TRIBHUVAN UNIVERSITY
INSTITUTE OF SCIENCE AND TECHNOLOGY

Central Department of Computer Science and Information Technology
Kirtipur, Kathmandu
2022

Report on:

“Range Search ”

Submitted By:
Sudhan Kandel
Semester: 2nd
Roll no: 2

Submitted To:
Asst.Prof. Jagdish Bhatta

Table of Contents

Figures

1. Range Search Problem

Range search is one of the most important fields in Computational Geometry and have range applications in database searching and geographical databases. A central problem in computational geometry, range searching arises in many applications, and a variety of geometric problems can be formulated as range-searching problem. A typical range-searching problem has the following forms. Let S be a set of n points in R^d, and let R be a family of subsets of R^d; elements of R are called ranges. Typical example of range search includes rectangles, half spaces, simplices, and balls. Processes S into a data structure so that for a query range $\gamma \in R$, the points in $S \cap \gamma$ can be reported or counted efficiently. A single query can be answered in linear time using linear space, by simple checking for each point of S whether it lies in the query range. Most of the applications, however, call for querying the same set S numerous times, in which case it is desirable to answer a query faster by preprocessing S into a data structure [1].

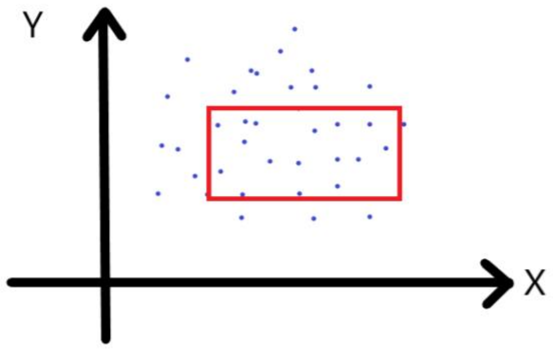


Figure 1: Range Search Problem

2. One-Dimensional Range Search

It is one of the easiest search problems, here we are given some set of points S of size n and a closed interval describing the range. Then we must return all the points that belong to the range [2].

2.1 One-Dimensional Range Searching using array

Additionally, this process is started by sorting the given value. We need to locate the starting point of the range, or the left boundary using binary search and start traversing and return the value between the intervals. All in all, this process is illustrated in the given figure below.

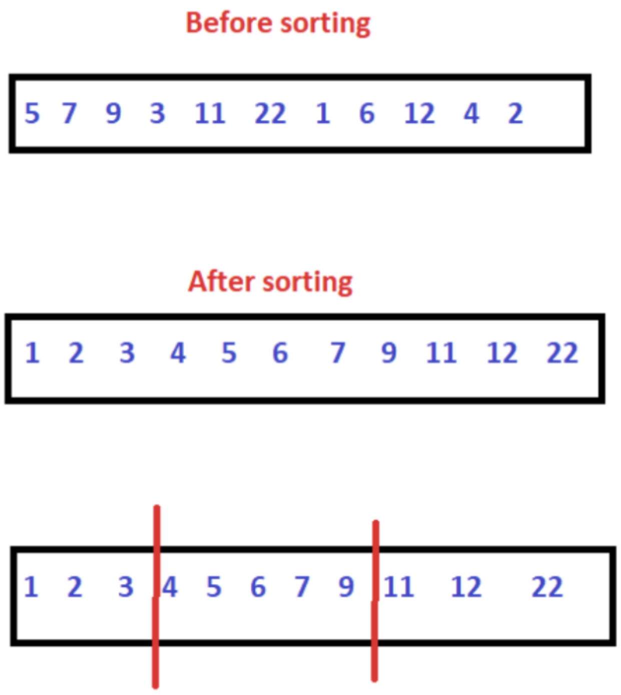


Figure 2: One-Dimensional Range Searching using array

In the above example the range is [4,10]

Time Complexity

- O(longn) for binary search and O (n longn) for sorting

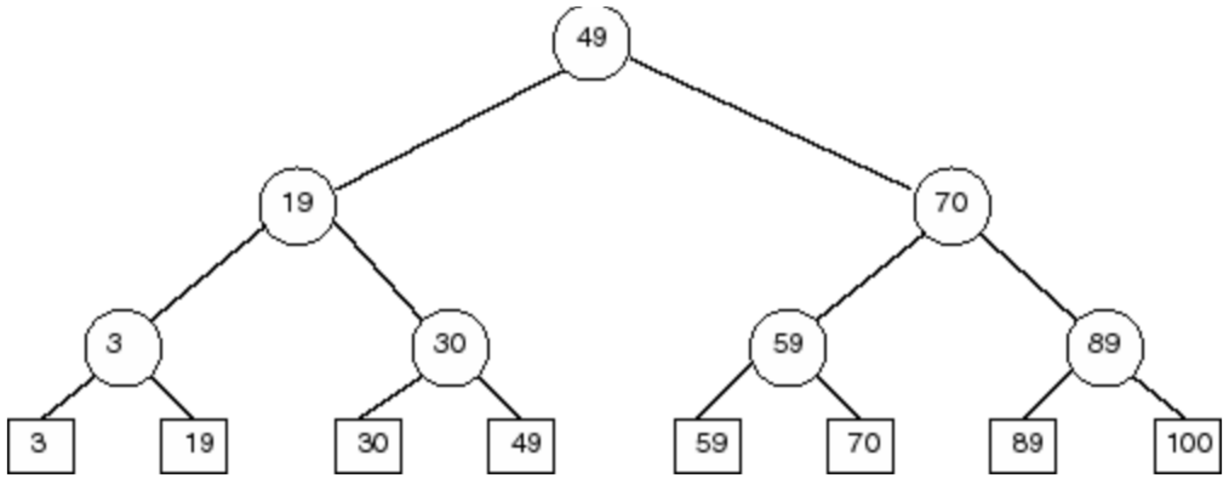
2.2 One-Dimensional Range Searching using binary search tree

Binary tree is a tree, which in turn is a hierarchical data structure where each node can be defined with a value of any data type and two pointers that point or store address of the next elements in the hierarchy. The pointers are also called as children, with them being left and right child; the root node is referred to as parent. And a binary tree has a maximum of two children [2].

Binary search tree is a binary tree which has the following properties:

1. The left subtree's data must be less than that of the root.
2. The right subtree's data must be greater than that of the root.

Range search using binary search tree is worked as follow. Initially we need to define the range [x, y] which is compared to the root node of the tree. If root value is greater than x and y, we must search in the left subtree and return values. Otherwise, we need to search in the right subtree and return values. These two cases are called total overlap. Additionally, there is something called partial overlap where x belongs to left and y belong to right. In this case we must search both the subtrees [2].

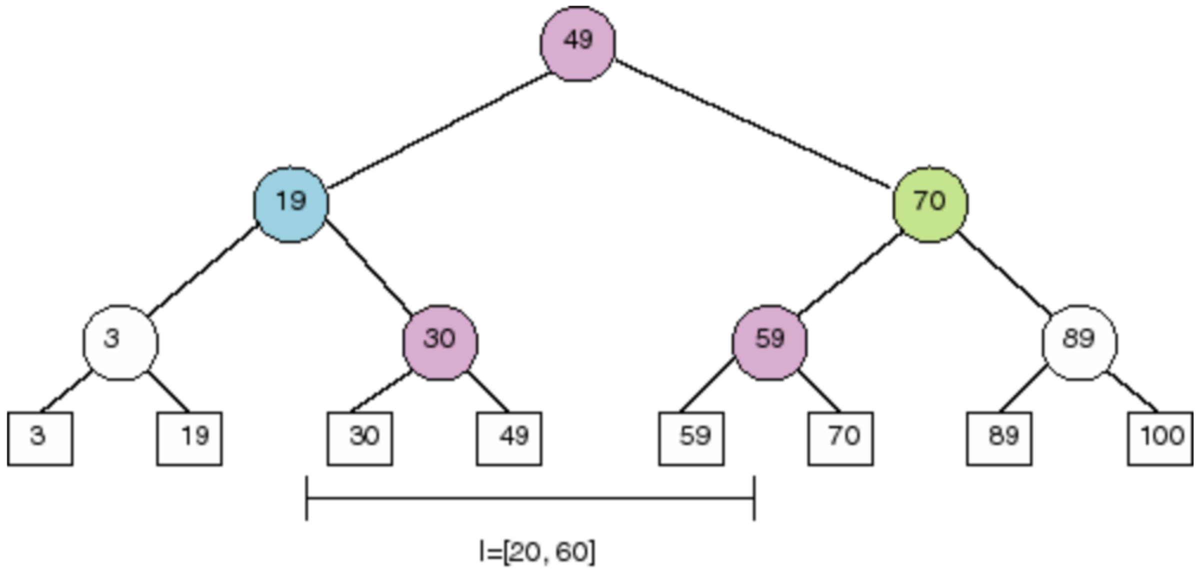


Let's us consider range is [20,60]

20<49 and 60>49

Here the condition is partial overlapping because x belongs to left sub tree and y belong to right.

Again 20>19 so search at right subtree and 60<70 search at left subtree.



Pseudo code:

```
FindPoints([x, x'], T)
   if T is a leaf node, then
      if x <= val(T) <= x' then
         return { val(T) }
      else
         return {}
      end if
   end if
   <else T is an interior node of tree>
   if x' <= val(T) then
      return FindPoints([x, x'], left(T))
   else if x > val(T) then
      return FindPoints([x, x'], right(T))
   else <interval spans splitting value>
      return FindPoints([x, x'], left(T)) union FindPoints([x, x'], right(T))
   end if
```

Time Complexity

Path lengths O (log n)

Range counting queries O (log n)

Range reporting queries O (log n +k)

O(n) storage and O(n log n) preprocessing → for sorting

3. Two-Dimensional Range Search

The variations in single dimensions with respect to range was basically open or closed intervals. But, when we go to higher dimensions, two or more, the range intervals could take up any shape or form. Some examples are listed below.

1. Rectangular range query, basically it's orthogonal. This was the cartesian product of the range. e.g., [x1, x2] *[y1, y2]. Rectangular range query need not to be orthogonal, it can be aligned as well.

2. Circular range queries, basically a circle with radius ‘r’, centered as ‘c’. Now what are the point inside this circle? That’s basically circular range query.

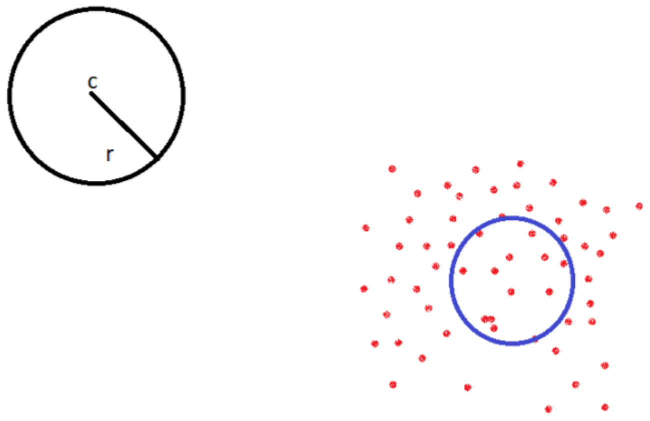


Figure 3: Circular range queries

3.1 Using KD-trees for two-Dimensional Range Search

The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, know as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension’s axis.

In higher dimensions we prefer the range tree s. it is an ordered tree data structure to hold a list of points. It allows all points within a given range to be reported efficiently and is typically used in two or higher dimensions. We will be looking a special type of the range tree called k-d tree where the d represents the dimension [2].

Let’s understand the working mechanism of KD tree this example. Let p be a set of points in a two dimension.

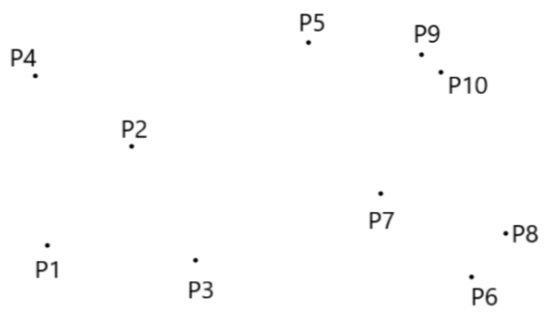


Figure 4: Set of points in a two dimension

Here, the basic assumption is no two points have same x-coordinate, and no two points have same y-coordinate. Each point has its x-coordinate and y-coordinate. Therefore, we first split on x-coordinate and then on y-coordinate, then again on x-coordinate, and so on. At the root we split the set P with vertical line l into two subsets of roughly equal size. This is done by finding the median x- coordinate of the points and drawing the vertical line through it. The splitting line is stored at the root. Pleft , the subset of points to left is stored in the left subtree, and Pright, the subset of points to right is stored in the right subtree. At the left child of the root we split the Pleft into two subsets with a horizontal line. This is done by finding the median y-coordinate if the points in P left . The points below or on it are stored in the left subtree, and the points above are stored in right subtree. The left child itself store the splitting line. Similarly Pright is split with a horizontal line, which are stored in the left and right subtree of the right child. At the grandchildren of the root, we split again with a vertical line. In general, we split with a vertical line at nodes whose depth is even, and we split with horizontal line whose depth is odd. The below figures illustrates this clearly [1].

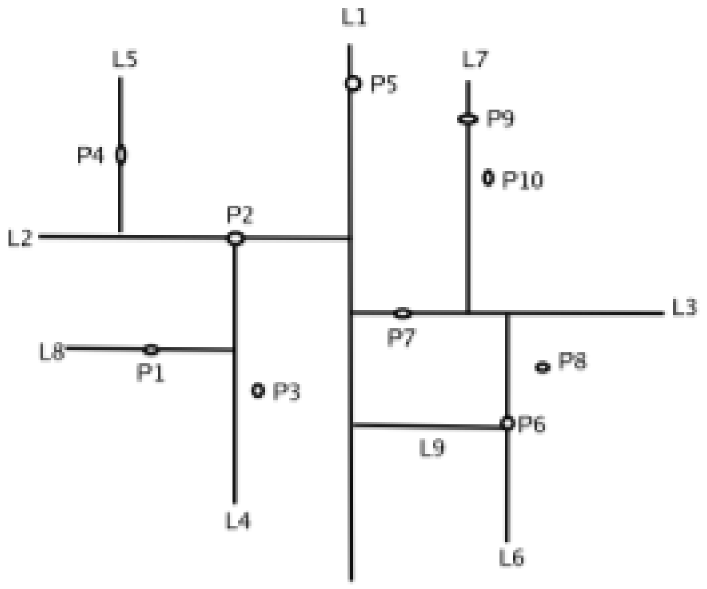


Figure 5: Subdivision

The k-d tree for the above figure will be

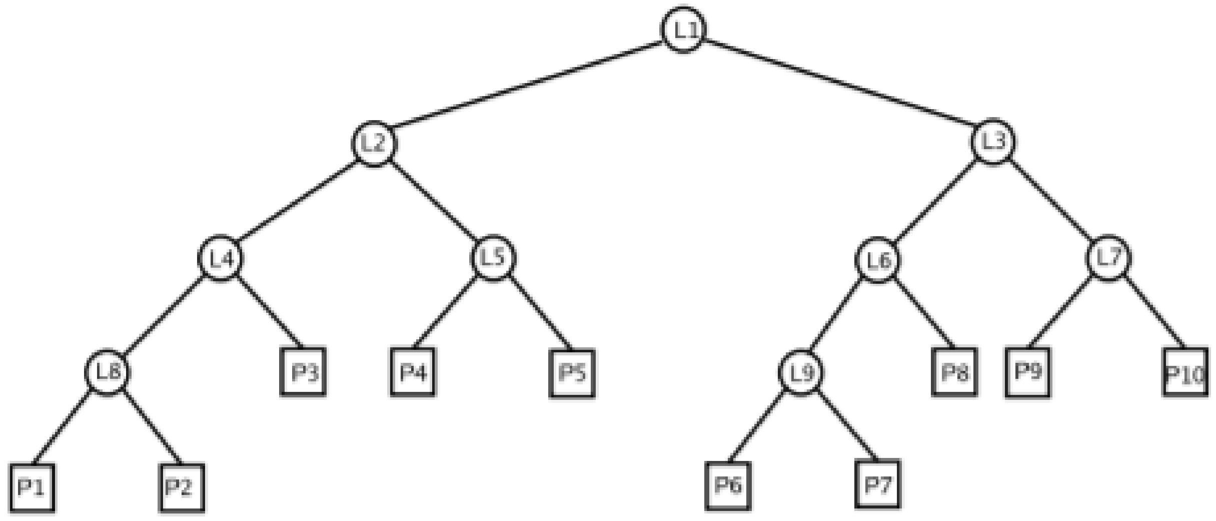


Figure 6: Tree Structure

Algorithm

Let us consider the procedure for constructing the kd-tree. It has two parameters, a set if points and an integer. The first parameter is set for which we want to build kd-tree, initially this the set

P. The second parameter is the depth of the root of the subtree that the recursive call constructs. Initially the depth parameter is zero. The procedure returns the root of the kd-tree.

1. If p contains only one point.
2. Then return a leaf storing this point
3. Else if depth is even
4. The split p into two subsets with a vertical line l through the median x-coordinate of the points in p. let p1 be the set of points to the left of l or on l, and let p2 be the set of points to the right of l.
5. else split P into two subsets with a horizontal line l through the median y-coordinate of the points in P.Let P1 be the set of points to the below of l or on l, and let P2 be the set of points above l.
6. vleft←BUILDKDTREE(P1, depth +1).
7. vright←BUILDKDTREE(P2, depth +1).
8. Create a node v storing l, make vleft the left child of v, and make vright the right child of v.
9. return v.

Time Complexity:

The most expensive step is to find the median. The median can be find in linear time, but linear time median finding algorithms are rather complicated. So first presort the set of points on x-coordinate and then on y-coordinate. The parameter set P is now passed to the procedure in the form of two sorted list. Given the two sorted list it is easy to find the median in linear time. Hence the building time satisfies the recurrence,

T(n)=O(1), if n=1,

O(n)+2T(ceil(n/2)),if n>1

which solves to O(q*n log n), where q denotes the number of queries. Because the kd-tree is the binary tree, and every leaf and internal node uses O(1) storage, therefore the total storage is O(n).

References

[1] Agarwal, Pankaj K and Erickson, Jeff and others, "Geometric range searching and its relatives," *Contemporary Mathematics,* vol. 223, 1999.

[2] Bentley, Jon Louis and Friedman, Jerome H, "Data structures for range searching," *ACM Computing Surveys (CSUR)*, 1979.