1998 International Computer Symposium
Workshop on Algorithms
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

SEGMENT B-TREE: A DYNAMIC AND EFFICIENT INTERVAL INDEXING DATA STRUCTURE

Wenbing Zhang and Jiang-Hsing Chu

Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA
jchu@cs.siu.edu

ABSTRACT

Spatial data are widely used in Computer Graphics, GIS, CAD, DBMS, etc. To find an efficient data structure is very important to the system performance. In this paper, we introduce a new data structure, which we call Segment B-tree, for dynamic and efficient interval indexing. Segment B-tree is an improvement to the segment tree by supporting dynamic updating and using B-tree as the frame structure. Algorithms are developed using this new data structure. We give a theoretical analysis of the complexity of those algorithms, which shows the data structure is very good for point query and insertion. We also give the empirical results, which are compared with the results of using other data structures. The empirical results verify the theoretical analysis.

1. INTRODUCTION

Spatial data are widely used in Computer Graphics, GIS, CAD, DBMS, etc. Those data include segments, rectangles, multi-dimensional points, etc. The representation of those data is very important to the system performance. In this paper, we will focus on data structures for segments.

Most database systems available do not support efficient indexing on segment data or other multi-dimensional data. The problem we want to solve is, given a collection of line segments or intervals, how to organize them to achieve a small space requirement and to find an efficient interval indexing algorithm for insertion, deletion and searching.

Several data structures and algorithms have been developed for solving this problem. While those data structures and algorithms are good in some cases, they have their drawbacks. In this paper, we will introduce a new data structure, which we call Segment B-tree, for efficient interval indexing. Before we proceed, we first give a brief survey of several data structures in this field.

1.1 Segment Tree

Segment tree [1] is a simple yet efficient data structure for interval indexing. Fig 1.1 shows an example of a segment tree. The end points of the segments are sorted in an array. The index of the array is stored at the leaf level of a complete binary tree. Each leaf node corresponds to an interval of [$K[i]$, $K[i+1]$). Each nonleaf node corresponds to an interval that is the union of the intervals represented by its children. For example, in Fig 1.1, the root corresponds to the interval of [10, ∞) and the node marked with * corresponds to the interval of [20, 35). Finally, each node is associated with a list of IDs of segments that cover the interval represented by the node.

In a segment tree containing N segments, a node ID can appear at up to $O(\log N)$ nodes. Thus the space complexity is $O(N \log N)$. According to [1], The insertion complexity is $O(\log N)$. The complexity of point query is $O(\log N + n)$, where n is the number of segments found. Deletion is a bit more complicated. On average, its complexity is $O(\log^2 N)$. However, by adding a linked list for each segment that links all the IDs of the segments in the tree, the deletion complexity can be reduced to $O(\log N)$.

Since all the possible end points must be known before we can build a segment tree, it does not support dynamic insertion and deletion. This limits the application of segment tree greatly. Another problem common to all binary tree based data structures is the performance drops significantly when the whole tree is too large to be brought into the main memory.

1.2 R-Tree

R-tree [2] is a very popular data structure for collection of multi-dimensional objects, esp. rectangles. Since segments can be regarded as 1-D rectangles, they can be represented by R-tree. The approach in R-tree is: Data segments are stored in leaf nodes. Each nonleaf node stores the minimum segments that cover the segments stored in its children. It is easy to see that the space complexity of R-tree is $O(N)$. Hence it is space efficient.

The problem with R-tree is the segments stored in one node can overlap and have no order essentially. Overlapping means we might have to search all nodes in a search. Because there is no order, it is necessary to compare all segments in a node when search the node. Therefore, the worst case complexity for search is linear. Another problem with R-tree is when a node is overflow or underflow, it is very inefficient to find an optimal way of rearranging the segments into different groups. This makes it inefficient in dynamic insertion and deletion.

1.3 Interval Tree

Interval tree [3] is a data structure designed specifically for solving the intersection query problem. However it can also be used for point query. Fig 1.3 shows an example of interval tree. Similar to the segment tree, all the possible end points are first sorted and stored in the leaf nodes of a complete binary tree. A nonleaf node is assigned a value between the maximum value stored in its left subtree and the minimum value stored in its right subtree. A segment is stored in the first node from the root whose value is overlapped by the segment. Segments stored at the same node are organized as a linked list or as a binary search tree, which is called the secondary structure. The tertiary structure is used for indexing the active nodes, which are either nodes with second structures or have active nodes in both of their children.

The interval tree is also space efficient. Its space complexity is $O(N)$. If the secondary structure is implemented as a binary search tree, then the complexities of insertion and deletion are both $O(\log N)$. The search complexity is $O(\log N + n)$. This data structure is very nice. The main problem is, like the segment tree, it does not support dynamic updating. Moreover, the data structure is quite complicated. Each node has 8 pointers. If the secondary structure is simply implemented as a linked list, the deletion and search performance could drop significantly.

Most other data structures, while having some advantages, also have their drawbacks. In a word, they are not both efficient and dynamic. Some data structures may also have some special restrictions. For example, priority search trees [4] require the end points of the segments to be all different, which is not practical. One further problem with almost all available algorithms is they do not distinguish whether a segment is open or closed, or they simply assume every segment is left closed and right open.

In the following sections, we will discuss a new data structure, which we call Segment B-tree, for interval indexing. Segment B-tree is an improvement to segment tree. It is B-tree based and is capable of dynamic updating. There are two reasons for choosing B-tree as the frame structure. First, dynamic updating a binary tree often makes it unbalanced, thus affects the performance; while a B-tree is always balanced regardless of dynamic updating. Second, B-tree is an external search tree, so the performance is much better than that of a binary tree when the data set is too large to be accommodated in the main memory.

With this new data structure, we develop efficient algorithms for insertion, deletion and query. The algorithms also handle the end point problem. We will give an analysis on the complexity of the algorithms and the storage requirement.

2. SEGMENT B-TREE

A Segment B-Tree is an augmented B-Tree. As mentioned before, B-tree has some nice properties. So we first very briefly recall some properties of B-tree. A B-tree of order M is an M-way search tree in which

1. All leaves are at the same level.
2. All internal nodes have at least $\lceil M/2 \rceil$ non-empty children. However the root may have only 2 children, but not less, unless it is also a leaf node.

Before we proceeds, we give a few conventions used in this paper:

1. "Segment" and "interval" are synonyms. However, in this paper, "segment" is used to refer the given collection of segments/intervals, while interval is used to refer a range.
2. A Key value is often referred as a point.

A Segment B-tree is a B-tree with additional structures used for interval indexing. In a Segment B-tree of order M, each node is defined as a structure which has the following fields ($MAX = M - 1$, $MIN = \lceil MAX/2 \rceil$):

1. $B[0..MAX]$: an array of pointers to the children/branches.
2. $K[1..MAX]$: an array of Key values stored in this node.
3. *count*: an integer which is the number of key values currently stored.

The above fields are the same as in a B-tree. In order to store segments, we let each node represent an interval. The root represents $(-\infty, \infty)$. Each leaf node represents an interval of ($pre(K[1])$, $succ(K[count])$). Each nonleaf node represents the union of the interval represented by all of its children.

We also add the following fields:

4. $iList[1..MAX]$: an array of linked lists where $iList[i]$ is the list of the IDs of the segments that cover the Key

point $K[i]$ but do not cover the interval represented by this node, except if this node is the root.

5. *eList*[1..*MAX*]: an array of linked list where *eList*[*i*] is the list of the IDs of the segments that have $K[i]$ as an end point.
6. *cList*[0..*MAX*]: an array of linked lists where *cList*[*i*] is the list of the IDs of the segments that cover the interval ($K[i]$, $K[i+1]$) but do not cover the interval represented by this node, except if this node is the root. Note here we assume $K[0]$ and $K[count+1]$ equal to the left and right end points of the interval represented by this node respectively.

From the above description, it is clear that a Segment B-tree is merely a B-tree with three kinds of linked lists, which are used to store IDs of segments. Note in this paper a segment is of general type, i.e., it can be open or closed at either end. Thus it
is defined as a structure with the following fields:

- *left, right* : the left and right end points;
- *lopen, ropen*: bits indicating whether the segment is left/right open;
- *id*: an integer used for reference of the segment. It can also be used as a pointer to the segment.

For convenience, we refer $B[i]$ ($B[i-1]$) as the child/branch in the right (left) of $K[i]$; *iList*[*i*] (*eList*[*i*]) as the *iList* (*eList*) of $K[i]$; *cList*[*i*] as the *cList* in the right of $K[i]$ or *cList* of the interval ($K[i]$, $K[i+1]$).

Compared with a segment tree, a Segment B-tree is much more complicated. However, they are essentially very similar. The *cList* in a Segment B-tree is just like the ID list in a segment tree. The other two lists are used for some additional features, which will be discussed later. They can be discarded if those features are not needed.

It's rather strange that a Segment B-tree can have no nodes while not empty. This happens when all the intervals are ($-\infty$, ∞). In this case, it has a *cList*. Since it is very rare and not useful, we will assume a segment B-tree is either empty or has at least a root.

The following figure shows a Segment B-tree node format. Note that the lists in a node are arranged in the order of *iList*, *eList*, *cList*.

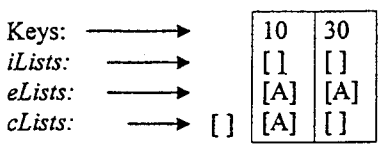


Figure 2. The segment B-tree node format

3. STATIC ALGORITHMS

In this section we give static algorithms for insertion and deletion. We also show how the Segment B-tree can be used for interval indexing. A static algorithm differs from a dynamic algorithm in that a static algorithm assumes that the set of possible end points is given and never changes. The static algorithms provide a basis for the dynamic algorithms, which will be discussed in the next chapter.

3.1 The Static Insertion Algorithm

As mentioned before, here we assume a set of possible end points is given. We first ignore all the lists and build a Segment B-tree that stores all the possible end points. This step is the same as for B-tree. When it finishes, all the lists are empty and the tree stores nothing. We can now insert a segment into the Segment B-tree. All the work an insertion algorithm need to do is to insert the ID of a segment into the corresponding lists. Below is a detailed description of the static insertion algorithm.

3.1.1 Insertion of Segments with Infinite Length

Insertion of ($-\infty$, ∞) is very simple. The segment ID is simply inserted into all the *cLists* and *iLists* in the root node.

We now consider the case that only one end point of the segment is infinity, say the segment is (x_0, ∞). An example is shown in Fig 3.1.1, where the segment A:(20, ∞) is to be inserted into the tree.

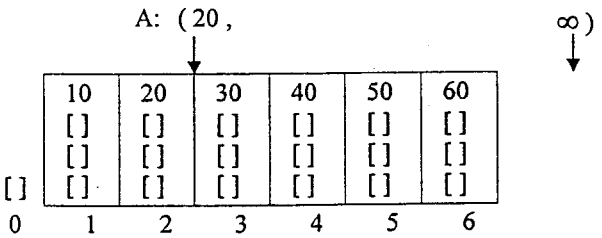


Figure 3.1.1 Insertion of segment (20, ∞)

We use a function to find the i, $0 \le i \le count$, such that $K[i] \le x_0 < K[i+1]$ (Note: for convenience, here we assume $K[0] = -\infty$ and $K[count+1] = \infty$). In the above example, $i = 2$. There are two possibilities: $x_0 = K[i]$ or $x_0 < K[i]$.

Next we need to update the lists. This step is fairly straightforward.

Case 1: $x_0 = K[i]$

It's easy to see that all keys in the right of $K[i]$ are in the segment, so we need to insert the segment ID, which is 'A',

into the *iLists* of *iList*[*i*], ... , *iList*[*count*]; Since *K*[*i*] is the left end point of segment A, 'A' is inserted into *eList*[*i*]; Finally we need to update the *cLists*. By definition if a segment covers the interval (*K*[*k*], *K*[*k*+1]) then the segment ID is in *cList*[*k*]. Thus, the segment ID should be inserted into the *cLists* of *cList*[*i*], ..., *cList*[*count*]. The result is shown in Fig 3.1.2.

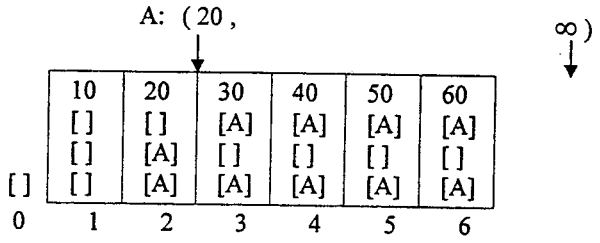


Figure 3.1.2 After insertion of segment A.

Case 2: $K[i] < x_0$

The analysis is almost the same as in case 1. In Fig 3.1.1, if we change segment A to (25, ∞), everything is the same except the *eList*[2] and cList[2] are not changed. However, we need to recursively insert segment A into the subtree rooted at *B*[*i*] (in the example it is B[2]). Since x_0 is in the Segment B-tree, so at some stage the recursion will end at case 1. Insertion of ($-\infty$, x_1) is similar and thusomitted.

3.1.2 Insertion of Segments with Finite Length

Now we discuss a more common case when the segment to be inserted is of finite length, say (x_0, x_1).

The first step is: compare x_0, x_1 with the keys in the root. We find some *i* and *j*, $0 \le i, j \le count+1$ such that $K[i] \le x_0 < K[i+1]$ and $K[j] \le x_1 < K[j+1]$. Again, here we assume *K*[0] is defined to be $-\infty$ and *K*[*count*+1] is ∞. We know that $i \le j$.

Case 1: $i < j$, $x_0 = K[i]$ **and** $x_1 = K[j]$

An example of this case is shown in Fig 3.1.3, where the picture before insertion of (20, 40) is shown in Fig 3.1.1.

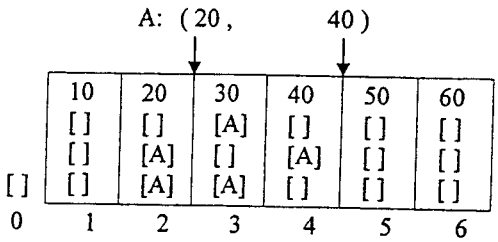


Fig 3.1.3 Insertion of segment (20, 40)

Both end points are found in the node. This case is very simple, we need to insert 'A' into the *iLists* of *iList*[*i*+1], ... , *iList*[*j*-1], dependent on whether the segment is closed or open, we may also need to update *iList*[*i*] and *iList*[*j*]. In this example since the segment is open in both ends, we do not need to update the *iLists*. We then insert 'A' into the end point lists of *eList*[*i*] and *eList*[*j*]. Finally, we insert 'A' into the cover lists of *cList*[*i*], ..., *cList*[*j*-1].

Case 2: $i < j$ **and** $x_0 = K[i]$ **and** $x_1 \neq K[j]$

An example of this case is shown in Fig 3.1.4.

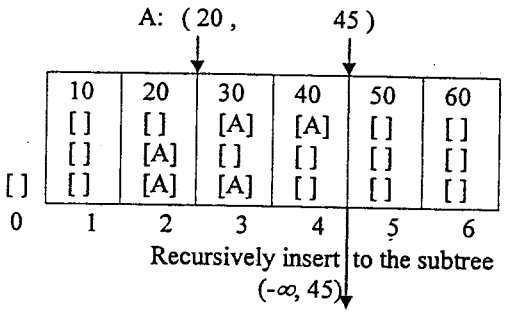


Fig 3.1.4 Insertion of the segment (20,45)

Similarly, we update the *iLists, eLists, cLists* as in case 1. After that, we recursively insert the segment A into the subtree rooted at *B*[*j*] (*B*[4] in this Example). To simplify the analysis, we can now think of the left end point of A to be $-\infty$. This is because the left end point of the interval corresponding to the subtree is covered by the segment. To this subtree, changing the left end point to $-\infty$ makes no difference. Thus the remaining question can be solved by the analysis used in 3.1.1.

Case 3: $i < j$ **and** $x_0 \neq K[i]$ **and** $x_1 = K[j]$

This is symmetric to case 2 and is skipped.

Case 4: $i < j$ **and** $x_0 \neq K[i]$, $x_1 \neq K[j]$

An example of this case is shown in Fig 3.1.5 and Fig 3.1.6.

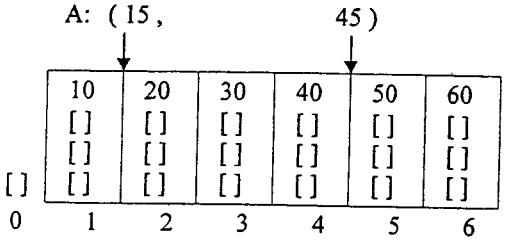


Figure 3.1.5 Before Insertion the segment A:(15, 45)

In this case $i = 1$, $j = 4$. It's easy to see that all keys between $K[i+1]$ and $K[j]$ (including $K[i+1]$ and $K[j]$) are in the segment A. So we insert 'A' into all the *iLists*: $iList[i+1]$, ..., $iList[j]$. Since neither 15 nor 45 is found in the node, so we do not need to update the *eLists*. Finally 'A' is inserted into *cLists* of $cList[i+1]$,..., $cList[j-1]$.

For our example, the result is:

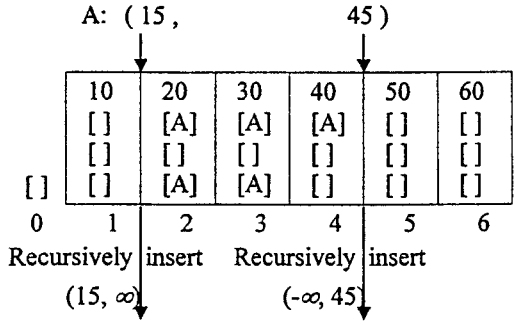


Figure 3.1.6 After insertion of A:(15, 45)

The next step is recursively insert A:(x_0, ∞) into the subtree rooted at $B[i]$ and insert A:(-∞, x_1) into the subtree rooted at $B[j]$. Again this step has been discussed in 3.1.1. (Note: We do not really change the end points of A).

Case 5: $i = j$ and $x_0 \neq K[i]$ and $x_1 \neq K[j]$

In this case, we do not need to make any change for the current node, but recursively insert A:(x_0, x_1) into the subtree rooted at $B[i]$.

Case 6: $i = j$ and $x_0 = x_1 = K[i]$

This is the degenerate case when the segment is simply a point. In this case, we only need to insert the segment ID into the *iList* and *eList* of this point. (Note: In this case the segment must be closed in both ends.)

3.1.3 Analysis of Space and Insertion Complexities

The space and insertion complexities of Segment B-tree are both near the same order as segment tree. A brief study is given below.

For a set of N segments, the space complexity of a Segment B-tree of order M is $O(MN\log_M N)$. The original B-tree requires a storage of only $O(N)$. The additional space required is for the linked lists. The *eLists* use $O(N)$ memory space. Each segment ID can only appear in the *cLists* / *iLists* of at most $2\log_M N$ nodes. Each node can have at most $M+1$ lists. Therefore, the total storage usage is $O(NM\log_M N)$. From this result, we can infer that the average length of *cLists* / *iLists* is $O(M\log_M N)$.

Insertion of a segment consists of recursively finding i, j and updating the lists. Recursively finding i and j takes at most $\log M \log_M N = \log N$ operations. Updating the lists takes at most $M\log_M N$ operations. The total complexity is $O(M\log_M N)$. Compared with a segment tree, both the space complexity and insertion complexity are $M/\log M$ times higher. However, M is usually not very large. A reasonable choice of M=16 causes the insertion and space complexity increase by 4 times. A smaller M could be used to ensure a good performance if the program is expected to run in core.

3.2 The Static Deletion Algorithm

The process of deletion algorithm is almost the same as in the insertion algorithm. Given a Segment B-tree rooted at the node *root*, we want to delete a segment A:(x_0, x_1) from the tree. It is also accomplished by recursion. All the steps are just the same as in insertion, except that we delete 'A' from each list where we insert 'A' in the insertion algorithm. However, the complexity for static deletion is not the same as that for static insertion. The reason is inserting a segment ID into a list requires only one operation, while deleting requires to find the ID first, which takes $O(L)$ operations, where L is the length of the list. As pointed out in 3.1.1, the average length of *iLists* and *eLists* is $O(M\log_M N)$, the complexity for deletion is $O(M^2 \log_M^2 N)$.

3.3 The Search Algorithms

There is no difference between static and dynamic search/query algorithms. Search algorithms are developed for solving query problems. Our data sturcture works well for alll types of queries, but due to space limitation we shall not discuss it here. Please refer to [6] for details.

4. DYNAMIC ALGORITHMS

Static algorithms are relatively easy since insertion and deletion do not change the underlying B-tree structure, hence they do not change other segment IDs that have been inserted in a list. However, they have some drawbacks and are not very useful in practice. The problems include: 1. Usually the set of segments is not fixed. So in general we just can not build the frame B-tree once and for all. Very

often we need to dynamically update the Segment B-tree. 2. After a segment is deleted, the end points still remain in the Segment B-tree, this causes waste of memory space and also makes the tree unnecessarily deeper and hence adversely affects the performance.

Dynamic algorithms must be developed to solve the above problems. Insertion of a new point may cause some nodes overflow and hence split, which changes the underlying B-tree structure, demanding some of the lists be updated. Similarly deletion of a segment may result one point no longer being an end point of any segment, hence this point should be removed from the tree. This again changes the underlying B-tree structure, and some of the lists should be updated.

We can develop dynamic algorithms from scratch. Probably it will be more efficient. But it is very complicated. So here we make use of static algorithms we have developed.

4.1 The Dynamic Insertion Algorithm

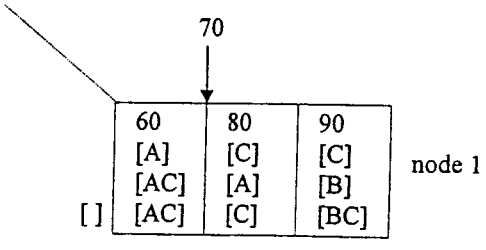
The dynamic insertion algorithm makes use of the static insertion algorithm. The algorithm has three steps:
1. Insert the left end point to the Segment B-Tree.
2. Insert the right end point to the Segment B-tree.
3. Update the *iLists*, *eLists* and *cLists*.

The third step is the same as in the static algorithm. However, the first two step is much more complicated.

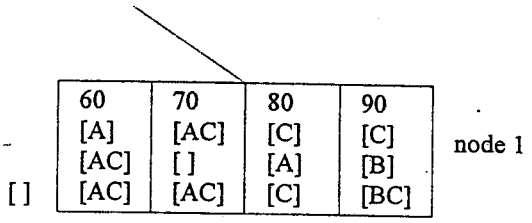
4.1.1 Dynamic Insertion of a New Point

We now discuss the first two steps. The problem is simply how to insert a new point into a Segment B-tree. We first give some examples and then give an outline of the algorithm at the end of this section.

Example 4.1.1: Fig 4.1.1 shows a fragment of a Segment B-tree with M=5. Suppose node 1 is a leaf node and 70 is to be inserted into the tree.
The first step is the same as insertion in a B-tree, the point 70 is first inserted into a leaf node. This can easily be done as in Fig 4.1.1.



a. Before point 70 is inserted



b. After point 70 is inserted

Figure 4.1.1 Insert 70 into the Segment B-tree with M=5

After the point is inserted into the leaf node, the number of keys is still less than MAX=4, so this node does not split. We now need to update the lists. First all lists in the right of 60 are moved right for 1 place. Since 70 is between 60 and 80, all segments that cover (60, 80) must also cover (60, 70), thus the *cList*[1] is not changed. Also all segments that cover (60, 80) must cover (70, 80), so the *cList*[2] is a copy of *cList*[1]. Finally, the *iList* for 70 is also a copy of the *cList* of (60, 80).

Example 4.1.2: This example is a bit more complex. Fig 4.1.2 shows a fragment of a Segment B-tree with M=5. Assume node 2 is the leaf node, where 70 is to be inserted. Fig 4.1.3 shows the segment B-tree after the insertion.

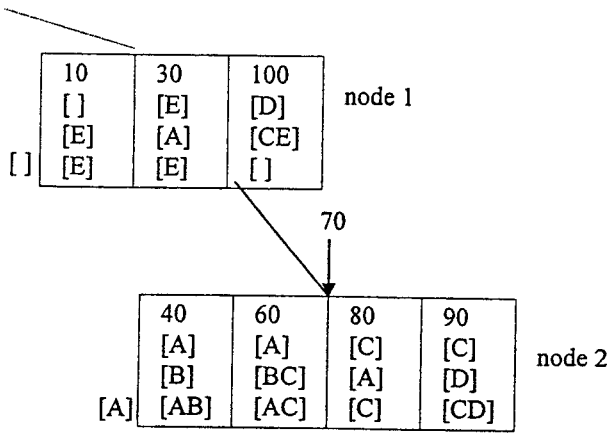


Figure 4.1.2 Before 70 is inserted. The relevant segments are{ A:(30, 80), B:(40, 60), C:(60, 100), D:(90, >100), E:(10, 100) }

In this example, after insertion the number of keys in the leaf node is greater than MAX=4. Thus the node need to be split. Nevertheless, we insert the point in the way as in the previous example, and the node comes to a temporary state, which need split soon. When the node is in this state, we update the lists. Then node 2 splits, and a new node 3 is

created. The tuple of (70, node 3, *iList*[3], *eList*[3]) is reinserted into its parent node.

Insertion of (70, node 3, *iList*[3], *eList*[3]) into node 1 is much harder than in the first example. We need to consider two factors. One is the same as in the previous example. We just copy the *cList* of (30, 100) to the *cLists* of (30,70) and (70, 100). Another factor is now the intervals represented by node 2 and 3 are smaller than the original node 2, some intervals do not cover (30, 100) may cover either (30, 70) or (70, 100). So we need to update the *cLists* in the right of 30 and 70, and also the *cLists* and *iLists* in the corresponding child nodes. We scan the *cList*[0] of node 2. If no segments in this list cover the interval represented by node 2, which is (30, 70), then we are done. Otherwise, the segments that cover (30, 70) are inserted into the *cList* of (30, 70) in node 1, and these segment IDs are removed from the *iLists* and *cLists* of node 2. This process is similarly applied to node 3. In this example, segment A covers the interval (30,70) and segment C covers the interval (70, 100), so 'A' is inserted into the *cList* of (30,70) and 'C' is inserted into the *cList* of (70, 100); And finally 'A' is deleted from the *cLists*/*iLists* in node 2 and the same is done for 'C' in node 3.

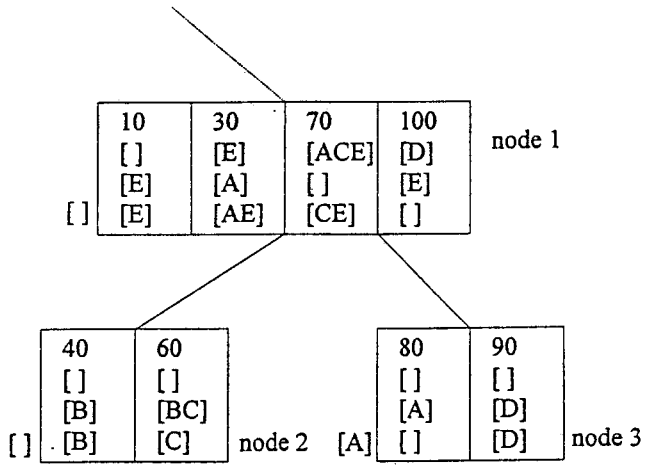


Figure 4.1.3 After insertion of point 70

4.1.2 Complexity Analysis

The complexity of the dynamic insertion algorithm is hard to estimate. So we just give a rough estimation. If a point is inserted into a leaf node and the node does not split, the complexity is $O(M\log_M N)$. The average length of *eLists* (*cLists*) is $O(M\log_M N)$. The total operations needed to split a node and update the $2M$-1 *iLists*/*cLists* is $O(M^2\log_M N)$. In the worst case, insertion of one point into a leaf node causes all the nodes from the leaf node to the root to split. That happens when the height of the B-tree is increased. The complexity in this case is $O(M^2\log_M^2 N)$. However, the probability for a node to split after insertion of one key is small. The number of keys in a B-tree ranges from $\lfloor M/2 \rfloor$ to M-1. Assume this number is uniformly distributed, then the probability for a node to have M-1 keys is about $2/M$ (A more accurate estimation on this probability is given in [5]. The results are very close). A node splits after insertion of a key only if it has M-1 keys before insertion. Thus the probability of splitting is only $2/M$. On average, the insertion complexity is

$$O(\frac{M-2}{M}\log_M N \cdot M \quad + \frac{2}{M}\log_M N \cdot M^2) = O(M\log_M N).$$

The performance is also dependent on the probability that different segments have common end points. If the probability is large, the depth of the tree is smaller and dynamic insertion becomes more like static insertion since only the third step is needed. In this case the insertion performance will be better.

4.2 The Dynamic Deletion Algorithm

The dynamic deletion algorithm also makes use of static deletion algorithm and has three steps:

1. Delete the segment ID from all lists that contain it.
2. Check if the *eList* corresponding to the left end point of the segment is empty. If it is empty, which means no other segments have this point as their end points. Then this point can be deleted from the segment tree, and thus delete it.
3. Apply step two with the right end point.

The first step is the same as static deletion, which only delete the segment ID from the tree. Step 2 and 3 are symmetric, we only consider how to delete a point from a Segment B-tree. The deletion algorithm first check whether the *eList* corresponding to this point is empty. If not, the step ends.

Due to space limitation, the details of dynamic deletion algorithms are not given here. Please refer to [6] for details.

5. CONCLUSION

In this paper we have introduced a new spatial data structure for interval indexing, which we call Segment B-tree. With this data structure we developed the algorithms for insertion, deletion and searching. The most impressive

property of this data structure is that it supports dynamic insertion and deletion while it is still very efficient in searching. We have given a theoretical analysis, which shows that the point query complexity of segment B-tree is exactly the same as that of segment tree. Also the insertion complexity is $O(M\log_M N)$, which is a bit greater than that of segment tree. The complexity of deletion algorithm is $O(M^2 \log_M^2 N)$, which is not as efficient as that for modified segment tree. However, Segment B-tree can also solve the intersection query problem in $O(\log N + n)$ while segment tree can not.

We have implemented the algorithms for insertion, deletion, and point query using Segment B-tree and our empirical results well match our theoretical analyses. The results are compared with the algorithms using segment tree and R-tree, which we have also implemented. All the comparisons are made when the programs run in core. We use the 2-3 Segment B-tree and 2-3 R-tree as the representatives in our tests, since they have the best performance when run in core. Our results show that Segment B-tree has almost the same performance as segment in search, but is worse than 2-3 R-tree in insertion and deletion. 2-3 R-tree has a very good performance in insertion and deletion, but it is poor in search. We are satisfied with the results because unlike segment tree, segment B-tree is a dynamic data structure. Since search is the most frequently performed operation, with a comparable performance in search as segment tree, segment B-tree is certainly the favor in a dynamic environment.

One possible way to solve the unsatisfactory performance in deletion is to modify the data structure in the same way as used in the (modified) segment tree. The method is by adding a linked list for each segment. The list links all the IDs of the segment in the Segment B-tree. Thus static deletion of a segment could be realized in $O(M\log_M N)$. However, unlike in a segment tree, change in the tree structure causes change of linked lists. This affects the performance of insertion and the last two steps of dynamic deletion. Further study need to be done on this problem.

Another possible way is to replace the linked lists by binary search trees. This can slow down the insertion performance to $O(M\log_M N \cdot \log(M\log_M N))$, but will improve the performance of deletion to $O(M\log_M N \cdot \log(M\log_M N))$. The actual performance is not tested.

Nevertheless, we believe the tradeoff is worthwhile even without those improvements we just mentioned. Static insertion is virtually useless in practice. Static deletion causes wastes of memory and makes the performance degenerate. The frequency of searching is the highest in practice, while insertion is the second and deletion is the lowest. However, each operation is important. Segment B-tree achieves the capability of dynamic updating at the small cost on memory usage and deletion. The overall system performance is still almost not affected.

Another special feature of this data structure is that it supports general types of segments, which most of other data structures do not.

REFERENCES

[1] J. L. Bentley, "Algorithms for Klee's Rectangle Problems", unpublished, Computer Science Department, Carnegie-Mellon University, 1977.

[2] Antonin Guttman, "R-trees: A Dynamic Index Structures for Spatial Searching", Proceedings of the SIGMOD conference, Boston, June 1984, pp45-47.

[3] H. Edelsbrunner, "A New Approach to Rectangle Intersections: Part I and II", International Journal of Computer Mathematics, vol.3-4,1983, pp209-229.

[4] E. M. McCreight, "Priority Search Trees", SIAM Journal on Computing, May, 1985, pp257-276.

[5] Hanan Samet, The Design and Analysis of Spatial Data Structures, Addison Wesley, 1990.

[6] W. Zhang, "Segment B-tree: A Dynamic and Efficient Interval Indexing Data Structure", Master Thesis, Southern Illinois University at Carbondale, 1997.