Advanced Data Structures: Exercises

Constantino Gómez and Cristobal Ortega
{name.surname}@est.fib.upc.edu

June 2016

Exercises from the first part of the course [3] about Multidimensional data structures and Metric Data structures

1 Preliminaries

1.1 Make a table showing the pros and cons of data structures

Data Structure	Advantages	Disadvantages
Array	• Quick insertion	• Slow search, deletions • Fixed size
Ordered array	• Search: O(log n) • Slow insert/delete: O(n)	• Fixed size
Array	• Insert/delete fast • Dynamic memory allocation • Almost no extra memory needed	• Insert / delete can be only done in the first element • No search without any auxiliary data structure
Queue	• Insert/delete fast • Dynamic memory allocation • Almost no extra memory needed	• Insert / delete can be only done in the first element • No search without any auxiliary data structure
Linked list	• Quick insertion, delete (at the beginning and in the end) • Dynamic memory allocation	• Search O(n) • Needs extra memory for auxiliary variables

Data Structure	Advantages	Disadvantages
BST	• Search, insert, delete: O(log n) • Dynamic memory allocation	• It can be converted to linked list (no balancing) • Needs extra memory for auxiliary variables
AVL (BST balanced)	• Search, insert, delete: O(log n) • Dynamic memory allocation	• It has more constant work since every insert / delete we have to check the level of the children and reorganize the structure if needed • Needs extra memory for auxiliary variables
Hash table	• Average insert / search / delete O(1) • Dynamic memory allocation	• Worst case O(n) • To keep efficiency changes in the size of the structure may require to change the hash function • Inefficient for low number of entries
Heap	• Fixed or dynamic array • Find and insert min O(1) in most cases	• Partially ordered, search O(n)

Data Structure	Advantages	Disadvantages
Heap	• Fixed or dynamic array • Find and insert min O(1) in most cases	• Partially ordered, search O(n)
Graph	• Abstract datatype, common implementations are: adjacency list, adjacency matrix, and incidence matrix	• Each implementation favors or penalizes a subset of operations: add vertex/edge, remove vertex/edge

Table 1: Data Structures pros and cons

2 Multidimensional Data Structures

2.1 Count (give) the number of elements (regions) of the partition of the space induced by k-d trees of any kind and of k dimensional quad trees.

- Kd-tree
  - Cells: *n+1*
  - Partitions: *n*
- Quad-tree
  - Cells: *3n+1*
  - Partitions: *2n*

2.2 Propose a node definition of relaxed k-d trees (in c++) together with the operations of insertion of a new element, partial match and orthogonal range searches. How could you implement deletions?

```
1    struct node {
2        type Key;
3        type discriminant;
4        int size;
5        node *left, *right;
6    };
```

Listing 1: Class Node of relaxed kd-trees

```
relax_kdtree insert(relax_kdtree t, type& x) {
    if ( t->size == 0)
        type nd = rand(0,dimensions); //not sure
        return insert_at_root(t, x, nd);
    else {
        int d = t->discriminant;
        if (x[d] < t->key[d] )
            t->left = insert(t->left,x)
        else
            t->right = insert(t->right,x);
    }
    return t;
}
```

Listing 2: Insert of relaxed kd-trees

```
relax_kdtree delete(relax_kdtree t, type& x) {
    //t cannot be empty and has to contain x
    //actually we have to check all the possible coordinates in
        t->key and x
    if (t->key == x)
        return delete_node(t)

    int d = t->discriminant;
    if (x[d] < t->key[d])
        t->left = delete(t->left,x)
    else
        t->right = delete(t->right,x)
    return t;
}
```

Listing 3: Deletion of relaxed kd-trees

```
//we have some query similar to: (3,*) or (3,2,*)

relax_kdtree partial_match(relax_kdtree t, type& q){
    //t cannot be empty
    //we have to check all the possible coordinates of the key
    if (t->key matches q)
        results.add(t->key)

    int d = t->discriminant;
    if (d == * ){
        results += partial_match(t->left, q)
        results += partial_match(t->right, q)
        return results;
    }
    if (x[d] < q)
        results += partial_match(t->left,q)
    else
        results += partial_match(t->right,q)

```

```
20      return results;
21  }
```

Listing 4: Partial match of relaxed kd-trees

```
1  //we have some query similar to: (3,2)x(radius 1, radius 2), we
       need to return the points inside that range
2
3  relax_kdtree orthogonal_match(relax_kdtree t, type& q){
4      //t cannot be empty
5      //we have to check all the possible coordinates of the key
6      if (t->key matches q)
7          results.add(t->key)
8
9      int d = t->discriminant;
10
11     if (x[d] < q) //Check lower bound and upper bounds
12         results += partial_match(t->left,q)
13     if (x[d] >= q) //Check lower bound and upper bounds
14         results += partial_match(t->right,q)
15
16     return results;
17 }
```

Listing 5: Orthogonal match of relaxed kd-trees

2.3 Propose an experiment to evaluate the average cost of partial match queries in relaxed k-d trees.

In the experiment, first we generate randomly a tree. For that we obtain a set of random points, we insert them in the tree.

```
1  NUM_EXPLORATIONS = x;
2
3  while (num_tests < NUM_EXPLORATIONS)
4      size = n;  // number of points
5      num_pm = m; // number of partial matches we will try to find
           in each randomized tree
6
7  P = generate_random points(n)
8
9  # Create randomized tree
10 for i in lenght(n):
11     T == Null
12     insert (T, P[i])
13
14 #Create random partial match
15
16 //start time accounting
17 for j = 0; j <  num_pm; j++
18 pm = generate_partial_match()
```

```
19 partial_match(T, pm)
20
21 //end time accounting
22
23 Increase total_time by (Time accounted / number of partial
       matches (num_pm))
24 increase num_tests
25
26 end while
27 //Report average total_time of total NUM_EXPLORATIONS
```

Listing 6: Pseudo-code on how evaluate the average cost of partial match queris in relaxed kd-trees

2.4 Propose an implementation of k dimensional quad trees (in c++) together with the operations of insertion of a new element and partial match search. What can you say about deletions?

```
1 struct node{
2     int capacity = 2^k;
3     Type_k[capacity] key;
4     Type_v value;
5     node** children[k]; //each of these will contain a subtree
          for a k dimension
6     type points[capacity];
7 };
```

Listing 7: Class Node of k dimensional quad trees

2.5 Set a recurrence for the average cost of partial match queries in relaxed k-d trees. Can you solve it?

- In each level of the tree ideally we can follow just one branch (or in the worst case, both)
- Each branch we follow it has a cost of log n
  - Ideally, we will have to follow only one branch
  - In the worst case we will have to follow all the branches (the partial match matches all the nodes) that means a cost of n*log n
- Then we have: log n $\leq$ average cost $\leq$ n*log n

2.6 Give an example of a 2-dimensional range tree together with a range query. Show graphically which are the nodes visited by the search algorithm and how it performs the search.

First of all, the algorithm to do a range query in a 2-dimensional range tree is:

```
2DRangeQuery([x0,y0]:[x1,y1]) {
    if node is a leaf:
        if range(node) is in range([x0,y0]:[x1,y1])
            return all points that are in the range
        else return NULL
    else
        if discriminant(node) < range([x0,y0]:[x1,y1]) //it
            depends on which is the discriminant
            return 2DRangeQuery(node->left)
        else if discriminant(node) > range([x0,y0]:[x1,y1])
            return 2DRangeQuey(node->right)
        else
            return 2DRangeQuery(node->left) + 2DRangeQuery(node
                ->right)
}
```

Listing 8: Algorithm to do a range query

For showing demonstration we used the slide we used in the theory class (figure 1):

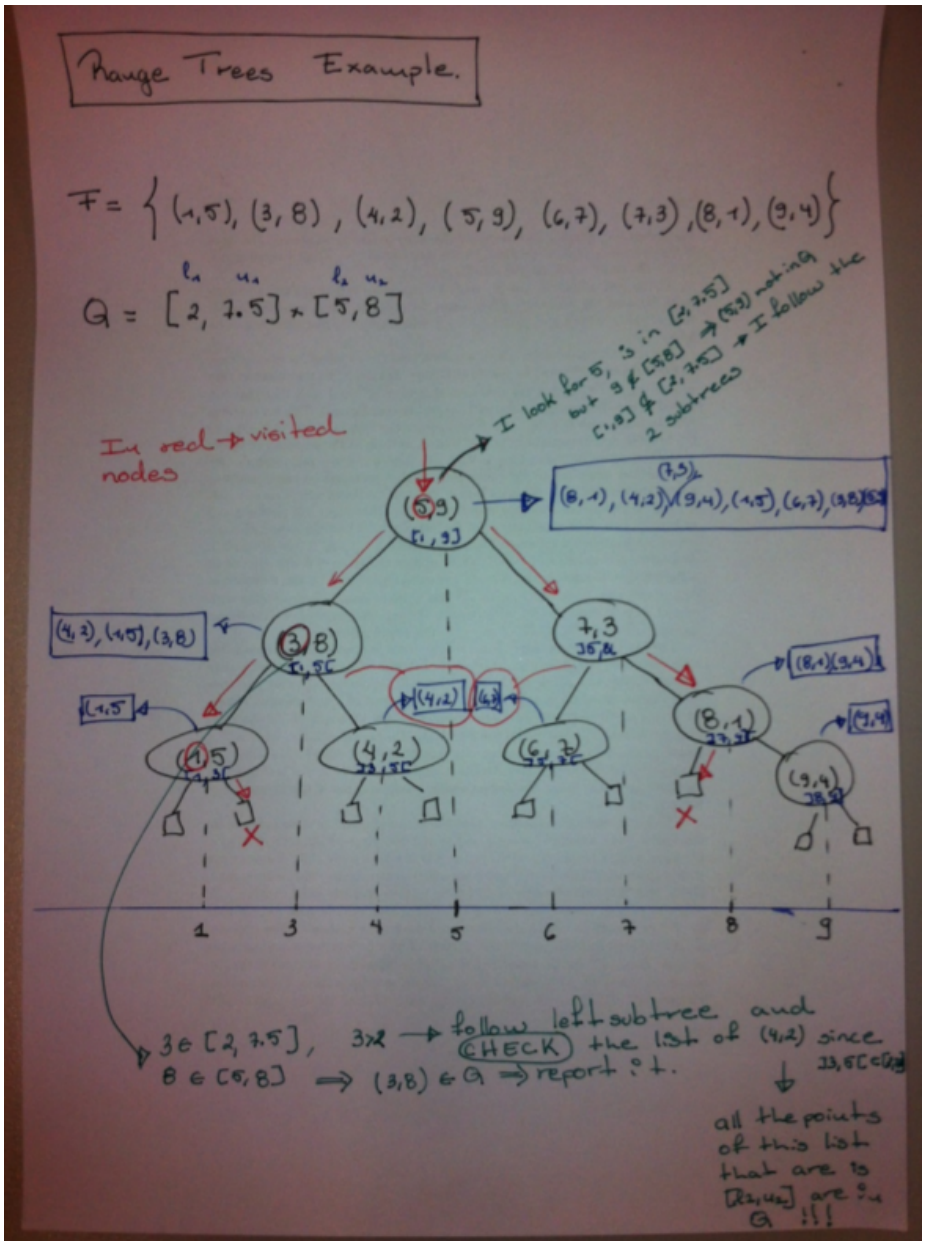

Figure 1: Range query example

Since we need to traverse the tree in 2 dimensions we spent there $\log^2$n (log n if it were just one dimension) and the points we report in the query K.
All sum up is: $O(log^2 n + k)$

2.7 What is fractional cascading? Give an example in a 2-dimensional range tree.

Instead having to traverse all the tree we could have some "sets" that represents each level of our range tree.

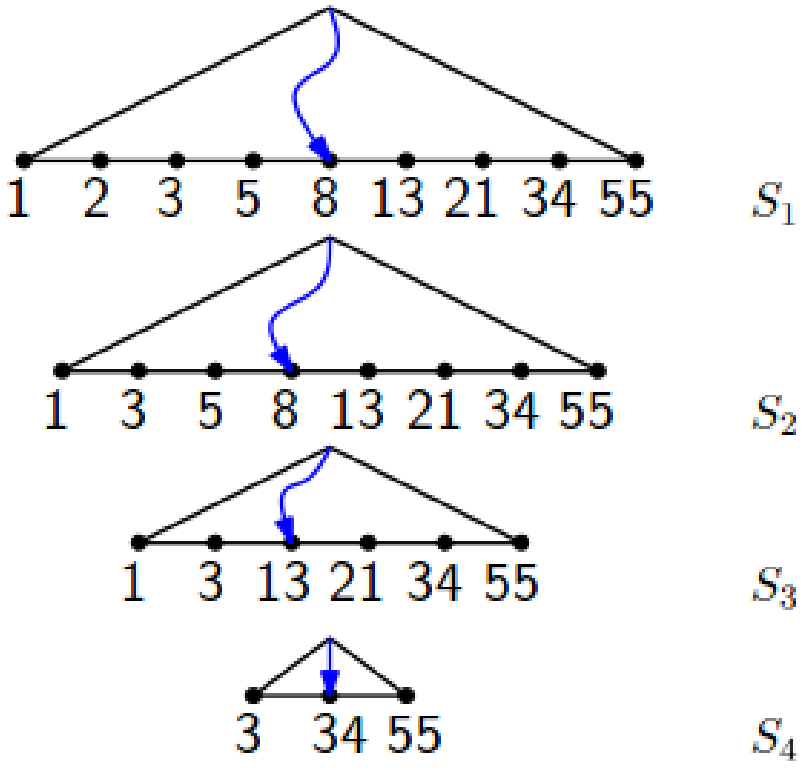

Figure 2: Sets on a tree

Then we just need to find x1 (from [x1,x2] for each Sm and iterate each set until find x2. But even better, it would be store pointers in each element of S_{m-1} to the smallest element in S_m that is $\leq$ element

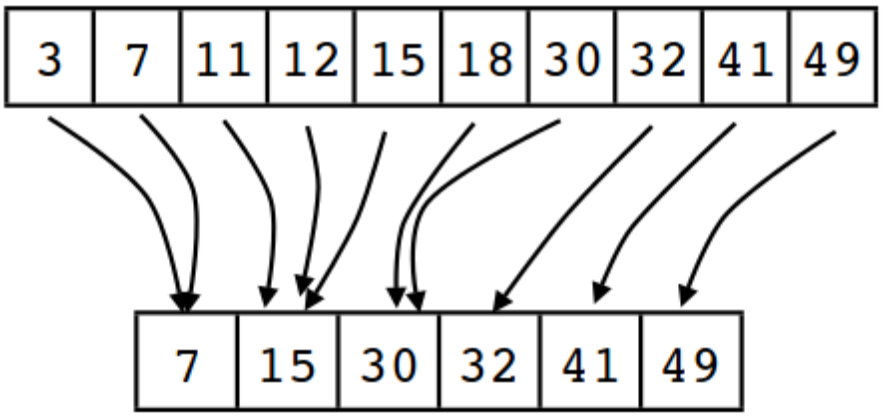

Figure 3: Pointers for fractional cascading

Then, our range search cost will be: *O(log n + k)* [5][4]

2.8 What are PR (Point region) k-d trees and quad trees?

K-d trees and quad trees are not divided by a key of each node, instead, the division is made by a fixed criteria and usually a spacial criteria: each node is determined by the coordinates of one point the the space, therefore instead of accessing the trees for a key, it is done by searching for a point.
And for the quad-tree the criteria instead of a point, usually it is the center of the region we want to divide: if we have a 2D space that we want to represent it as a quadtree, we would get its center, and divide the space in 4, these 4 would be the children. This also happens in octrees but for 3D spaces.

Then, a *Point Region* means the fixed point that we used as a *key* for the node.

2.9 Show how to represent a 3-D object in a delimited region space using octrees. Give an example

If we need to divide a 3D object with an octree, we can get the center of the space containing it (or get the maximum and minimum values of each dimension of the object to use them as container):

- The center point (*PR* will be our *key* for our root
- Each children will be defined as a sub-space of the space divided for our PR: upper right in the back, upper left in the back, upper right in the front, etc.

In the case we want to have a maximum of points per children, we can apply it recursively.

One example is illustrated in figure 4

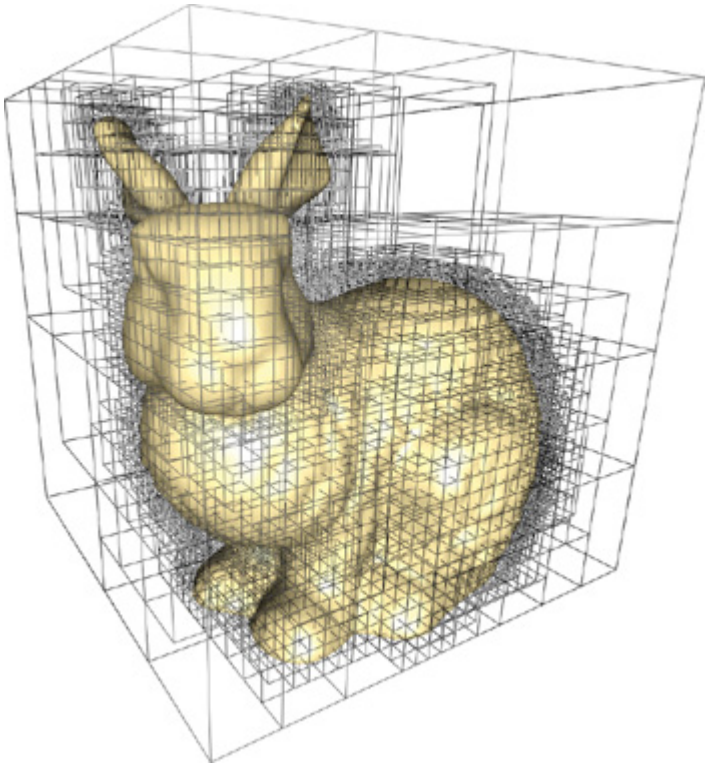

Figure 4: An object contained in octrees

3 Metric Data Structures

3.1 Discuss the dynamism of the data structures presented in this topic.

This kind of data structure are used to do different searches:

- Range queries
- Nearest neighbor queries
- k-Nearest neighbor queries

And in a *general* data structure these queries are usually resolved with brute force (checking each point against all the others); this is too costly. To be able to do such queries efficiently, we need metric data structures.

The problem in these metric data structures comes with insertions and deletions. These data structures are usually built once and *queried* enough times to payback the built cost; while permitting efficient insertions is quite usual, deletions are rarely handled. In several indexes one can delete some elements, but there are selected elements that cannot be delete at all, in that situation, deleting an element could require to scrap and rebuild the data structure. [2]

3.2 What are approximate nearest neighbor queries? Look for a data structure that deals with them, describe it and give its properties.

Nearest neighbor $queryNN(q)$: retrieve from U the closest element to q:
$NN(q) = u \in U | d(u,q) \leq d(v,q) \forall v \in U$

An example would be: Geometric Near-neighbor Access Tree (GNAT) It is based on the philosophy that the data structure should act as a hierarchical geometrical model of the data as opposed to a simple decomposition of the data which does not use its intrinsic geometry.

The offered performance can be seen in table 2, where n is the number of nodes (non empty) and k is the average degree (equal to N/n). As the tables specifies, the author of the data structure have been not able to determine a domain or boundaries for the query time. [1]

Characteristic	Cost
Space	O(n_m + N)
Build	O($N_m \times log(m) \times N$
Query time	Not analyzed

Table 2: GNAT characteristics

References

[1] Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574–584, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[2] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, September 2001.

[3] Amalia Duch. *Slides from Advanced Data Structures course.* FIB, 2016.

[4] Carl Kingsford. *Slides from CMSC 420: Data Structures course.* University of Maryland, 2016.

[5] Marc van Kreveld. *Slides from Geometric Algorithms course.* Utrecht University, 2016.