C.S. 252 Prof. Roberto Tamassia
Computational Geometry Sem. II, 1992–1993

One-dimensional range searching. Two-dimensional range-searching.

Lecture 01
Date: Feb 3, 1993
Scribe: Mark Marcus

1 Introduction

This class cover three areas. The first area was an introduction to dynamic algorithms. The second area was a review of data structures used frequently in Computational Geometry. The third area was “one-dimensional range searching” and “two-dimensional range-searching”.

2 Introduction to Dynamic Algorithms

A dynamic algorithm was characterized by the concept of an “on-line” sequence of operations on a data structure. These operations being query and update. Each operation may be performed before the next operation is known.

For example, if you had a directed graph, an intermixing of updates and queries to this graph can be made. A query might be: is there a directed path between two nodes of the graph? An update may be: add an edge to the graph.

The performance of dynamic alogrithms is measured by the asymptotic space needed, the aymptotic time to perform a query and the asymptotic time to perform an update. Typically these performance characteristics indicate worst-case analysis. Average-case analysis being too difficult – one of the reasons for this is that it is hard to define what an average-case is.

If a sequence of N operations takes time T, then the amortized time complexitiy of an operation in the sequence takes time T/N. A worst-case amortized time complexity guarantees what the average performance will be for an asymptotically large n.

3 Review of Frequently Used Data Structures

This section mainly covers how different kinds of binary search trees can be used to implement the dynamic dictionary. The section on “Dynamic Dictionaries” covers the basic definition of a dynamic dictionary. The “Balanced Binary Search Trees” section covers basic properties of all balanced binary search trees. The “Properties of Red-Black Trees” and the “Properties of BB[α] Trees” sections respectively cover the properties of these special cases of the binary search tree.

3.1 Dynamic Dictionaries

A dynamic dictionary, S, can be modeled as a dynamically evolving subset of a totally ordered universe. An example of a totally ordered universe is a set of integers.

Two query operations that must be provided are *search* and *locate.*

Examples:

Search(q):	Is q in S?
Locate(q):	find largest $x \in S$ such that $x \leq q$

Four update operations that must be provided are *insert, delete, split,* and *splice.*

- Insert(x, S).

  This operation inserts element x into dictionary S.

- Delete(x, S).

  This operation deletes element x, if it exists, from dictionary S.

- Split(S, x; $S^{'}$, $S^{''}$), where $y \in S^{'} \Leftrightarrow y \leq x$, and $y \in S^{''} \Leftrightarrow y > x$.

  This operation creates two dictionaries: $S^{'}$ and $S^{''}$. Each dictionary is a subset of dictionary S. $S^{'} \bigcup S^{''} \equiv S$. $S^{'}$ contains all the elements of S that have a value less than or equal to x and dictionary $S^{''}$ contains all the elements of S that are more than x.

- Splice($S^{'}$,$S^{''}$; S), where $x^{'} \in S^{'}$ and $x^{''} \in S^{''} \Leftrightarrow x^{'} \leq x''$.

  This operation produces a new dictionary S. Dictionary S is a union of the elements of dictionaries $S^{'}$ and $S^{''}$.

In the following sections, specializations of the generic Binary Search Tree are used to implemement the Dynamic Dictionary. All variations discussed have the following two properties: (1) nodes of the the Binary Search Tree represent elements of the Dynamic Dictionary and (2) an inorder visit of the Binary Search Tree visits the nodes of the elements of Dynamic Dictionary in sorted order.

3.2 Properties of a Binary Search Trees

A "Binary Search Tree" is a binary tree where all the values in a right-subtree are greater than all the values in the left-subtree. Given that h is the height of a binary search tree, T, then the following Complexity of Operations hold.

- Query: $O(h)$
- Insert or Delete before search: $O(h)$
- Insert after search: $O(1)$

  Delete after search: $O(h)$
- Split: O(h); See Figure 1.
- Splice: O(1)

figure=split.ps,height=3in

Figure 1: Example of Split

figure=rotate.ps

Figure 2: Example of Rotation

A “rotation” is an elementary restructuring of a subtree that persevers the inorder sequence. Figure 2 illustrates a rotation. Rotations are used to keep a balanced search tree balanced after an insertion or deletion.

General information about balanced search trees:

- A balanced search tree for n elements uses space $O(n)$ and height $O(\log n)$
- Queries and updates take time $O(\log n)$
- Updates are performed in two phases:
  - search for the element to be inserted/deleted with a query operation
  - rebalance the tree by means of roatations.
- Classes of balanced search trees
  - height-balanced; e.g., red-black
  - weight-balanced: e.g., BB[α]
- Balanced search trees are often augmented with secondary structures stored at the internal nodes.
- The secondary structures need to be updated after a rotation.

3.3 Properties of Red-Black Trees

- Rebalancing after an insertion or deletion can be done with $O(1)$ rotations (useful if the secondary data structures can be updated in polylog time)
- Red-black trees are equivalent to 2-3-4 trees
- A variation of red-black trees (equivalent to level linked 2-3-4 trees) has the following additional properties:
  - The rebalancing time in a sequence of insertions and deletions is $O(1)$ amoritized
  - If the nodes representing the elements to be inserted/deleted are knwon, updates take time $O(1)$ amortized.

3.4 Properties of BB[α] Trees

Consider a BB[α] tree augmented with secondary structures. Let $f(l)$ be the time to update teh secondary data structures after a rotation, where l is the number of leaves in the subtree involved in the rotation. The amortized rebalancing time of an update operation in a sequence of insertions and deletions is

$$\begin{array}{ll} O(1) & \text{if } f(l) = O(l^a), a < 1 \\ O(\log^{c+1} n) & \text{if } f(l) = O(l \log^c l), c \geq 0 \end{array}$$

4 Range Searching

Range Searching is given a set P points in a d-dimensional space, $Euclidean^d$, answer the "Range Query": Report the points of P contained in an *orthogonal query* r. "The choice of search domain is frequently referred to as *orthogonal query*, possibly because the hyperplanes bounding the hyperrectangle are each orthogonal to a coordinate axis."[6] An example of a two-dimensional range query is shown in Figure 4.

In general:

$$r = (a_1, b_1) \times (a_2, b_2) \times \ldots \times (a_d, b_d)$$

If $d = 1$ then the query range r is an interval on the line. If $d = 2$ the the query range r is a rectangle with sides parallel to the axes.

Variations to the "range query" are (1) count the points in query range r and (2) if points have associated weights, compute the total weight of points in r.

One-Dimensional Range Searching
See Figure 4 for example.

- use a balanced search tree T with internal nodes associated with the points of P
- thread nodes in in-order
- Query for range $r = (x', x'')$
  - serach for x' and x'' in T, this gives nodes μ' and μ''
  - follow threads from μ' to μ'' and report points at internal nodes encountered

Complexity of One-Dimensional Range Searching

- Space requirement for n points: $O(n)$
- Query time: $O(\log n + k)$, where k is the number of points reported
- Time for insertion or deletion of a point: $O(\log n)$.
- Note that thread pointers are not affected by rotations.

Exercises from Class

Queries can be performed in $O(\log n + k)$ without using threads. One could walk the path from the root to μ' in $O(\log n)$ time. As the walk was performed all nodes less then μ' could be spliced out of the tree. A similar treatement for μ'', but this time you splice out nodes greater than μ''. Then the answer can be obtained by a pre-order walk of the tree, enumerating the nodes as you go, will cost in time complexity $O(k)$. The total of all three operations being $O(2 \log n + k)$ which is equivalent to $O(\log n + k)$.

1-D range counting queries can be performed in $O(\log n)$ time. This can be accomplished by keeping at each node the number of children it has. As you perform the two $O(\log n)$ walks to find μ' and μ'' you can subtract off the number of nodes not in the answer.

Assuming that the points have weights, you can find in time $O(\log n)$ the heaviest point in the query range. The two paths from the root to μ' and μ'' can be found in $O(\log n)$ time. Each node can store the heaviest node in its sub-tree. A traversal from the path from μ' to the root can calculate new heaviest nodes eliminating the nodes not found in the answer to the query. A similar operation can be performed for μ'', the root node will then contain the heaviest node for the answer to the query.

4.1 One-dimensional range searching

In order to extend the technique mentioned above to higher dimensions, an alternate structure for 1-D range searching is needed. This technique is more complex than a simple balanced search tree.

4.1.1 One-dimensional range searching: data structure

See Figure 4.1. A "Range Tree" is a balanced search tree T, with the following characteristics:

- leaves $\leftrightarrow$ points, sorted by x-coordinate.
- node $\mu \leftrightarrow P(\mu)$ is the subset of points at the leaves in the subtree μ.

The space needed to n points in the one-dimensional ranage tree is $O(n \log n)$

4.1.2 One-Dimensional Range Queries

In Figure 4.1.2 the bottom bold line segment represents the query range (x', x''). A query range corresponds to a set of the so-called "allocation nodes". An allocation node of μ of T for the query range (x', x'') is such that (x', x'') contains $P(\mu)$ but not $P(\text{parent}(\mu))$.

- the allocation nodes are $O(\log n)$
- they have disjoint point-sets
- the union of their point-sets is the set of oints in the range (x', x'')

The Query Algorithm is:

1. find the alocation nodes of (x', x'')

2. *for each* allocation node μ report the points in $P(\mu)$

4.1.3 How to Find the Allocation Nodes

Each node μ of T stores both $\min(\mu)$ and $\max(\mu)$. $\min(\mu)$ is the smallest x-coordinate in $P(\mu)$ and $\max(\mu)$ is the largest x-coordinate in $P(\mu)$. Each node also stores the so-called "point-set" at each node. A point-set is a set of all the leaves represented by $P(\mu)$.

The following recursive porcedure marks all the allocation nodes of (x', x'') in the subtree μ.

```
Procedure Find($\mu$)
if $x' \leq \min(\mu)$ and $x'' \geq \max(\mu)$
    then mark $\mu$ as an allocation node
    else if $\mu$ is not a leaf then
        if $x' \leq \max(\text{left}(\mu))$
            then Find(left($\mu$))
        if $x'' \geq \min(\text{right}(\mu))$
            then Find(right($\mu$))
```

4.1.4 Dynamic Maintenance of the Range Tree

See Figure 4.1.4. An algorithm for the insertion of a point p onto the line is as follows:

- create a new leaf λ for p in T. We use a red-black tree for T.

- rebalance T by means of rotations In a rotation, we need to perform split/splice operations on the point-sets stored at the nodoes involved in the rotation. We use any type of balanced trees for the point-sets.

Insertion time is $O(log^2 n)$. There may be $O(\log n)$ rotations, each rotation may take $O(\log n)$ to insert or delete an element form the point-set data structure associated with each node. Similarly for deletions.

4.2 Two-dimensional range-searching.

See Figure 4.2. For two-dimensional range searching a two level data structure is used, namely, the "2-D Range Tree".

The primary data structure is a 1-D range tree T based on the x-cooridnates of the points

- leaves $\leftrightarrow$ points, sorted by x-coordinate.

- node $\mu \leftrightarrow P(\mu)$ is the subset of points at the leaves in the subtree μ.

The secondary data structure associated with each node μ is a data structure for 1-D range searching by y-coordinate in the set $p(\mu)$. This data structure can be either a 1-D range tree or a balanced tree.

4.2.1 Two-dimensional Range Queries with the 2-D Range-Tree

See Figure 4.2.1. Query algorithm for range $r = (x', x'') \times (y', y'')$

- find the allocation nodes of (x', x'')
- *for each* allocation node μ perform a 1-D range query for range (y', y'') in the secondary structure of μ

4.2.2 Space and Query Time

The space used for n points depends on the secondary data structures: $O(n \log^2 n)$ space with 1-D range trees and $O(n \log n)$ space with balanced trees.

Query time for a 2-D range query:

- $O(\log n)$ time to find the allocation nodes.
- Time to perfrom a 1-D range query at allocation node μ: $O(\log n + k_\mu)$, where k_μ points are reported

Total time: $\sum_\mu (\log n + k_\mu) = O(\log^2 n + k)$

4.2.3 Dynamic Maintenance of the Range Tree

Algorithm for the insertion of a point p.

- create a new leaf λ for p in T
- rebalance T by means of rotations.
- *for* each ancestor μ of λ *do* insert p in the secondary data structure of μ

When performing a rotation, we rebuild from scratch the secondary data structure of the node that becomes the child (ther seems to be nothing better to do).

The cost of a rotation at a node μ is $O(|P(\mu|) = O(\#$ leaves in subtree of$\mu)$. By realizing T as a $BB[\alpha]$tree, the amortized rebalancing time is $O(\log n)$.

The total insertion time is dominated by the for-loop and is $O(\log^2 n)$ amortized.

Similar considerations hold for deletion.

4.2.4 Rotation in a 2-D Range Tree

See Figure 4.2.4.

- The secondary data structure of μ'' is the same as the one of v'.
- The secondary data structure of v'' needs to be constructed.
- The secondary data structure of μ' needs to be discarded.

4.2.5 Summary of Two-Dimensioanl Range Tree

- Two-level tree structure (RR-tree)
- Reduces 2-D range queries to a collection of $O(\log n)$ 1-D range queries
- $O(n \log n)$ space
- $O(\log^2 n + K)$ query time
- $O(\log^2 n)$ amortized update time

References

[1] Y. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," *Proceedings of the IEEE, Vol 80, No. 9, Sept. 92* , 1412 – 1417, 1992.

[2] T. Cormen, C. Leiserson, and R. Rivest, "Introduction of Algorithms," *MIT Press and McGraw-Hill*, 1990.

[3] L.R. Guidbas and R. Sedgewick, "A dichromatic framework for balanced trees," *FOCS* 1978.

[4] S. Huddleston and K. Mehlhorm, "A New Data Structure for Representing Sorted Lists," *Acta Informatica 17*, 157-184, 1982.

[5] N. Blum and K. Mehlhorn, "On the Average Number of Rebalancing Operations in Weight-Balanced Trees," *Theoretical Computer Science 11*, 303-320, 1980.

[6] F. P. Perperata and I. Shamos, "COMPUTATIONAL GEOMETERY An Introduction", *Springer-Verlag*, 1985.