the\ NULL\ node\ X\ in\ which\ the\ new\ node\ N\ will\ be\ inserted$$
$$Replace\ X\ by\ N$$
$$If\ violation\ occured, solve\ it$$

2.2.2.1.2 Search

Searching in a red-black tree is similar to search in a regular binary search tree. We either go left or right or return the current value. The value of the current node helps in making the decision of 'Where to go?'.

$$Node := Root$$
$$While\ (Node\ not\ NULL)$$
$$if(Node.value = Goal)\ return\ Node$$
$$if(Node.value > Goal)\ Node = Node.Left$$

$$else\ Node = Node.Right$$
$$return\ NULL$$

2.2.2.1.3 Range Searching

The range search here depends on finding the lower bound and the upper bound of the range according to x coordinates, and then filtering the points out the points which do not satisfy the conditions (out of y's range).

$$Tree1 := Tail(Tree, Range.minX)$$
$$Tree2 := Head(Tree2, Range.maxX)$$
$$Result := Filter(Tree2)$$

2.2.2.2 Analysis

Since the tree is approximately balanced, the depth of the tree is $O(\log(n))$ where n is the total size of the tree i.e. the number of points.

2.2.2.2.1 Time

Search process requires a number of iterations which doesn't exceed the depth of the tree, which is $O(\log(n))$ iterations. Since each iteration needs constant time, the total time of the search process is $O(\log(n))$.

Insertion process includes searching which is $O(\log(n))$, and may include some rotations and re-coloring processes which are constant in total. Thus, the insertion process will also take $O(\log(n))$ time.

Range searching operation takes logarithmic time for both finding upper and lower bounds but probably takes linear time to filter out the points that do not fall inside the range. Just imagine a range query in which the x dimension is too large while the y dimension is relatively small like the one in Figure 2-2. This range may contain a small number of points while the finding the upper and lower bounds will result in a large tree in which most of the points do not belong to the range. The process of filtering the resulting tree will take linear time.

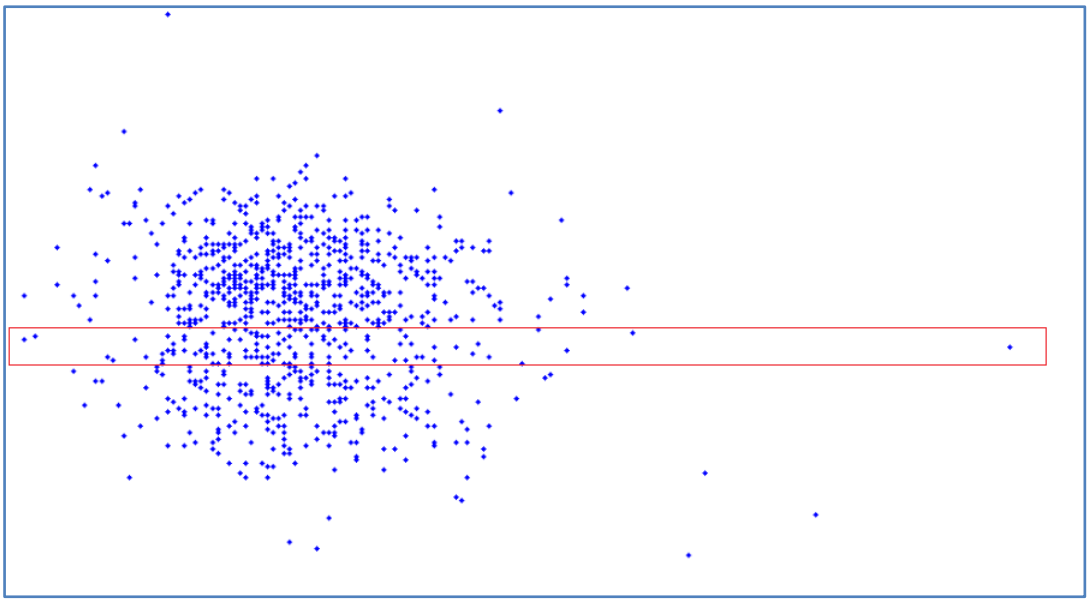


Figure 2-2 Range search query with large x dimensions

Therefore, this method will take linear time in the worst case.
More formally, we will define the complexity as follows:
Worst case: $O(n+k)$
'Good' case: $O(\log(n)+k)$

Where k is the size of the answer.

2.2.2.2.2 Space

The space of this structure is **linear** *,in terms of number of points,* since it stores for each node: position, data of the point, two pointers to left and right children plus a bit for color (red or black).

2.2.3 Generalization

This structure can be generalized to have a key with more than two dimensions (d dimensions). Let's say the key is a vector $x_1 \quad x_2 \quad \dots \quad x_d$, the values are sorted according to their first dimension, if the first dimension is equal, then we compare the next dimension and so on. Searching and range searching just depend on the first dimension

In range searching, time linearity in the dimensions, other than the first one, also appears here and makes a huge problem. The generalization of this structure is a disaster in case of range searching. Time for range searching: $O(d \times n + k)$

The space complexity remains **linear**.

2.3 Quadtree

2.3.1 Overview

2.3.1.1 Definition

As stated in[3], a **quadtree** is a tree data structure in which each internal node has exactly four children. Quadtrees are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular (see Figure 2-3), or may have arbitrary shapes. This data structure was named a quadtree by Raphael Finkel and J.L. Bentley in 1974. A similar partitioning is also known as a *Q-tree*.

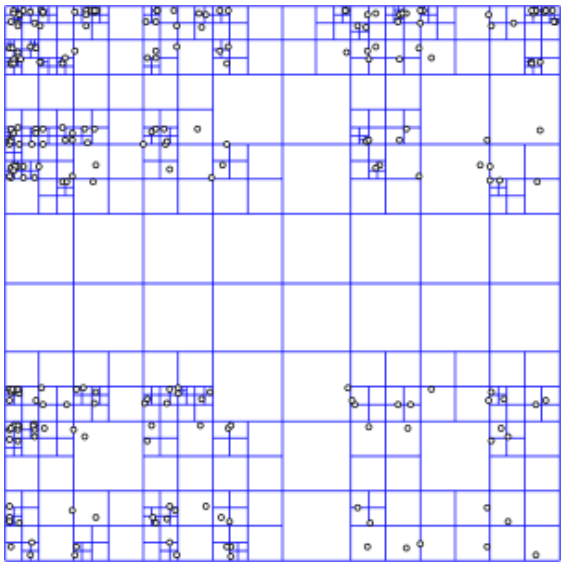


Figure 2-3 An example of spatial subdividing[3]

There are variant forms of quadtrees and they have many different specifications but the property that those forms share and we are interested in is that this structure provides a fine partitioning of the space. However, we are reviewing the most famous usages of quadtrees later.

2.3.1.2 Structure

The quadtree is a rooted tree whose nodes can either have four children or no children at all. The four children of a node divide the space of this node into four nodes with equal sizes. So, each node represents a square or a rectangle (i.e. quadric). Besides, each node in the quadtree has members defining the position and the size of it as well as other data members.

Figure 2-4 illustrates the subdividing of the space of some node, recursively we can get some sort of trees similar to the one shown in Figure 2-4.

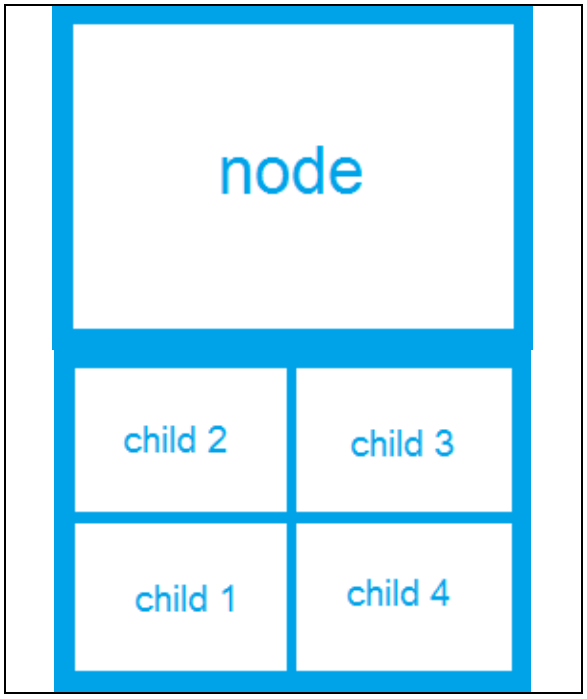


Figure 2-4 Quadtree spatial subdividing

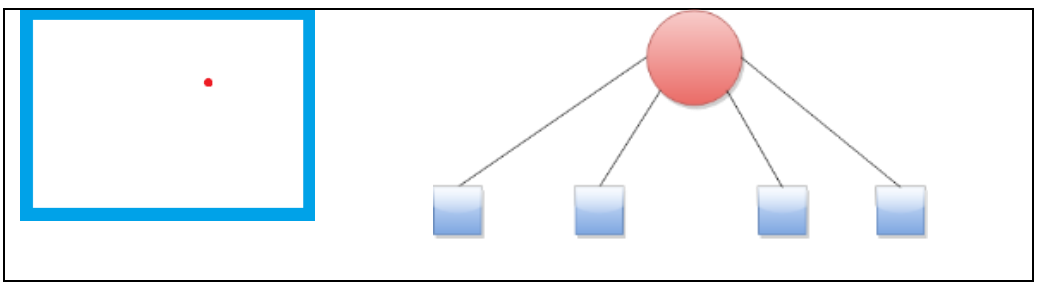
2.3.1.3 Common usages

Quadtree is most famous for representation of points inside maps, image representation and game development. For more details about those usages, you can see the references stated bellow.

2.3.1.3.1 Representation of points inside maps

Consider a rectangular map of points (probably cities, hotels or even stars). Instead of representing these data using traditional structures (e.g. arrays, maps, hashing tables) we will represent it using the hierarchical model we've talked about: quadtree.

Let's say that each quadric can contain at most one point, and if one point is about to be inserted in a quadric that already contains a point, this quadric will be split into 4 equally-sized quadrics and each point will be inserted in the right quadric. A small example is illustrated through Figure 2-5.



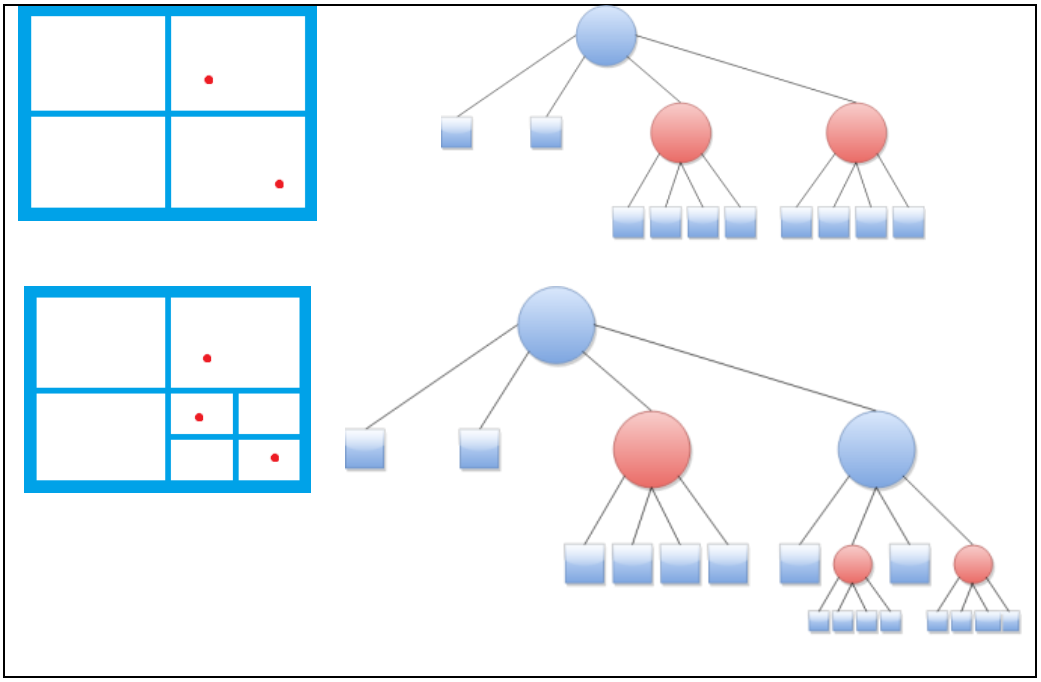


Figure 2-5 Quadtree insert sample

This model is adapted in this project and will be studied in details.

2.3.1.3.2 Image representation

One of the most common methods to represent an image is to store a two-dimensional array of pixels and set each element to the value of a color. However, there are many ways to compress images but we are not interested into them right now; one can find them in any book about multimedia systems.

For simplicity, let's consider only black and white colors and the whole image as the root node of the tree. We keep dividing nodes into four equal quadrics, as mentioned before, until the whole node have a single color.

This sort of quadtrees is called a 'region quadtree'[4]

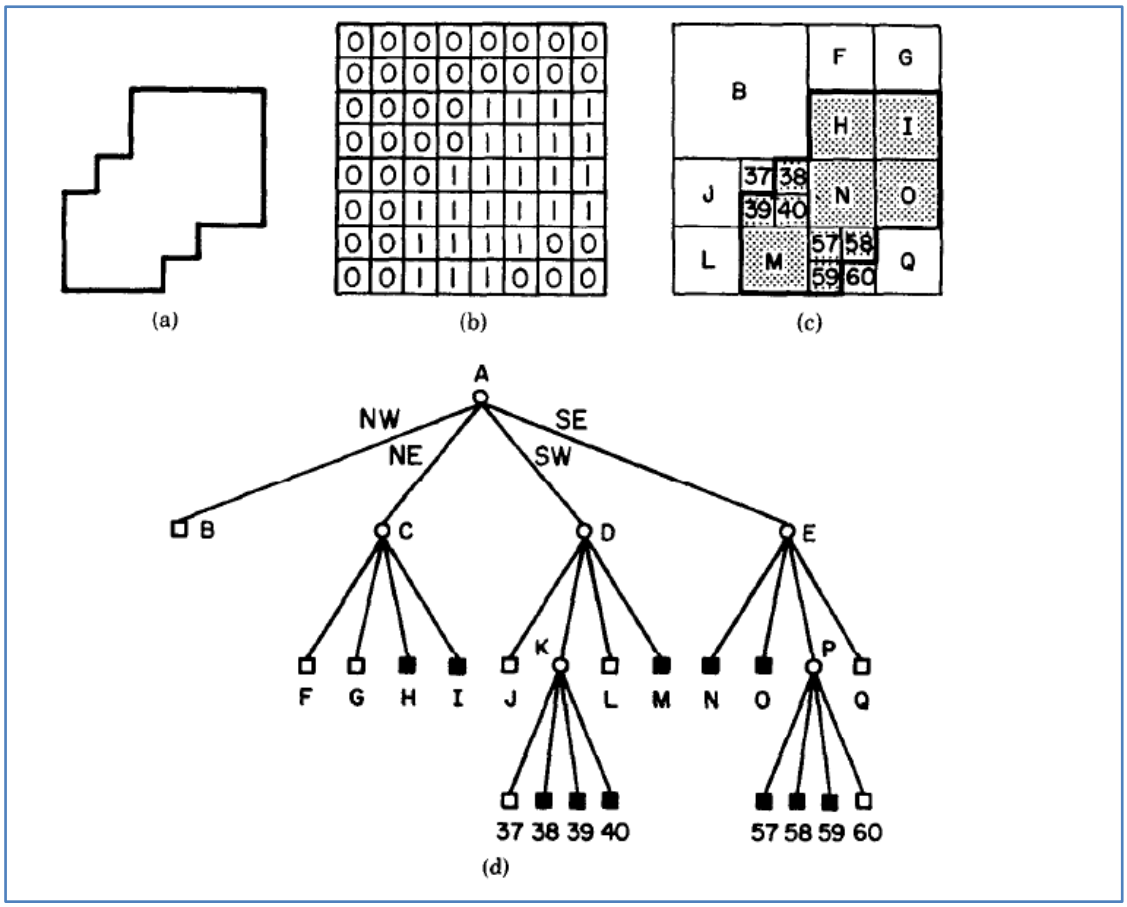


Figure 2-6 image representation using quadtree [4]

2.3.1.3.3 Game development (Collision detection)

One of the most used operations in game development in general is to detect whether two objects hit each other. Brute force solution for such a problem may exhaust the resources, since it requires many comparisons between objects and most of these comparisons are useless because the two objects involved may be too far from each other. However, one of the commonly used methods to compare only objects which are close is to use the quadtrees.

The idea here is to check only the objects which have a part of them inside the same quadric of another object.

All objects are inserted into the structure, and any object that cannot fully fit inside a node's boundary will be placed in the parent node[5]. So to detect whether an object collides with one another, we should only check the objects that lie in the same node or in an ancestor node.

Figure 2-7 shows samples of collision detection taken from [6], collisions are colored in red.

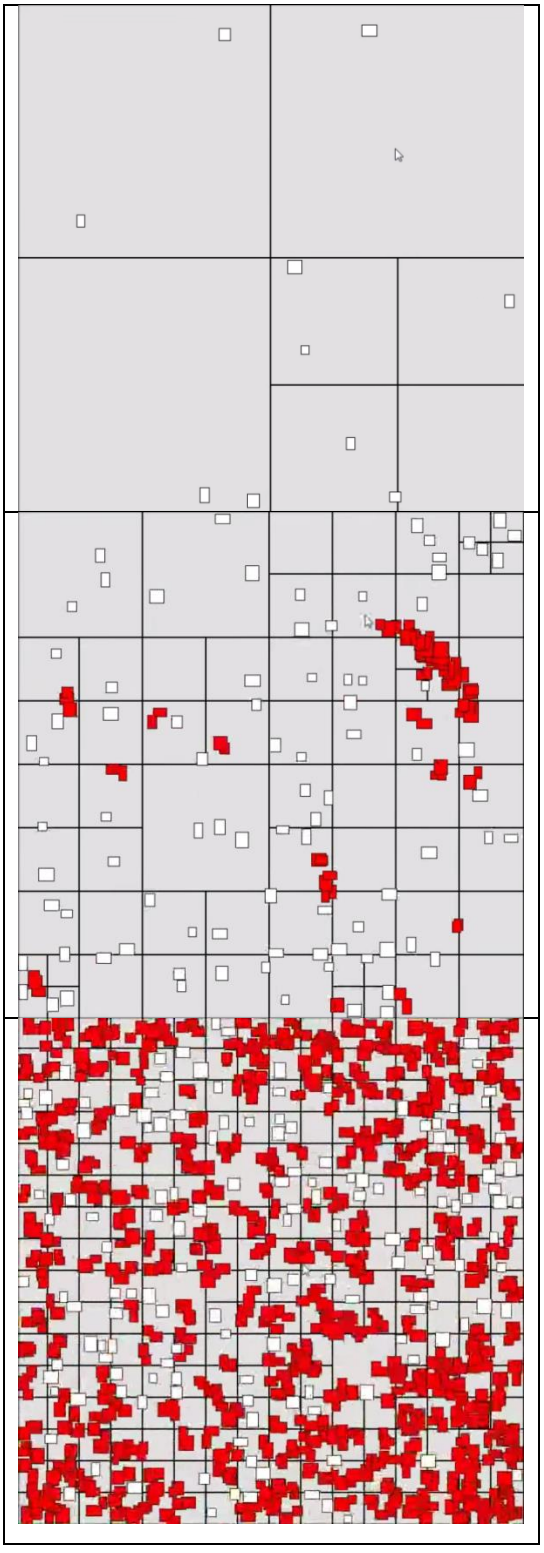


Figure 2-7 Collision detection using quadtree

2.3.2 Study in our problem

2.3.2.1 Introduction

In our problem, we are representing the points of the map using a quadtree. This pointer-based structure will contain our points and their data attached to their spatial information, which means that a search process will consider the position information of an object. Here

we review algorithms for each operation, for details about implementation see section 3.421 and testing see chapter 4.

2.3.2.2 Algorithms

2.3.2.2.1 Insertion

```
boolean insert(Point p){/* true when the point is inserted, false
                                    otherwise */
    - if the node is not a leaf or contains another point, then insert p inside
      one of its children according to its position
    - else, insert it into this node
    }
```

2.3.2.2.2 Finding

```
Point find(int x,int y){
    - if found in this node, return it
    -else if found in a child node (according to its position), return the result
    - else, NULL  otherwise
}
```

2.3.2.2.3 Range Searching

```
Set <Point> rangeSearch(Range r){
    - if r doesn't intersect with the range of this node, return an empty set
    - else if r contains the range of this node, return all data in this node
    - else, return union of range searching for all children
}
```

2.3.2.3 Analysis

Let's consider the number of points to be inserted is $O(n)$, and the area we are dealing with is $O(w \times h)$.

2.3.2.3.1 Time

- For insertion and finding, on each step we are getting rid of ¾ of the space, so the space is divided by 4 on each step until the node is a leaf or the space is undividable. Therefore, the complexity of each insertion or finding operation is logarithmic in the area $O(\log_4(w \times h))$.
  However if we considered the number of points to insert (regardless the area of space), we will find that: in the worst case, the operation time is linear in n, which means for each operation (insert or find) the time is $O(n)$. Theoretically, these are bad news, but if we thought of a practical application, the worst case will rarely happen because it needs a certain positioning for each point. This positioning usually happens with small n, so we claim that we are still good. Anyway, this claim will be studied and documented in the testing phase.
- For range searching operation, things are a bit more complicated, so for simplicity we will consider that paths to nodes in the range are disjoint in the worst case. Thus, the complexity will turn to multiply the complexity of finding a single point by the number of point in the considered range which means $O(k \times \log_4(w \times h))$ _*or* $\mathrm{O(k \times n)}$ *considering the number of points.*

2.3.2.3.2 *Space*

Considering a space full of points (in the worst case), the tree will have a depth of $O(\log_4(w \times h))$, and at level i, we have 4^i points. Totally, we will have $O(w \times h)$ space. However, considering the size of the input (number of points to be inserted), the size of the tree is **linear** in this number $O(n)$.

2.3.3 Generalization

In the case of more than two dimensions, we have a key $x_1 \quad x_2 \quad \dots \quad x_d$. For each dimension we should subdivide the space into two equal-sized parts, so we get a smallest part a 'quadric of d dimensions' instead of two.

Time complexity for insert and search is also **logarithmic** in the size $O(\sum_{1 \le i \le d} \log(size(i))$.

Time complexity for range searching becomes $O(k \times \sum_{1 \le i \le d} \log(size(i)))$.

Space complexity of the structure becomes $O(\prod_{1 \le i \le d} log(size(i))$ *–or* $O(d \times n)$ *considering the number of points.*

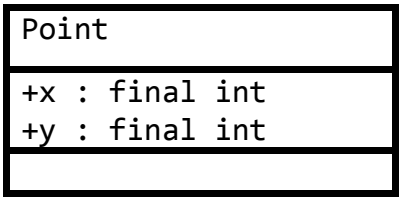
The generalization for a quadtree on 3 dimension is a very known structure known as ***Octree***

3 Basic Design and Implementation

Here we show the basic design and implementation of the project. The containers being studied all implement the interface TDK`Struct` *(two-dimensional-key structure)*. Some supporting structures for our problem are implemented and shown properly.

3.1 Point

The very simple, yet very impressive structure in our problem is the `Point` class which is defined as the key of the containers used. This class has only two members; `x` and `y`. The structure is meant to be immutable for the sake of simplicity. *For the benefit of our structures, this class needs to be comparable.*



An extension of `Point` is `DataPoint` class which contains data of any type.

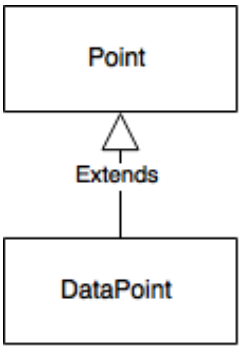


Figure 3-1 Point and DataPoint classes

3.2 Range

Another impressive structure, especially in the range searching problem is the `Range` class. This class provides some operations that simplify working with intersection and inclusion of multiple ranges.

3.3 TDKStruct

The main container structures all implement the interface of a TDK`Struct` which implies the existence of space borders (the space which contains the points) and the operations of insert, search and range search.

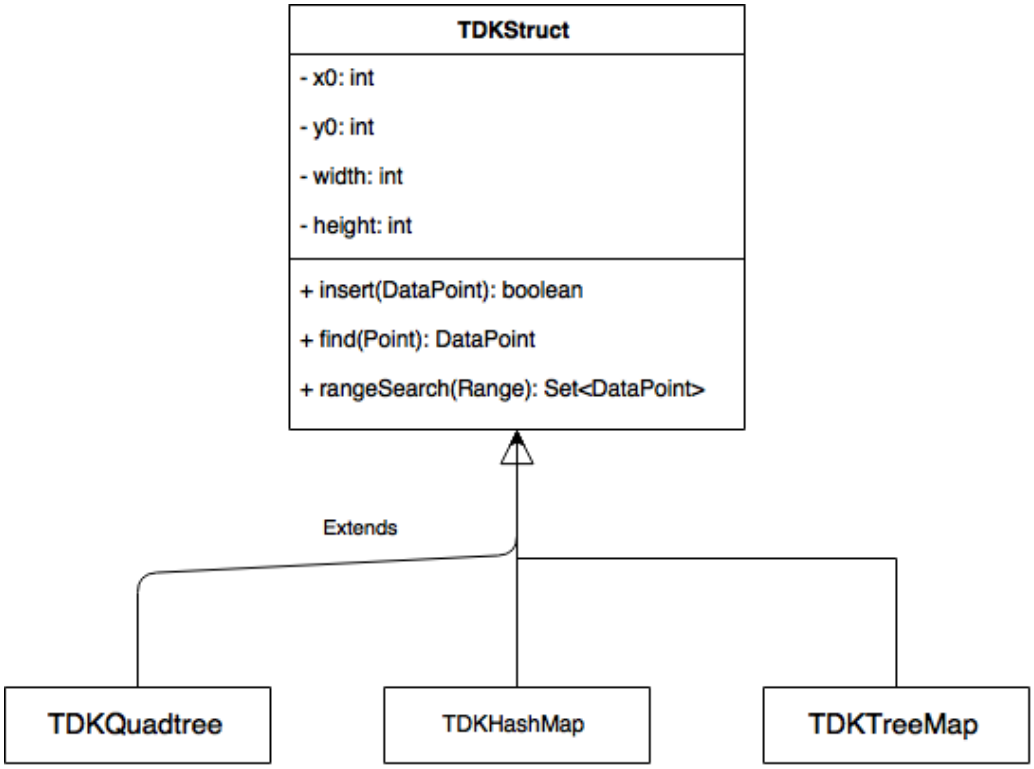


Figure 3-2 Main structures classes

3.4 TDKQuadtree

The quadtree is a self-referencing structure it is either a leaf or a node with four **children**. **Data** are exclusively held in the leaves. Other data members can hold the position and size information.

TDKQuadtree
+isleaf : boolean +p : point* // data with exact position +child : Quadtree*[4]

Generally, a quadtree Q can contain a point P iff this condition holds:

$$P.x \geq Q.x_0 \ \& \ P.x < Q.x_0 + \ Q.X$$

$$P.y \geq Q.y_0 \ \& \ P.y < Q.y_0 + \ Q.Y$$

Child index follows the rule that the LSB means that the child is to the right and the second LSB means that the child is to the top.

10	11
00	01

Children with LSB set have $x \geq x_0 + \frac{X}{2}$

Children with second LSB set have $y \geq y_0 + \frac{y}{2}$

- Algorithm of insertion

```
boolean insert(DataPoint p){/* true when the point is inserted, false
                               otherwise */
    - set the current point to p : is leaf && no point currently in the
      quadtree
    - insert the current point and p into the correct children : is leaf && a
      point is currently in the quadtree
    - choose the correct child to insert p into : is NOT leaf
    }
```

- Algorithm of finding a point
  This method was written in an iterative style in order to perform better, the code is equivalent to the following.

```
DataPoint find(int x,int y){
    - current point : is leaf && current point == (x,y)
    - child[idx].find(x,y) : is not leaf && child[idx] != NULL
    - NULL : otherwise
}
```

- Algorithm for range searching
  This method is also implemented in an iterative style equivalent to the following, and without the overhead of creating and deleting pointers or union of sets.

```
Set<DataPoint> rangeSearch(Range r){
  -   {} : r doesn't intersect with the current range
  -   {current point} : is leaf
  -   ⋃_{c:child} c.rangeSearch(r) : otherwise
}
```

3.5 TDKHashMap

An extension of `java.util.HashMap` which also implements the `TDKStruct` interface.

The implementation of the `insert` method is equivalent to `java.util.HashMap.put()` method, but with few modifications. The implementation of `find` method is trivial as well and almost equivalent to `java.util.HashMap.get()`.

The implementation of `rangeSearch` method implies searching the whole keys and filter out the keys which do not fall within the range.

```
Set<DataPoint> rangeSearch(Range r){
  -   for each key k in keyset()
    o  if k.fallsWithin(r) then result.add(find(k));
}
```

3.6 TDKTreeMap

An extension of `java.util.TreeMap` which also implements the `TDKStruct` interface.

The implementation of the `insert` method is equivalent to `java.util.TreeMap.put()` method, but with few modifications. The implementation of `find` method is trivial as well and almost equivalent to `java.util.TreeMap.get()`.

The implementation of `rangeSearch` method implies searching for the predecessor of the beginning of the range (lower-left corner) and the successor of the end of the range (the top-right corner) and then search the keys between them and filter out the keys which do not fall within the range.

```
        Set<DataPoint> rangeSearch(Range r){
          -   predecessor = lowerBound(r.lowerLeft());
          -   successor = upperBound(r.upperRight());
          -   for each key k in (predecessor:successor)
            o  if k.fallsWithin(r) then result.add(find(k));
}
```

4 Testing

4.1 Introduction

Each one of our structures is tested according to two different kinds of tests. The first kind of tests is the test in which we just have two types of queries: insert and search and the second is the one in which we have range search queries. These two kinds of tests are carried out to compare the run time for each structure (average time insert, search and range search).

Tester classes are reviewed for each kind of test. Moreover, test data which is also decomposed into two types: one of which contains uniformly distributed queries in each operation scattered across the space of the input file, and another one in which we try to simulate the real distribution of the cities in the real world contains queries distributed across the space according to more complex distribution. We call this distribution **clustered distribution** since it depends on clustering points around positions of interest.

These distributions are illustrated in details in Appendix B.

4.2 Insert-Search tests

In these tests we have two types of queries: insert and find, these queries are stored in an input file. In this file, the two types of queries are separated: first, the insertion queries come and then the search queries.

4.2.1 Tester class

A tester class is used to:

1- Read the input file and store the queries in memory.
2- Test each structure whether it answers right or wrong for the queries.
3- Report the running time(the average time for insert and search), and other considerations (related to depth measure).
To see this tester class in details refer to Appendix A

4.2.2 Test data and results

Test data also has two types: uniform and clustered, for each of which we run the tester and compare our structures.

4.2.2.1 Uniformly distributed tests

In these tests, we have a number of files, each one of which contains a number of points $\#points$, a number of queries $\#queries$ and a space of $w \times h$. Those numbers are increasing across the files. Results are shown in the following table

	Average

			insert			search		
Space w×h	~#queries	~#points	HashMap	TreeMap	Quadtree	HashMap	TreeMap	Quadtree
100x100	200	100	0.001395	3.38137	21.15718	0.000834	0.004859	69.80453
	2000	1000	0.001732	2.853028	7.057899	0.000863	0.010995	7.930082
	10000	5000	0.001515	3.838709	3.136586	0.001037	0.003449	5.14327
1000x1000	10000	5000	0.001199	3.831961	2.785827	0.000594	0.002362	5.933348
	20000	10000	0.001423	1.880658	4.800108	0.000748	0.002313	4.652797
	200000	100000	0.000302	2.490901	5.063292	0.000249	0.001109	15.21406
2e5x2e5	400000	200000	0.000305	2.335949	3.51403	0.000131	0.001411	10.70299
	1000000	500000	0.000353	2.156555	2.631846	0.000116	0.001353	9.348227
	2000000	1000000	0.00027	3.340877	2.428343	8.66E-05	0.001915	12.85363

Table 4-1 uniform distribution results

We see an average insertion time for each structure as:

HashMap	TreeMap	Quadtree
0.000944	2.901112	5.841679

Table 4-2 average insertion for uniform distribution

An average search time:

HashMap	TreeMap	Quadtree
0.000518	0.003307	15.73144

Table 4-3 average search for uniform distribution

Measuring the average depth of the Quadtree we got values for insertion and searching, those values are showed in this table.

#points	depth	#queries	depth
98	5.214286	196	4.612245
963	6.738318	1836	6.246732
3940	7.565482	6338	7.179867

4986	8.017449	9941	7.458505
9942	8.540837	19800	7.953535
95172	10.01189	181320	9.499906
200000	10.72746	400000	10.13321
500000	11.38228	1000000	10.79103
1000000	11.88266	2000000	11.29326

Table 4-4 quadtree depth in uniform test

4.2.2.2 Clustered distributed tests

4.2.2.2.1 Clustered distributed test #1

In this test, we have a space of 10000×10000 and a number of points scattered across it according to the clustered distribution. For each test, we have a fixed number of search queries which is 10000.

In this table we show the average time for insertion and searching for each query, #points refer to the number of insertion queries

			insert			search		
space/#queries	#points	~#points	HashMap	TreeMap	Quadtree	HashMap	TreeMap	Quadtree
1e4x1e4/1e4	1117	1000	0.00111	4.47002	3.959912	0.000523	0.001341	0.005655
	9999	10000	0.001389	1.52311	2.610935	0.000703	0.000727	0.009827
	93993	100000	0.000268	2.273124	2.770372	0.000717	0.001167	0.008048
	846881	1000000	0.000312	2.056671	1.168611	0.00097	0.001404	0.002738

Table 4-5 clustered test results

We see an average insertion time for each structure as:

HashMap	TreeMap	Quadtree
0.00077	2.580731	2.627458

Table 4-6 clustered test average insert

An average search time:

HashMap	TreeMap	Quadtree
0.000728	0.00116	0.006567

Table 4-7 clustered test average search

Measuring the average depth of the Quadtree we got values for insertion and searching, those values are showed in this table.

#points	depth	#queries	depth

98	5.214286	196	4.612245
963	6.738318	1836	6.246732
3940	7.565482	6338	7.179867
4986	8.017449	9941	7.458505
9942	8.540837	19800	7.953535
95172	10.01189	181320	9.499906
200000	10.72746	400000	10.13321
500000	11.38228	1000000	10.79103
1000000	11.88266	2000000	11.29326

Table 4-8 clustered test quadtree depth

4.2.2.2.2 Clustered distributed test #2

In this test we have a space of 200000×200000 and a number of points scattered across it according to the clustered distribution. For each test, we have a fixed number of search queries which is 100000.

			Average Time					
			insert			search		
space/#queries	#points	~#points	HashMap	TreeMap	Quadtree	HashMap	TreeMap	Quadtree
2e5x2e5/1e5	1117	1000	0.001151	3.06707	4.6524	0.000133	0.00019	0.0004
	10016	10000	0.001275	1.646762	3.376345	0.000216	0.000253	0.000947
	90750	100000	0.000297	2.052824	2.7087	0.0003	0.000534	0.00081
	918468	1000000	0.00029	2.220613	1.225214	0.000277	0.000939	0.00088

Table 4-9 clustered test results

We see an average insertion time for each structure as:

HashMap	TreeMap	Quadtree
0.000753	2.246817	2.990665

Table 4-10 clustered test average insert

An average search time:

HashMap	TreeMap	Quadtree
0.000232	0.000479	0.000759

Table 4-11 clustered test average search

4.2.2.2.3 Deduction of Clustered distribution tests

In the two previous tests, we see an average insertion time for each structure as:

HashMap	TreeMap	Quadtree
0.000761	2.413774	2.809061

Table 4-12 conclusion average insert

An average search time:

HashMap	TreeMap	Quadtree
0.00048	0.00082	0.003663

Table 4-13 conculsion average search

Measuring the average depth of the Quadtree we got values for insertion and searching, those values are showed in this table.

#points	insert	#queries	seach
1117	8.302596	10000	4.137
9999	9.248725	10000	9.2484
93993	10.58316	10000	10.44134
846881	12.04194	10000	12.0243
1117	8.307073	100000	2.07045
10016	9.258187	100000	4.87115
90750	10.59862	100000	10.38306
918468	12.31203	100000	12.0568

Table 4-14 clustered quadtree average depth

4.2.2.3 *Deduction of insert-search tests*

We can notice some behavior from these results:

- We see that the HashMap structure is a way faster than the others in the operations of insertion and search.

In the following diagram we show the average time for insert upon the three test phases discussed above.

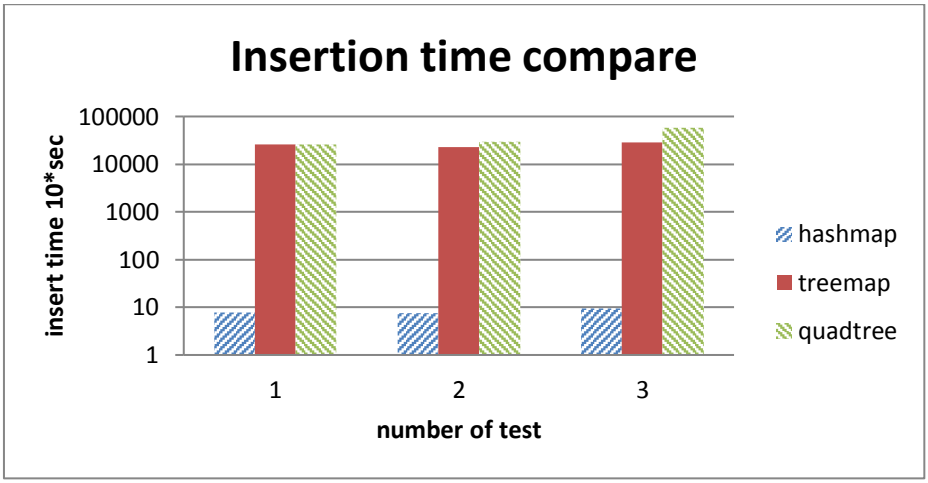


Figure 4-1 Insertion time compare

In the following diagram we show the average time for search upon the three test phases discussed above.

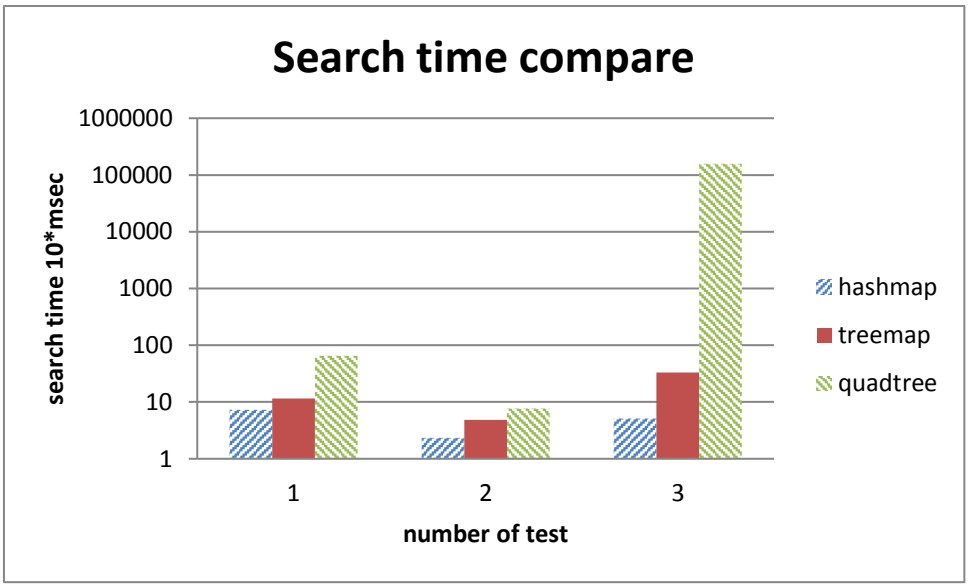


Figure 4-2 search time compare

We can also notice that the search time for a Quadtree in case of uniform distribution is a way larger than the clustered one.

- For most cases the total time of insert and search in the HashMap structure remains almost the same, and the average time decreases while the size increases.

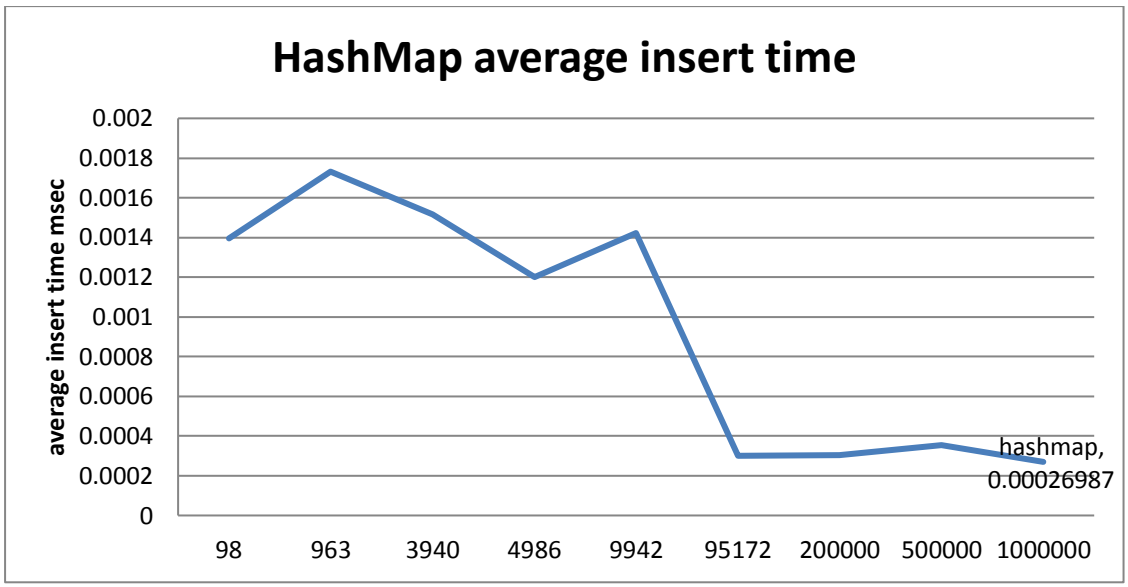


Figure 4-3 HashMap average insert time

- The overhead of rotations in the TreeMap structure appears when comparing its insert time to its search time, we see that the insertion is significantly slower than searching.
- The overhead of pointers creation in the Quadtree structure appears when comparing its insert time to its search time, we see that the insertion is significantly slower than searching.

- The depth of the Quadtree is approximately logarithmic in the size of the Quadtree (number of points) as the tables illustrated and as shown in the following diagram.

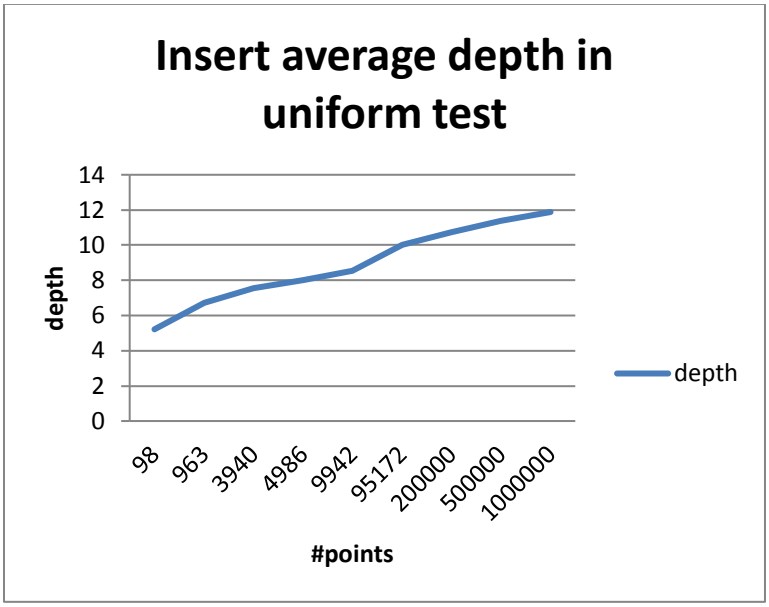


Figure 4-4 Insert average depth in uniform test

4.3 Insert-Range search tests

In these tests we have two types of queries: insert and range search, these queries are stored in an input file. In this file, the two types of queries are separated: first, the insertion queries come and then the range search queries.

4.3.1 Range-tester class

A range-tester class is used to do the same steps stated in 4.2.1 but applied on range search.

4.3.2 Test data and results

Test data also has two types: uniform and clustered, for each of which we run the tester and compare our structures. Ranges sizes follow different distributions in each test.

4.3.2.1 Uniformly-distributed data sets

Points are uniformly distributed across the space, range size also follow uniform distribution of mean α for each of its dimensions.

The following table shows the total time it takes to respond to all the queries.

Range Tests					
α	#queries	~#points	HashMap	TreeMap	Quadtree
100	10	100	4.403331	33.21054	12.08309
100	100	1000	71.6385	183.1102	85.96185
5000	500	5000	304.5696	352.5391	291.3605
5000	1000	20000	15011.95	14209.04	6120.08
100000	1000	100000	9804.735	8177.367	4811.16

100000	10000	100000	79628.03	68237	41979.73

Table 4-15 uniform range result

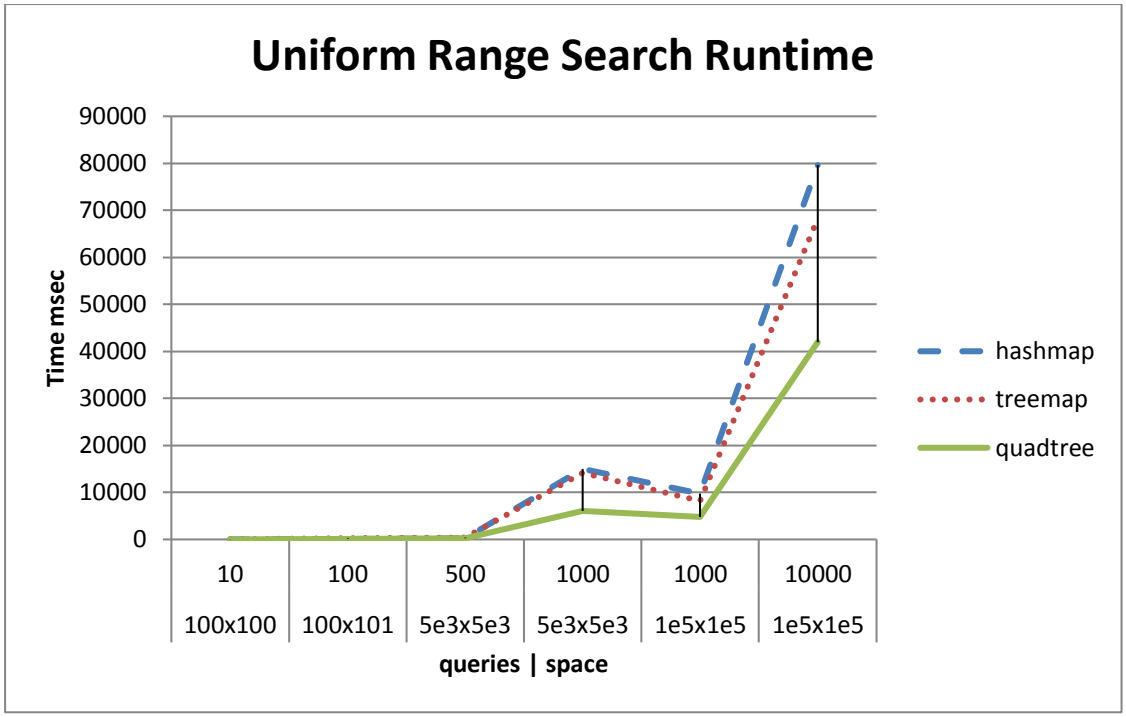


Figure 4-5 Uniform Range Search Runtime

It is obvious that a Quadtree performs better than the other structures, regardless of the average size of the query.

4.3.2.2 Cluster-distributed data sets

Since these data sets are considered more close to real distributions and the main aim of the project is to come up with a good structure to solve orthogonal range search query problem, we look into them in more depth and perform more tests using those distributions.

Range query sizes follow a normal distribution with mean of α and a standard deviation of σ in both of its dimensions (width and height).

Four data sets are used in this section. In each one of them, we have fixed mean and standard deviation of the query size. In all of these tests, the number of queries is 100000.

4.3.2.2.1 Cluster-distributed data set#1

In this set we have $(\alpha = 50, \sigma = 10)$. The following table shows the total time of range search operations for each structure in melliseconds.

#points	HashMap	TreeMap	Quadtree
288	51.38028	54.65016	20.54497
639	78.23715	74.89667	20.96987
4648	815.8888	69.93993	28.47271
8543	1600.813	68.17876	30.46994
45140	11299.89	81.59898	44.22064
94115	37631.49	97.02136	30.32995
472422	214169.8	171.1487	27.16394
932133	448184.5	275.0546	28.05315

Table 4-16 Uniform Range Search Runtime#1

We have an average time of:

HashMap	TreeMap	Quadtree
89229.01	111.5611	28.77815

Table 4-17 Average Uniform Range Search Runtime #1

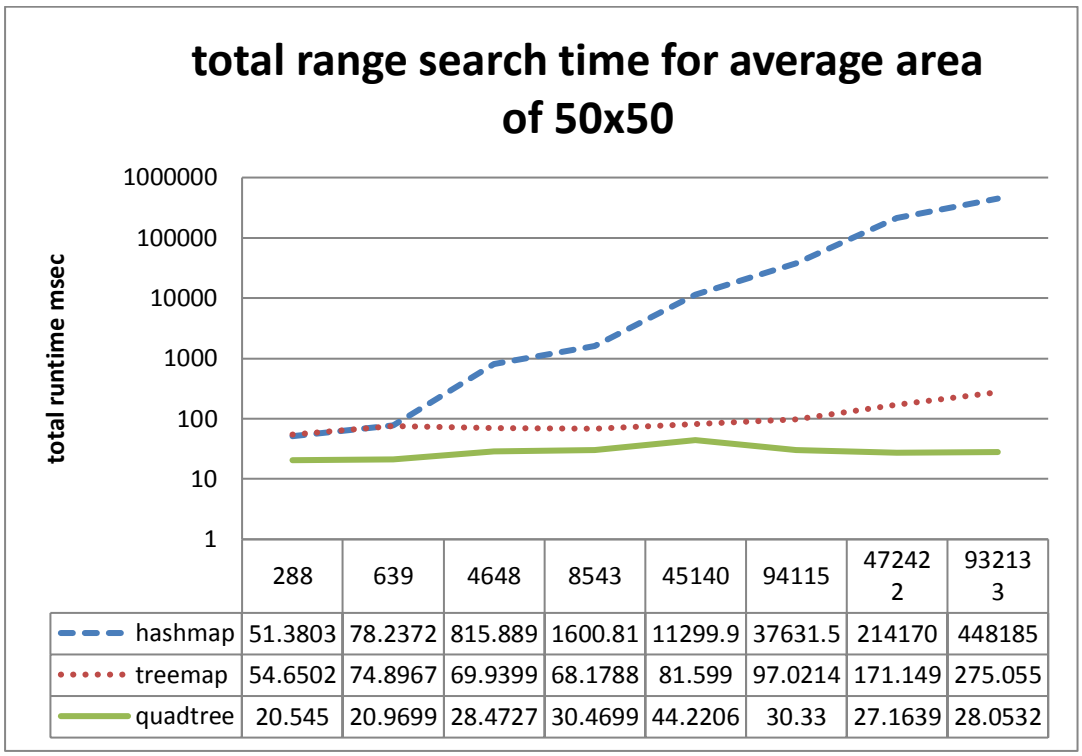


Figure 4-6 total range search time for average area of 50x50

We can easily notice that the Quadtree and the TreeMap perform a way better than the HashMap which is noticed to be linear in the number of points.

4.3.2.2.2 Cluster-distributed data set#2

In this set we have $(\alpha = 500, \sigma = 100)$. The following table shows the total time of range search operations for each structure in milliseconds.

#points	HashMap	TreeMap	Quadtree
288	48.12232	61.62342	20.09872
639	77.20672	90.66841	39.67854
4648	812.1386	56.73893	30.46214
8543	1566.876	118.6608	32.00902
45140	11077.41	154.9126	50.07767
94115	32389.48	258.5862	35.84173
472422	216769.3	989.8936	44.39101
932133	450848.7	2433.926	74.34574

Table 4-18 Uniform Range Search #2

We have an average time of:

HashMap	TreeMap	Quadtree
89198.66	520.6263	40.86307

Table 4-19 Average Uniform Range Search #2

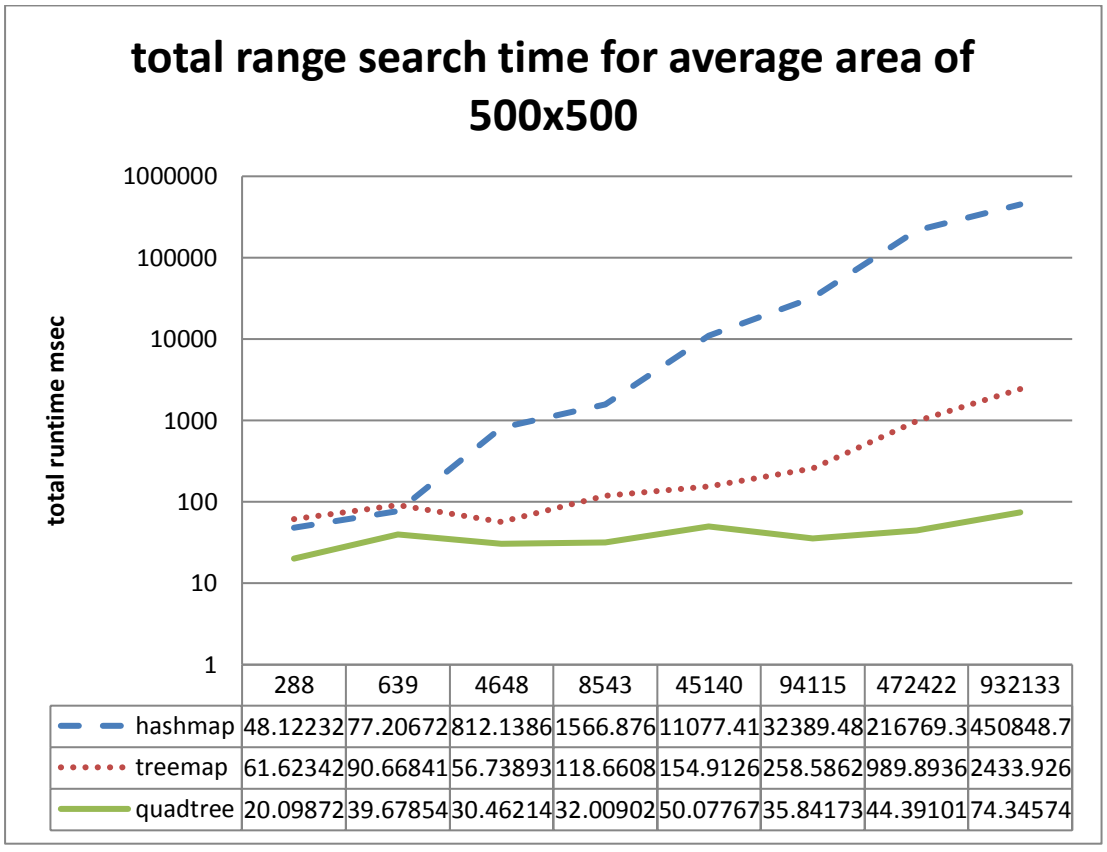


Figure 4-7 total range search time for average area of 500x500

The HashMap performance approximately remains the same while the time of other structures increased in total.

4.3.2.2.3 Cluster-distributed data set#3

In this set we have ($\alpha = 5000, \sigma = 1000$). The following table shows the total time of range search operations for each structure in melliseconds.

#points	HashMap	TreeMap	Quadtree
288	48.12232	61.62342	20.09872
639	77.20672	90.66841	39.67854
4648	812.1386	56.73893	30.46214
8543	1566.876	118.6608	32.00902
45140	11077.41	154.9126	50.07767
94115	32389.48	258.5862	35.84173
472422	216769.3	989.8936	44.39101
932133	450848.7	2433.926	74.34574

Figure 4-8 Uniform Range Search#3

We have an average time of:

HashMap	TreeMap	Quadtree
49229.28	1832.095	308.2739

Table 4-20 Average Uniform Range Search#3

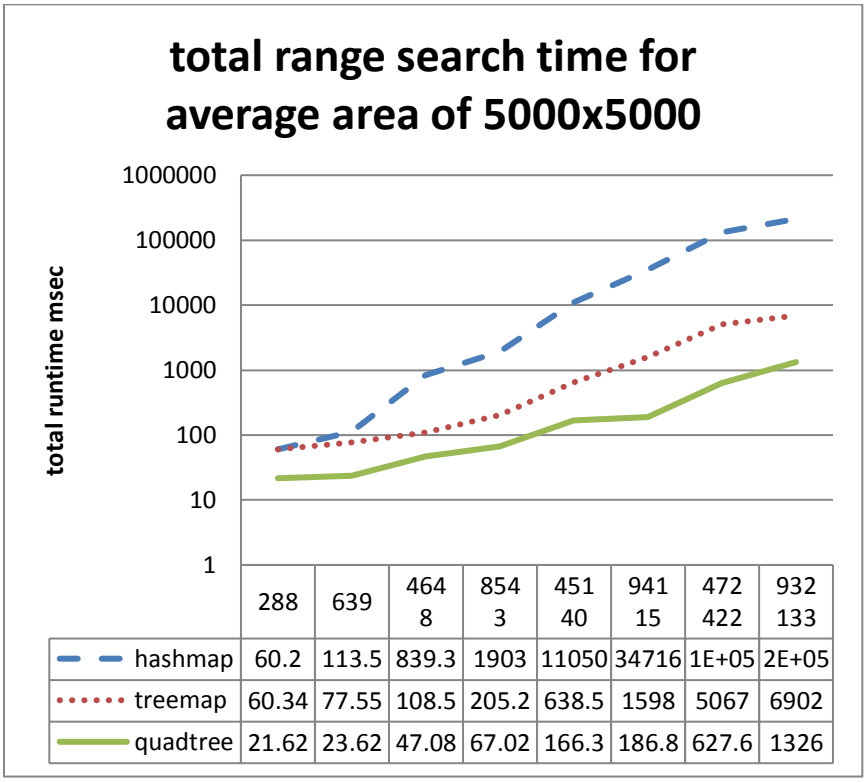


Figure 4-9 total range search time for average area of 5000x5000

The same thing appears again: HashMap performance remains the same while others' decreases.

4.3.2.2.4 Cluster-distributed data set#4

In this set we have huge queries which reaches the quarter of the space ($\alpha = 50000, \sigma = 10000$). The following table shows the total time of range search operations for each structure in melliseconds.

#points	HashMap	TreeMap	Quadtree
31	39.53727	86.45719	32.40268
688	229.2864	373.4086	199.1421
8543	2421.681	2451.032	1870.646
94115	27788.66	31359.51	22317.01
932133	291370.4	427916.1	308857.5

Table 4-21 Uniform Range Search#4

We have an average time of:

HashMap	TreeMap	Quadtree
64369.92	92437.31	66655.34

Table 4-22 Average Uniform Range Search#4

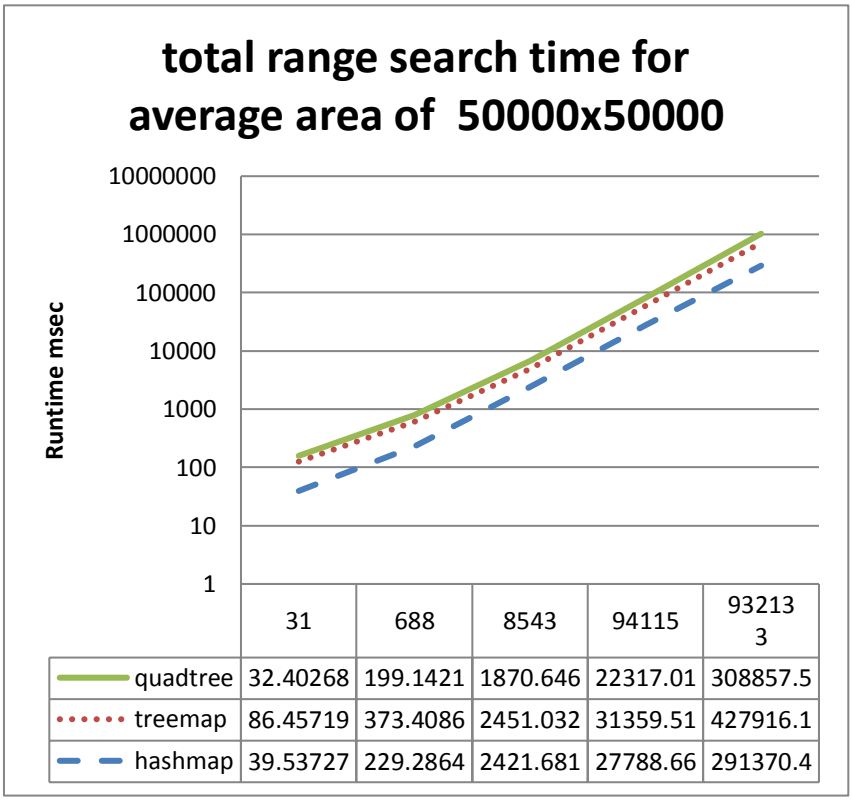
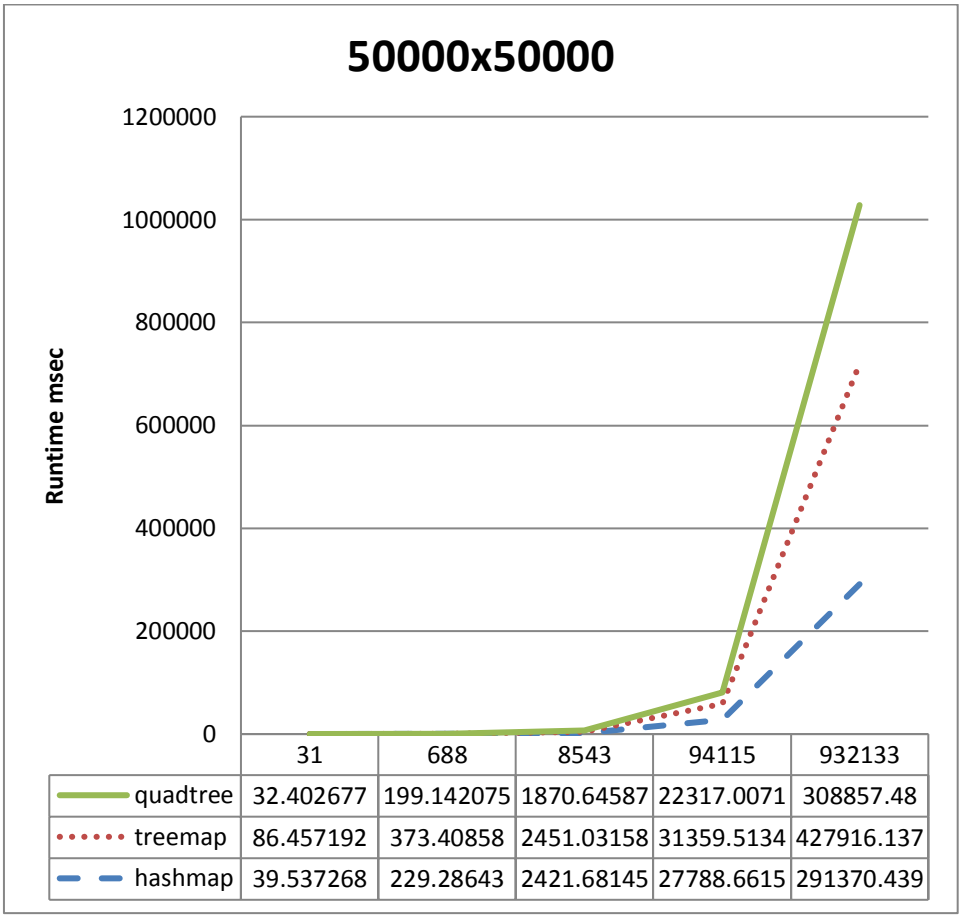


Figure 4-10 total range search time for average area of 50000x50000

Like what was going in the previous tests, Quadtree and TreeMap performance keeps decreasing and reaches a very bad point in which it becomes worse than HashMap's performance.

4.3.2.3 Deduction of insert-range search tests

- In general, we noticed that the Quadtree performs better than both structures in range search. However, for a huge query, we notice that both Quadtree and TreeMap perform worse, because of the large size of the result which overruns the speed detection of the nodes needed to be returned. This problem follows the linearity in the size of the answer stated in a previous section as k, see chapter 2.
- To show the difference between the structures in range search for huge range queries, we again show the diagram of the huge query, but this time with a linear axis for the runtime instead of the logarithmic in the previous one.



	31	688	8543	94115	932133
quadtree	32.402677	199.142075	1870.64587	22317.0071	308857.48
treemap	86.457192	373.40858	2451.03158	31359.5134	427916.137
hashmap	39.537268	229.28643	2421.68145	27788.6615	291370.439

Figure 4-11 total range search time for average area of 50000x50000 linear

- The time for range searching in the case of a HashMap is independent of the size of the board. It is just linear in the size of the data set. The following diagram shows the values of the runtime upon the change of the size of the data set and the space size.

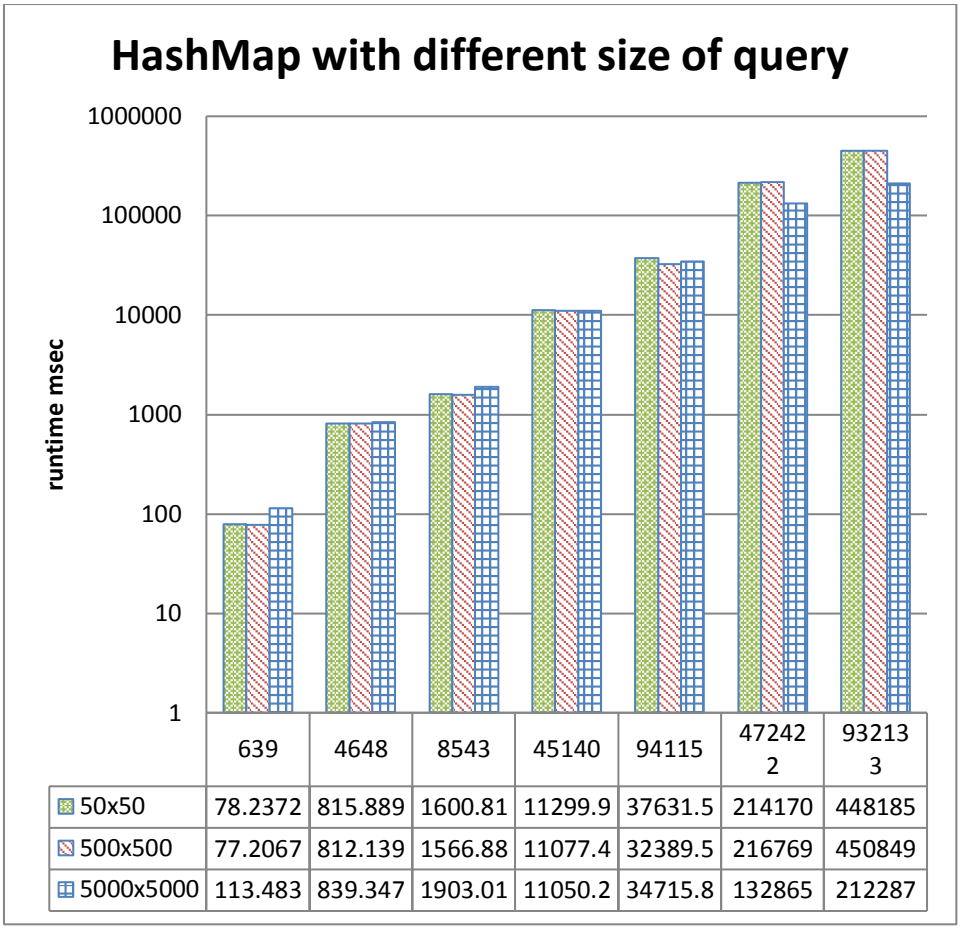


Figure 4-12 HashMap with different size of query

4.3.2.4 *TreeMap worst case*

As stated in section 2.2.2.2.1,the worst case of the treemap appears when the width is too big comparing to the height. However, we generated test cases in which this case happens and tested it on treemap and quadtree to see the results.

Size(100x)	treemap	quadtree
5000	1620.856	55.8237
10000	2601.983	73.29827
20000	4825.287	96.52475
40000	8933.011	142.761
80000	17837.45	221.3468
160000	4165.863	86.17327

Table 4-23 TreeMap worst

The results of comparing, showed in the table above, show really that it the treemap performs slower in range search in this case comparing to the quadtree. TreeMap conducts many operations to test some points which are not in the queried range.

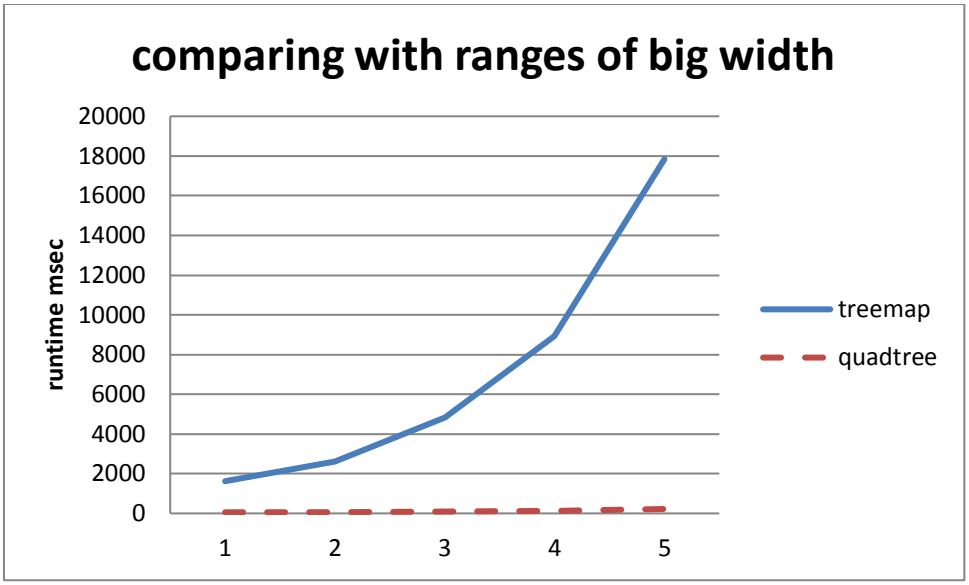


Figure 4-13 TreeMap worst

5 Profiling

In this chapter, we show the memory and CPU usage by each structure on the tests of range queries which are conducted. For each structure, we show the results of three major tests. Tests have query sizes of (100x100,1000x1000,10000x10000) respectively, a size of 1000000 points (insertions).

These tests were chosen because they are large enough (considering the number of insertions), and show how results differ across several sizes for a query.

5.1 CPU usage

The portion of time used by the most time-consuming methods is shown properly in this section, and is compared between considered structures.

We include screenshots taken from Netbeans profiler after conducting the three tests discussed above.

5.1.1 HashMap CPU usage

The tests conducted result in these graphs:

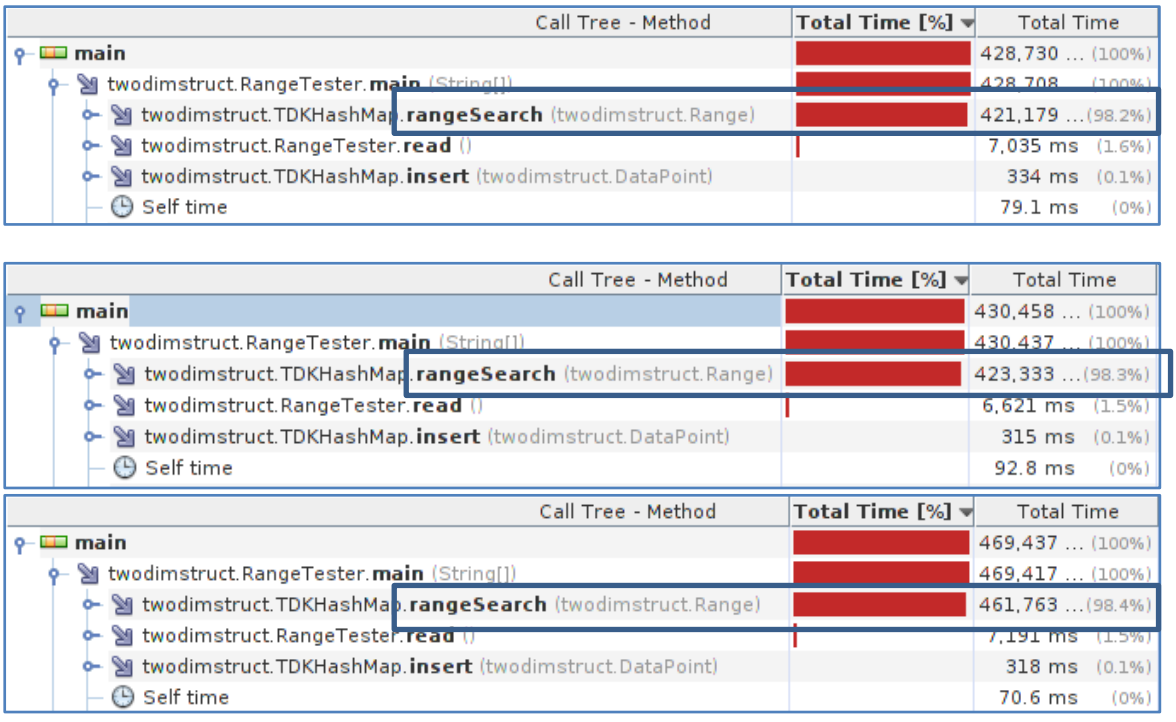


Figure 5-1 hashmap cpu usage

In this case, the time for range search overruns all other methods and becomes almost equal to the total running time.

5.1.2 TreeMap CPU usage

Call Tree - Method	Total Time [%]	Total Time	Invocations
main		6,013 ms (100%)	1
twodimstruct.TDKTreeMap.**insert** (twodimstruct.DataPoint)		5,546 ms (92.2%)	932133
twodimstruct.TDKTreeMap$1.**compare** (Object, Object)		3,815 ms (63.5%)	16837632
Self time		1,636 ms (27.2%)	932133
twodimstruct.TDKStruct.**isOut** (twodimstruct.Point)		68.4 ms (1.1%)	932133
twodimstruct.Point.**<init>** (int, int)		26.4 ms (0.4%)	889443
twodimstruct.TDKTreeMap.**rangeSearch** (twodimstruct.Range)		692 ms (11.5%)	9945
Self time		401 ms (6.7%)	9945
twodimstruct.TDKTreeMap$1.**compare** (Object, Object)		151 ms (2.5%)	462772
twodimstruct.Point.**fallsWithin** (twodimstruct.Range)		123 ms (2.1%)	2159244

Call Tree - Method	Total Time [%]	Total Time	Invocations
main		38,347 ... (100%)	1
twodimstruct.TDKTreeMap.**rangeSearch** (twodimstruct.Range)		33,113 ... (86.4%)	10000
twodimstruct.TDKTreeMap.**insert** (twodimstruct.DataPoint)		5,551 ms (14.5%)	932133
twodimstruct.TDKTreeMap.**<init>** (int, int, int, int)		0.998 ms (0%)	1

Call Tree - Method	Total Time [%]	Total Time	Invocations
main		38,347 ... (100%)	1
twodimstruct.TDKTreeMap.**rangeSearch** (twodimstruct.Range)		33,113 ... (86.4%)	10000
twodimstruct.TDKTreeMap.**insert** (twodimstruct.DataPoint)		5,551 ms (14.5%)	932133
twodimstruct.TDKTreeMap.**<init>** (int, int, int, int)		0.998 ms (0%)	1

Figure 5-2 treemap cpu usage

We can notice that the larger query size is, the more portion of time is consumed by the range search method. This portion becomes to be greater than the portion consumed by the insert method.

5.1.3 Quadtree CPU usage

The same thing noticed in the TreeMap case also appears here: the larger query size is, the more portion of time is consumed by the range search method.

Call Tree - Method	Total Time [%]	Total Time	Invocations
main		8,221 ms (100%)	1
twodimstruct.TDKQuadtree.**insert** (twodimstruct.DataPoint)		8,215 ms (99.9%)	932133
twodimstruct.TDKQuadtree.**rangeSearch** (twodimstruct.Range)		244 ms (3%)	9690

Call Tree - Method	Total Time [%]	Total Time	Invocations
main		15,094 ... (100%)	1
twodimstruct.TDKQuadtree.**insert** (twodimstruct.DataPoint)		14,420 ... (95.5%)	932133
twodimstruct.TDKQuadtree.**rangeSearch** (twodimstruct.Range)		1,220 ms (8.1%)	10000
twodimstruct.Point.**<init>** (int, int)		85.1 ms (0.6%)	932133
twodimstruct.Range.**<init>** (int, int, int, int)		2.93 ms (0%)	10000
twodimstruct.TDKQuadtree.**<clinit>**		2.20 ms (0%)	1
twodimstruct.TDKQuadtree.**<init>** (int, int, int, int)		0.342 ms (0%)	1
twodimstruct.TDKQuadtree.**disableStats** ()		0.022 ms (0%)	1

Call Tree - Method	Total Time [%]	Total Time	Invocations
main		14,442 ... (100%)	1
twodimstruct.TDKQuadtree.**rangeSearch** (twodimstruct.Range)		7,581 ms (52.5%)	9976
twodimstruct.TDKQuadtree.**insert** (twodimstruct.DataPoint)		7,085 ms (49.1%)	932133
twodimstruct.Point.**<init>** (int, int)		33.8 ms (0.2%)	932133
twodimstruct.TDKQuadtree.**<clinit>**		4.83 ms (0%)	1
twodimstruct.Range.**<init>** (int, int, int, int)		1.37 ms (0%)	9976
twodimstruct.TDKQuadtree.**<init>** (int, int, int, int)		0.460 ms (0%)	1
twodimstruct.TDKQuadtree.**disableStats** ()		0.014 ms (0%)	1

Figure 5-3 quadtree cpu usage

Another note is the call stack which is formed by the recursive call of the insert method. This recursive call causes more running time for the insert method, but we can consider that this effect is insignificant since we showed in the previous chapter that the depth of the quadtree grows logarithmically in tree size for reasonable tests.

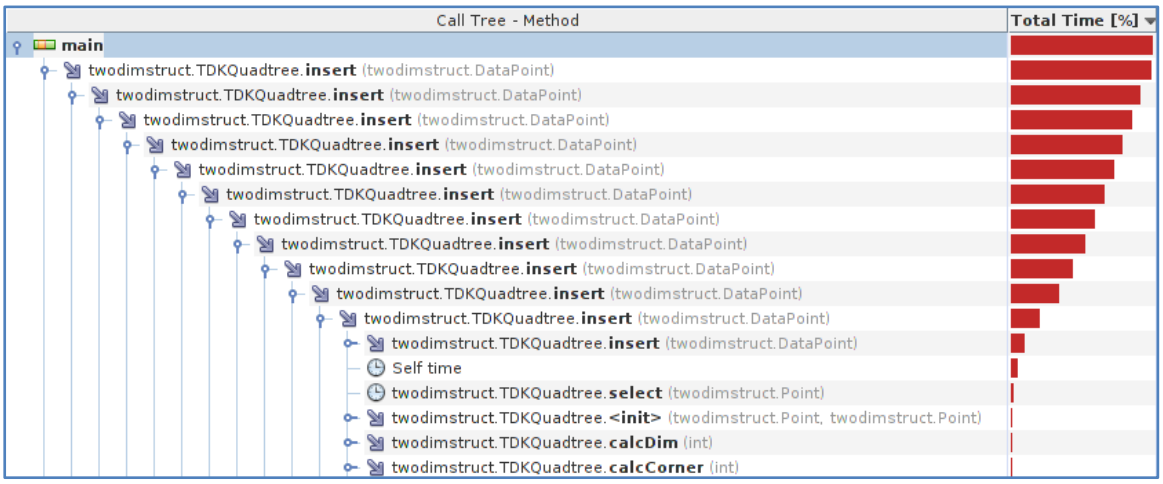


Figure 5-4 quadtree stack

5.2 Memory usage

Here we show the maximum heap used by each structure. These diagrams show the running for each test. The used heap means the used memory by the application.

5.2.1 Hashmap memory usage

On these diagrams we show the start of answering process.

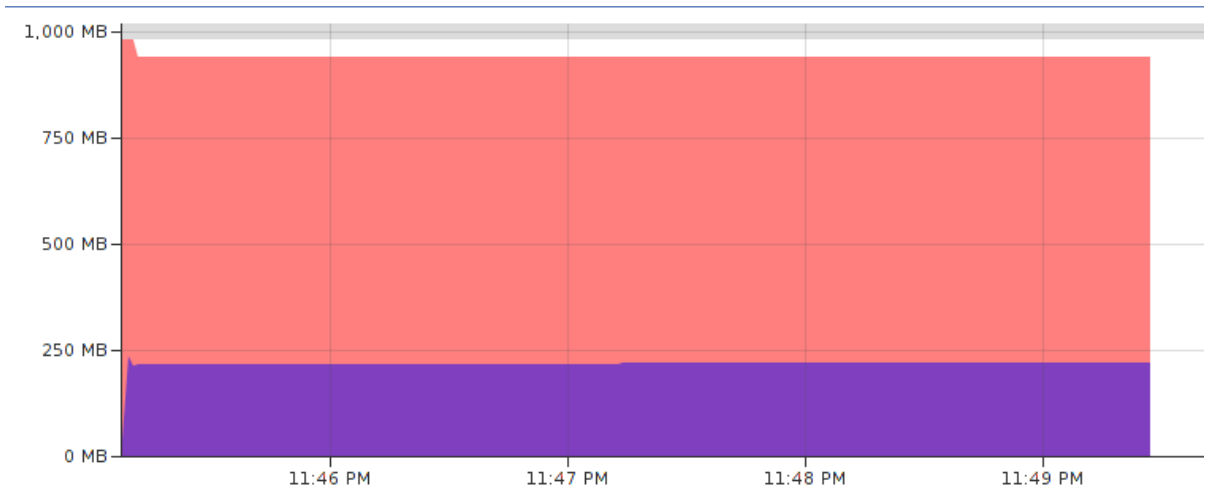


Figure 5-5

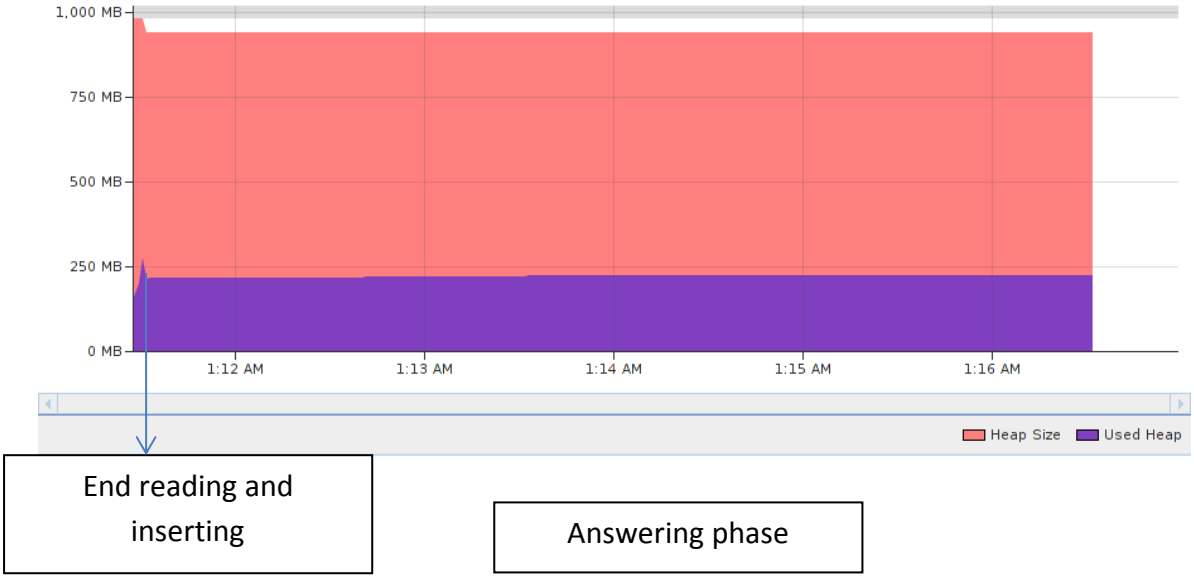


Figure 5-6

- We can notice that the hashmap pre-allocates the needed memory before inserting.
- Maximum heap doesn't exceed 250MB.

5.2.2 TreeMap memory usage

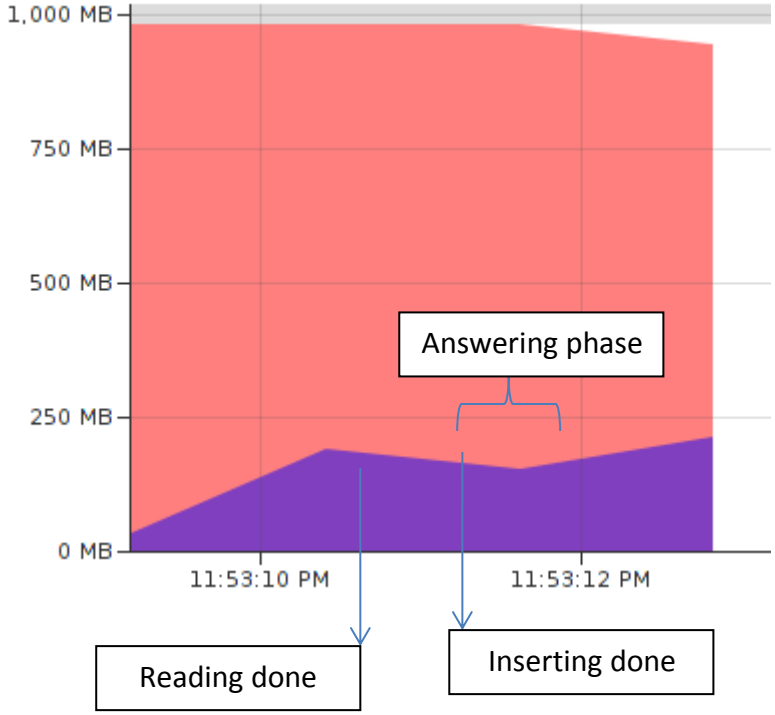


Figure 5-7

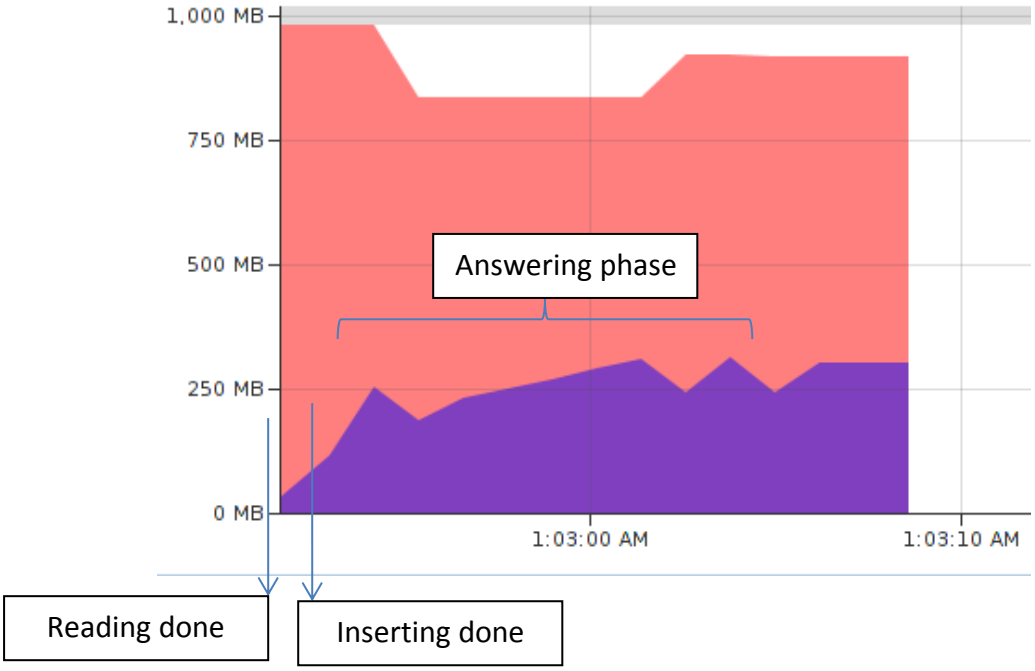


Figure 5-8

- Those diagrams show us a bit more memory usage than the hashmap.
- After building the tree, more memory could be allocated using the range search operations.

5.2.3 Quadtree memory usage

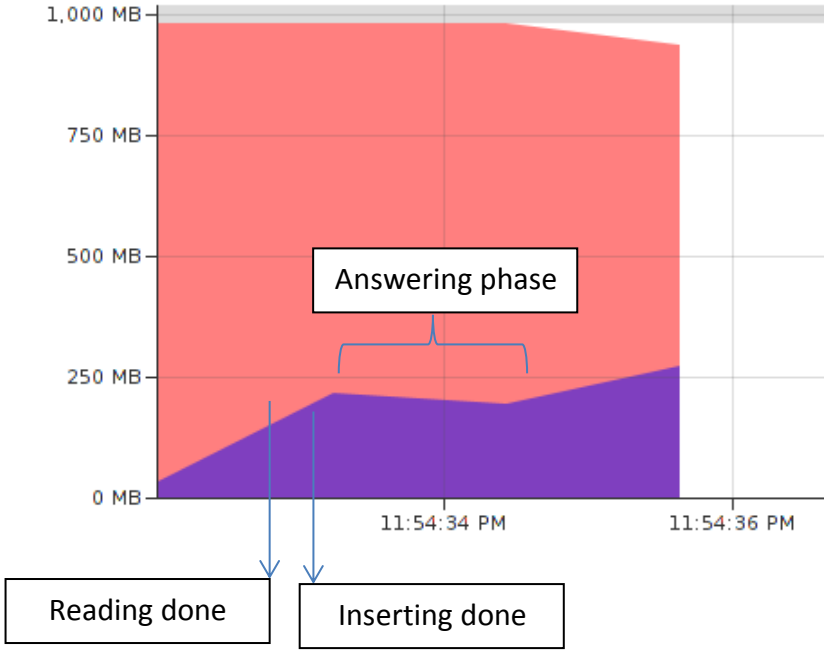


Figure 5-9

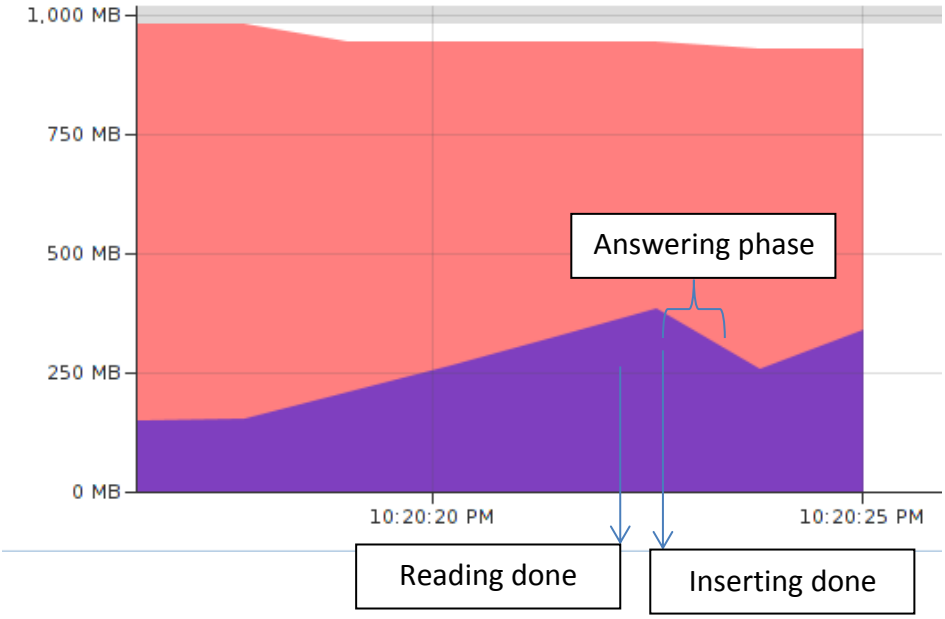


Figure 5-10

- This structure also takes a bit more memory than the hashmap.

5.2.4 Results

- There is no significant difference between the three methods regarding the memory usage.
- HashMap structure reserves the needed space before starting of insertion, while the other two tend to be more dynamic in acquiring and releasing the memory.
- The increment in memory usage for the other two structures may result out of the overhead needed to implement linked trees rather than hash functions in HashMaps.

6 Results and challenges

6.1 Results

After showing the points in which we tested structures in details, we show the main results we can deduce from the testing and profiling phases:

- We can deduce that for a dynamic structure with just insert and search operations and multidimensional key, hash tables do better based on all considerations (memory, CPU, implementation simplicity).
- For a dynamic structure with operations of insert and range search (both operations of approximately similar range) and having a multidimensional key, we see that a quadtree performs a way better than the other data structures proposed in respect to running time.
- If the rate of the range search operations was low, a red black tree could perform better because we saw that the slow part of the execution was the range search part.
- For huge query sizes, it is better to use the simplest structure i.e. hash table. The simplicity of this structure omits the overhead of pointers creation that we can see in the other ones.
- Hash tables are the best at memory usage, they preserve the needed memory on building phase and before insertion. However the difference in memory usage is not that significant difference.
- A quadtree structure, could be very suitable for applications that needs spatial analysis. We proposed ideas for applications that could use a quadtree: meteorology, recommendation services, web searching, etc.

6.2 Challenges

Several hardware limitations prevented us from performing easier job to get this accuracy in results and maybe more accuracy.

We weren't able to carry out some tests due to the memory limitations. Several tests took a big amount of time and it was a waste of power and time. However, we tried to overcome these limitations by increasing the number of test cases to get more accurate results.

7 Future prospects

This study can be conducted on larger data in a larger environment, this allows us to reduce the number of tests which were carried out in this study and maybe to get more accurate results by carrying out the same amount of tests.

We can make use of these structures in some big applications.

This study can also be generalized to include more complex structures which can solve the problem.

As we can see from the discussion in previous chapters, each structure can be parallelized to get better performance and faster execution. More precisely, range searching in a quadtree structure depends on tree search and needs multiple paths to be considered in searching and then reduced into a result variable. Red-black trees and hash tables have also a chance to be executed in parallel.

8 Project limitations

In this project, we dealt with the structures (hash tables, red-black trees, quadtrees) with operations of (insert, search, orthogonal range search).

We didn't study deletion because it is considered to be trivial since the other basic operations were discussed. Non-orthogonal range search wasn't studied because it is related to some advanced geometric level besides that orthogonal range search is considered to give approximate results to nin-orthogonal's.

9 Appendix A. Deeper look at the implementation

In this appendix we review some important parts of the code which is written in Java. To see the basic design and implementation, you can refer to chapter 3.

9.1 Tester Class

In the main method of this class, we read input file, process insert queries and process search queries. While processing the queries we record some statistics in `TDKQuadtree.stats`.

```
public static void main(String [] args) throws FileNotFoundException {
        TDKStruct container = new TDKQuadtree(x0,y0,width,height);
        read(); // read input file
        TDKQuadtree.enableStats();
        TDKQuadtree.initializeStats();

        insertionTime = System.nanoTime();
        for (int i = 0; i < n; i++) { // insert all
            container.insert(all[i]);
        }
        insertionTime -= System.nanoTime();
        insertionTime = -insertionTime;

        searchTime = System.nanoTime();
        for (int i = 0; i < qn; i++) { // search queries
            allData[i] = container.find(qx[i], qy[i]);
        }
        searchTime -= System.nanoTime();
        searchTime = -searchTime;

        averageDepthFind = TDKQuadtree.stats.get(StatTag.depth_accum) / qn;
        // qn is the number of queries
        write();

        report();

    }
```

9.2 Range Tester Class

This class is very similar to tester class with a little difference which is:

- It doesn't record any statistics.
- Processing the search queries is different since the types of queries are different.

```
            // the searching section becomes like this
            searchTime = System.nanoTime();
            for(int i=0;i<qn;i++){
                Set s = container.rangeSearch(new Range(qx1[i], qy1[i], qx2[i],
                                                        qy2[i]));
                allData[i] = s.size();
            }
            searchTime -= System.nanoTime();
            searchTime = - searchTime;
```

9.3 TDKQuadtree.insert()

This method is responsible for inserting a node into a quadtree instance. The method follows a recursive style and this is the one adapted in the testing phase.

TDKQuadtree.updateStats() was not included in this copy of the code because here we consider that statistics is turned off

```
        public boolean insert(DataPoint<T> p) {
        int idx = select(p),thisidx = select(this.p);
        if (isleaf && this.p == DataPoint.NONE) { // empty leaf
            this.p = p;
            return true;
        }
        if(isOut(p)){ // out of range
            return false;
        }
        if (isleaf) {
            if (p.equals(this.p)) {
                return true; // already in
            }
            if (child[thisidx] == null) { // no child already here
                child[thisidx] = new TDKQuadtree(
                                    calcCorner(thisidx), calcDim(thisidx));
            }
            child[thisidx].insert(this.p);
            this.p = DataPoint.NONE;
        }
        if (child[idx] == null) { // split the two points
            child[idx] = new TDKQuadtree(calcCorner(idx), calcDim(idx));
        }
        child[idx].insert(p);
        isleaf = false;
        return true;
    }
```

9.4 IterativeQuadtree.insert()

Much simpler and faster implementation of the insert method is the iterative version of the insert method. This version is considered to be little more faster since it gets rid of the overhead of context switching.

```
    public boolean insert(DataPoint<T> P) {
        int idx;
        if (isOut(P)) {
            return false;
        }
        TDKQuadtree<T> q = this;
        idx = q.select(P);
        while (q.getChild(idx) != null) {
            q = q.getChild(idx);
            idx = q.select(P);
        }
        TDKQuadtree newq = new TDKQuadtree(q.calcCorner(idx), q.calcDim(idx));
        q.setChild(idx, newq);
        return true;
    }
```

9.5 TDKQuadtree.rangeSearch()

This method is responsible for searching for the set of node which exists in a given range. This version of the method follows a iterative style and is the one adapted in the testing phase.

```
    public HashSet<DataPoint<T>> rangeSearch(Range R) {
        HashSet<DataPoint<T>> ret = new HashSet<>();
        MyStack<TDKQuadtree> stack = new MyStack<> ();
        stack.push(this);
        TDKQuadtree q;
        while(!stack.empty()){
            q = stack.pop();
            if(q.isLeaf()
                        && q.getPoint()!=Point.NONE
                        && q.getPoint().fallsWithin(R)){
                ret.add(q.getPoint());
            } else {
                for(TDKQuadtree c : q.getChildren()){
                    if(c !=null && c.getRange().intersects(R)) {
                        stack.push(c);
                    }
                }
            }

        }

        return ret;
    }
```

9.6 TDKQuadtree.recursiveRangeSearch()

Another version of the range search method which have an overhead with the context switching. We include it because it is easier to understand.

```
    private void recursiveRangeSearch(Range R,TDKQuadtree q){
        if(q.getRange().intersects(R)){
            if(q.isLeaf()
                   && q.getPoint() != DataPoint.NONE
                   && q.getPoint().fallsWithin(R)){
                current.add(q.getPoint());
            } else {
                for(TDKQuadtree c : q.getChildren()){
                    if(c !=null) {
                        recursiveRangeSearch(R, c);
                    }
                }
            }
        } }
```

9.7 Stats class

This abstract class contains the basic operations needed for updating statistics. These basic operations are:

1- Set: to change a value of a certain measure.
2- Add: to accumulate values to a certain measure
3- Get: to know the value of a certain measure

We have two static instances of this class:

1- `DISABLED`: in which we do nothing. It is just like turning the statistics off. In this case there is no code for both set and add operations.
2- `ENABLED`: the code for both operations does what it is expected to do.

```
        public static final Stats ENABLED =  new Stats(){
                @Override
                public void add(StatTag tag, Double value) {
                    if(containsKey(tag)){
                        put(tag,value + get(tag));
                    } else {
                        put(tag,value);
                    }
                }

                @Override
                public void set(StatTag tag, Double value) {
                    put(tag,value);
                }
        };
```

Changing the statistics state in the Quadtree class is by calling either disableState() or enableState(). The code for these methods just changes the reference of the statistic instance to either `ENABLED` or `DISABLED`.

```
    static public void enableStats(){
        stats = Stats.ENABLED;
    }

    static public void disableStats(){
        stats = Stats.DISABLED;
    }
```

10 Appendix B. Test cases generation

In this appendix and for each type of tests, we include the methods in which we generated the test cases, then we see the code for test generators. Moreover, we include visualization for some distributions in tests. We consider 3 types of queries: inser, find, range search. The test generators are implemented using C++.

10.1 Uniform distribution

10.1.1 Uniformly distributed insert-find tests

The insert queries have a random number generator (RNG) and follow a uniform distribution in which x is uniformly distributed across the range $[0, nd]$, and y is also distributed uniformly across $[0, nd]$.

(nd is illustrated later in the code)

Search queries follow the same distribution in which x and y fall in the same range.

The code for generating test cases is included briefly to walk deeper inside the test generation phase which gave us the results discussed in the testing chapter. It performs initialization, writing insert queries, writing search queries and writing answers for search queries.

```
int main(int argc,char *argv[]){
	freopen(argv[2],"w",stdout);// input file
	ofstream fout(argv[3]);      // expected output file

	int nd = parsInt(argv[0]); // the limit of the space size
	int cnt = parsInt(argv[1]);// the total number of the points

	default_random_engine generator; // RNG
	uniform_int_distribution<int> distribution(st,nd); // distribution

	// insert queries
	int n = cnt;
	for(int i=0;i<cnt;i++){
		int a = distribution(generator);
		int b = distribution(generator);
		if((a,b) already generated){
		   n--;
		   continue;
		}
		string s = generateString();
		v.insert(make_pair(a,b));
		cout<<a<<" "<<b<<" "<<s<<endl; // insert query to input file
		fout<<s<<endl;                 // answer to the output file
	}
	cout<<"-1 -1"<<endl;// end of insert queries




	// search queries
```

```
        for(int i=0;i<n;i++){
         cout<<v1[i]<<" "<<v2[i]<<endl; // search query
        }
        for(int i=0;i<n;i++){ // search queries that have no answer
               int a = distribution(generator);
               int b = distribution(generator);
               if(v.count(make_pair(a,b))){
             continue;
               }
               cout<<a<<" "<<b<<endl; // search query
               fout<<"Not Found"<<endl; // its answer
        }
        cout<<"-1 -1"<<endl; // end of all queries
    return 0;
}
```

10.1.2 Uniformly distributed range search tests

The insert queries have a RNG and follow a uniform distribution in which x is uniformly distributed across the range $[0, nd]$, and y is also distributed uniformly across $[0, nd]$.

Range search queries follow the same distribution in which x and y fall in the same range. It is assumed that the answer for a range query is the number of points that fall within this range

The code for generating test cases is included briefly. It performs initialization, writing insert queries, writing range search queries and writing answers for search queries.

```
int main(int argc,char *argv[]){
        freopen(argv[0],"w",stdout);// input file
        ofstream fout(argv[1]);     // expected output file

        default_random_engine generator; // RNG
        uniform_int_distribution<int> distribution(st,nd); // distribution

        int nd = parseInt(arg[2]); // limit
        int n =        parseInt(arg[3]); // points count
        int q =  parseInt(arg[4]); // range queries
        for(int i=0;i<n;i++){
               int a = distribution(generator); // generate
               int b = distribution(generator);
               string s = generateString();
               m[(a,b)]=s; // m is a map:(pair -> string)
              cout<<a<<" "<<b<<" "<<s<<endl; // output query
        }
        cout<<"-1 -1"<<endl; // end of insert queries
        for(int i=0;i<q;i++){// range search queries
               int a[2],b[2];
               a[0] = distribution(generator);
               b[0] = distribution(generator);
               a[1] = distribution(generator);
               b[1] = distribution(generator);
               sort(a);sort(b);
               cout<<a[0]<<" "<<b[0]<<" "<<a[1]<<" "<<b[1]<<endl; // output query
               int P=0;
               for(auto M:m){ // count the number of points inside this range
                      (x,y) = M.entry;
```

```
                    if(x>=a[0] && x<a[1] && y>=b[0] && y<b[1]){
                        P++;
                    }
                }
                fout<<P<<endl; // output query answer
        }
        cout<<"-1 -1 -1 -1"<<endl; // end of search queries
    return 0;
}
```

10.2 'Clustered' distribution

The name 'clustered' comes from dividing the points into groups around some points of interest Points of interest have more than one level, a point of a certain level is close to some point from the previous level.

This distribution works like this: assume we have a number of points of interest scattered uniformly across the space, let's call these points **level 1 points**.

For each point p_i of level 1, we have a number of points q_i scattered close to p_i, the distance form p_i to any of $q_{i,j}$ point follow normal distribution with a certain mean (usually 0) and standard deviation. Let's call these points **level 2 points**.

Level 3 points have a similar distribution to **level 2 points**; a set of points $r_{i,j}$ have a distance from $q_{i,j}$ that follows a normal distribution with a certain mean and standard deviation.

Level 3 points are the actual points that will be included in the output of this process.

An output of this algorithm is shown in the following figure.

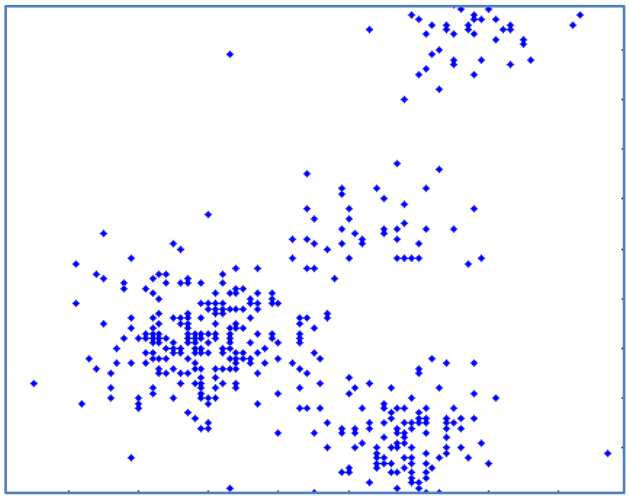


Figure 10-1

This distribution is considered to be closer to the real distribution of the cities across the land: we have points of interests around which we have cities with different distances from those points of interest.

The following figure shows the regions in which we have points. The center of each region may be a point of interest.

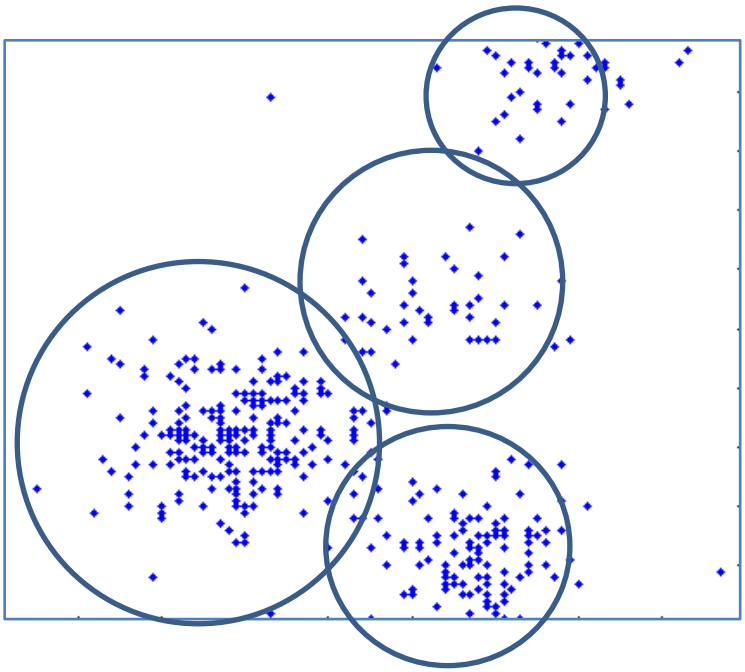


Figure 10-2

To illustrate the idea of levels of points we include this snapshot which is a region taken form a visualization of a generated input file.

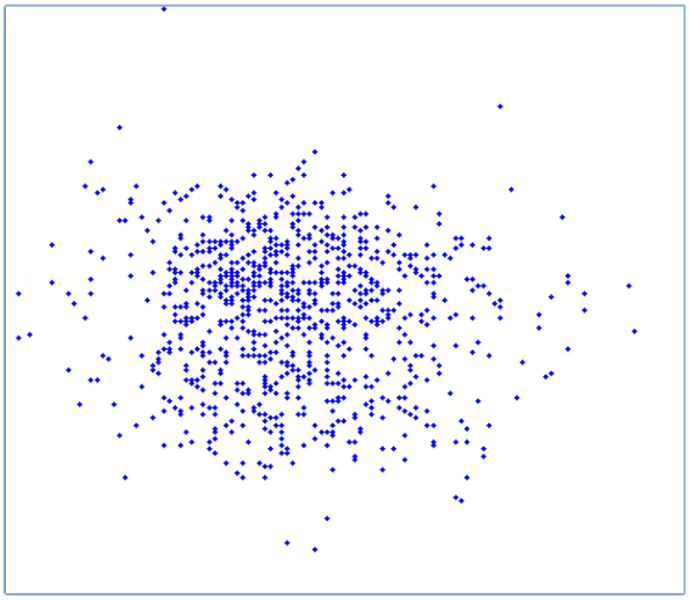


Figure 10-3

In this picture, we can expect a point of **level 1** to be near the center of the region a marked with the big solid point in the middle of the region. Possible regions of level 2 could be the regions denoted by the circles.

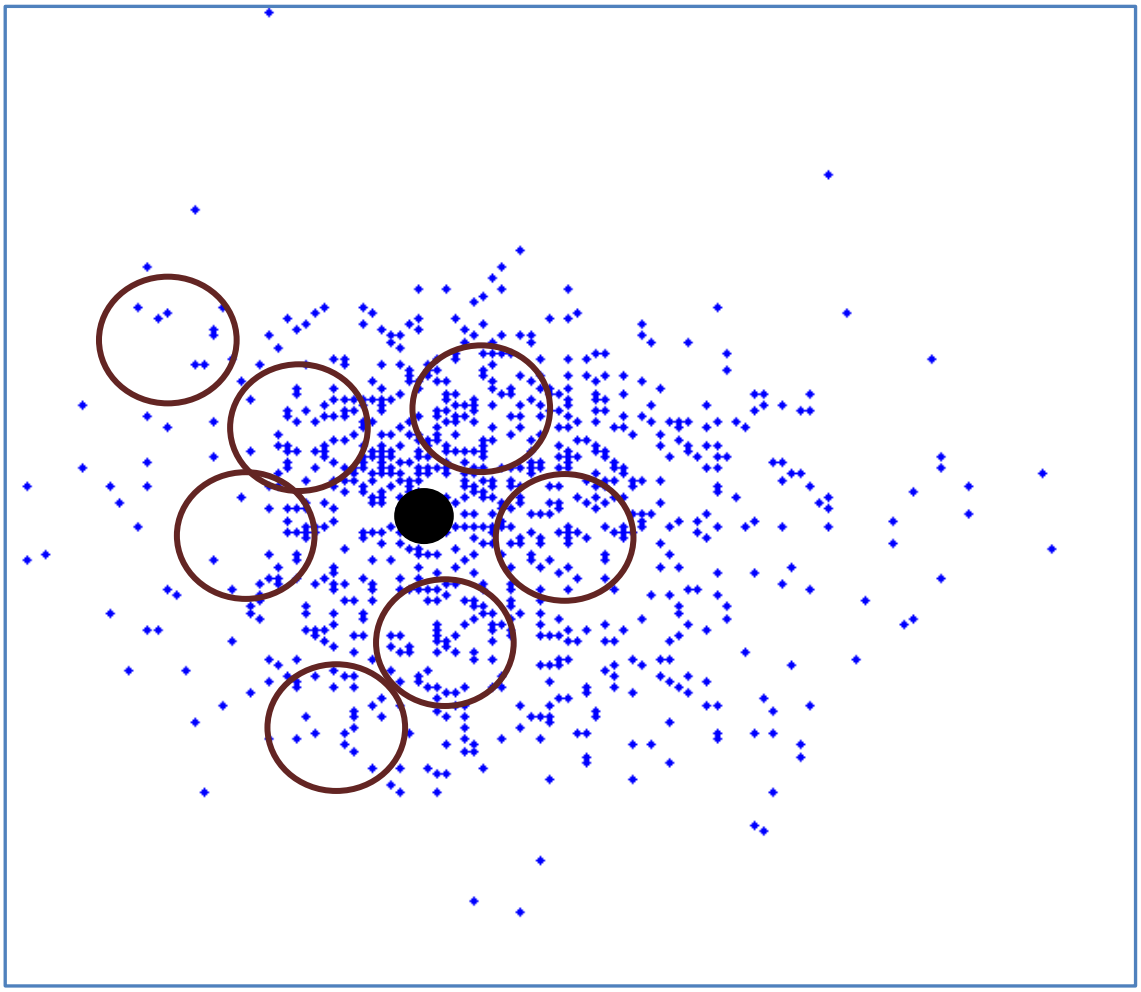


Figure 10-4

This distribution is used to generate insert queries for both insert-search tests and insert-range search tests.

The code used to generate a point p with a distance form another point (a, b) following a normal distribution is included in this function:

```
void generate(double a,double b,
              pair<int, int > &p,
              normal_distribution<double> distribution){
    p.first = min(nd,max(st,(int)(distribution(generator)+a)));
    p.second = min(nd,max(st,(int)(distribution(generator)+b)));
}
```

To generate the points of **level 1,** we used this snippet:

```
int l[1] = city_count_distribution(generator); // number of level 1 points
for(int i=0;i<l[1];i++){
       level[1][x].add(min(nd,max(st,distribution(generator))));
```

```
        level[1][y].add(min(nd,max(st,distribution(generator))));
}
```

To generate the points of **level 2 or more (level *i*)** around point *j* of the previous level, we used this snippet:

```
int l[i] = city_count_distribution(generator);
for(int k=0;k<l[i];k++){
        point p;
        gen(level[i][x][j],level[i][y][j],distribution);
        level[i][x].add(p.x);
        level[i][y].add(p.y);
}
```

10.2.1 Clustered distributed insert-search tests

Follow the same code stated in section 10.1.1 but with a slight difference: the generation of the insert tests follow the rules stated earlier in this section (clustered distribution).

10.2.2 Clustered distributed insert-range search

Also follow the same code stated in section 10.1.2. The generation of the insert tests follow clustered distribution.

10.3 Tables of the test cases generated

10.3.1 Uniform insert-search

Input file#	Space size	Number of points
01	100x100	100
02	100x100	1000
03	100x100	5000
04	1000x1000	5000
05	1000x1000	10000
06	1000x1000	100000
07	2e5x2e5	2.00E+05
08	2e5x2e5	5.00E+05
09	2e5x2e5	1.00E+06

Table 10-1

10.3.2 Uniform insert-range search

Input file#	Space size	Number of points	Average range size
01	100x100	100	10
02	100x100	1000	100
03	5000x5000	5000	500
04	5000x5000	20000	1000
05	100000x100000	100000	1000
06	100000x100000	100000	10000

Table 10-2

10.3.3 Clustered insert-search

space	#points	Number of queries
1e4x1e4	1117	1.00E+04
	9999	
	93993	
	846881	
2e5x2e5/1e5	1117	1.00E+05
	10016	
	90750	
	918468	

Table 10-3

10.3.4 Clustered insert-range search

query size	#points	query size	#points
50x50	288	5000x5000	288
	639		639
	4648		4648
	8543		8543
	45140		45140
	94115		94115
	472422		472422
	932133		932133
500x500	288	50000x50000	31
	639		688
	4648		8543
	8543		94115
	45140		932133
	94115		
	472422		
	932133		

Table 10-4

11 Biblography

[1] N. Wirth, *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, M. London, M. B. Company, and B. B. Ridge, *Introduction to Algorithms, Second Edition*, vol. 7. 2001.

[3] "Quadtree - Wikipedia, the free encyclopedia." [Online]. Available: http://en.wikipedia.org/wiki/Quadtree.

[4] H. Samet, "Hierarchical spatial data structures," *Des. Implement. Large Spat. Databases First Symp.*, pp. 193–212, 1989.

[5] Steven Lambert, "Quick Tip_ Use Quadtrees to Detect Likely Collisions in 2D Space - Tuts+ Game Development Tutorial," 2012. [Online]. Available: http://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374.

[6] V. Truong, "Quadtree Collision Detection," 2012.