BITS-Tree -- An Efficient Data Structure for Segment Storage and Query Processing

K.S.Easwarakumar

Professor,Department of Computer Science & Engineering,

Anna University, Chennai 600 025, INDIA.

easwara@cs.annauniv.edu

T.Hema

Research Scholar,Department of Computer Science & Engineering,

Anna University, Chennai 600 025, INDIA.

hema@cs.annauniv.edu

ABSTRACT

In this paper, a new and novel data structure is proposed to dynamically insert and delete segments. Unlike the standard segment trees [3], the proposed data structure permits insertion of a segment with interval range beyond the interval range of the existing tree, which is the interval between minimum and maximum values of the end points of all the segments. Moreover, the number of nodes in the proposed tree is lesser as compared to the dynamic version of the standard segment trees, and is able to answer both stabbing and range queries practically much faster compared to the standard segment trees.

Indexing terms/Keywords

Segment Trees, Stabbing Query, Threaded Binary Tree, Height Balancing..

Academic Discipline And Sub-Disciplines

Computer Science; Computational Geometry

SUBJECT CLASSIFICATION

Advanced Data Structures Subject Classification; Geometric data structures

TYPE (METHOD/APPROACH)

Theoretical analysis

Council for Innovative Research

Peer Review Research Publishing System

1. INTRODUCTION

Solving geometrical problems computationally seems to be difficult due to its complex nature such as varying dimensionality and shape. Several algorithms are available in literature [4][8] for storing and retrieving geometrical objects. Segment tree [3] is one such data structure designed to handle intervals on the real line, whose extremes belong to a fixed set of abscissas. The static segment tree requires O(n log n) space, and the stabbing query processing time is O(k+log n), where n is the number of segments and k is the number of output segments. Though it looks reasonable in space and query processing, it does not support dynamic insertion and deletion of segments. However, in the dynamic segment trees, one can insert or delete a segment in the existing segment tree, but storage space requirements are likely to get increased as the range of the tree is not based on the segment end points. The range of the tree is predefined, and one can insert or delete the segments whose interval contained in this range, thus in those cases where the range of the segment falls beyond the range of the segment tree, reconstruction of the tree is the only possibility, which is not advisable. Also, this may practically increase the time as the number of nodes to be traversed get increased.

Variants of segment trees are used in packet classification problem. One such data structure is the Fat Inverted Segment Trees (FIS-trees) [5][9] with space requirements of $O(n^{1+1/l})$ and having complexity for insertion/deletion as $O(n^{1/l} \log n)$, where l is the height of the tree. Here, the tree is compressed (made `fat') by increasing the degree to more than two in order to decrease the height, however this has an upper bound on the total number of insertions and deletions allowed. Also, Agarwal et. al [2] proposed a linear-size data structure for the stabbing semi-group problem by combining the features of interval and segment tree, with the assumption that the end points of all intervals belong to a fixed set of points.

In this paper, a novel data structure, BITS-tree (**B**alanced **I**norder **T**hreaded **S**egment Tree), is proposed to dynamically insert and delete segments, with interval range of the node is only based on the end points of segments stored in the tree. In addition to this, it answers both stabbing and range queries efficiently. Unlike in the dynamic version of the standard segment trees, the root of the BITS-tree does not have the information of overall interval range of nodes in the tree, the BITS-tree also permits insertion of segment with any interval range, and thus it is a real dynamic tree suitable for all situations.

2. BITS-Tree

Definition 1: A BITS-tree is a height balanced two-way inorder-threaded binary tree T that satisfies the following properties.

1. Each node v of T associates with a range r and a list of segments containing the range r.

2. The range of a node cannot overlap with a range of any other node, other than at the end points.

3. The range of the nodes are sorted according to the inorder sequence.

4. It has a special node, called dummy, with range φ (empty), and its list is φ .

5. The inorder predecessor of the first node of the inorder sequence is dummy. Similarly, the inorder successor of the last node of the inorder sequence is dummy.

For example, consider the BITS-tree given in Figure 1. The inorder sequence of range is
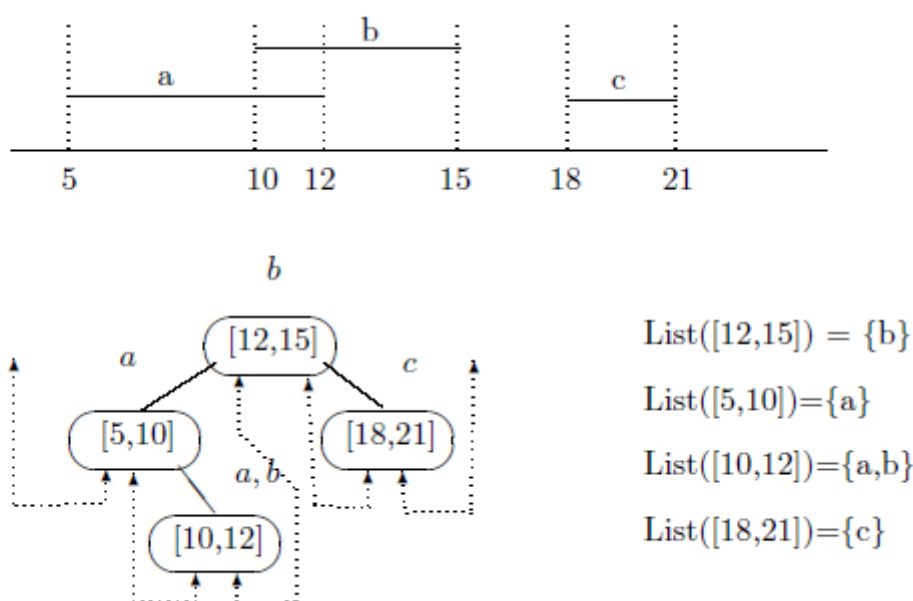

List([12,15]) = {b}

List([5,10])={a}

List([10,12])={a,b}

List([18,21])={c}

Figure 1: A BITS-Tree

[5,10], [10,12], [12,15] and [18,21], which is sorted in ascending order, as this satisfies the binary search tree properties on ranges. So, searching for a particular range could be done as in the case of binary search tree, using the following definition. The dotted lines in figure 1 denote threads. The hanging threads point to the dummy node.

Definition 2: Given two segments S = [p,q] and T = [m,n], S < T implies q ≤ m, that is the segment S falls left of T. Similarly, S > T implies n ≤ p, and this means S falls right of T.

Definition 3: Given two segments S = [p, q] and T = [m,n], where p ≠ q and m ≠ n, p < m < q < n denotes S left overlaps with T. Similarly, m < p < n < q denotes S right overlaps with T. Otherwise, S is said to be contained in T when m < p < q ≤ n or m ≤ p < q < n, or covered up T when p < m < n ≤ q or p ≤ m < n < q.

All four cases stated in definition 3 are commonly called as S and T overlaps. Note that, S left overlaps with T also implies T right overlaps with S. Similarly, S contained in T also implies T covered up T.

Definition 4: Given two overlapping segments S = [p,q] and T = [m,n], S U T and S ∩ T are respectively defined as [min{p,m}, max{q,n}] and [max{p,m}, min{q,n}].

Definition 5: Given two overlapping segments S and T, S U T, say [a,b], partitions into L(S,T), C(S,T) and R(S,T). Here, C(S,T) = S ∩T, say [p,q]. Now,

$$L(S,T) = \begin{cases} [a,p] & \text{if } a \neq p \\ \varphi & \text{Otherwise} \end{cases}$$

$$\text{and} \quad R(S,T) = \begin{cases} [q,b] & \text{if } q \neq b \\ \varphi & \text{Otherwise} \end{cases}$$

Definition 6: Let S = [p,q] and T = [m,n] be the two overlapping segments and their associated lists respectively be list(S) and list(T). Now, the list associated with C(S,T), L(S,T) and R(S,T) are defined as follows.

$$list(C(S,T)) = list(S) \cup list(T)$$

$$list(L(S,T)) = \begin{cases} list(S) & \text{if } p < m \\ list(T) & \text{if } p > m \\ \varphi & \text{Otherwise} \end{cases}$$

$$list(R(S,T)) = \begin{cases} list(S) & \text{if } n < q \\ list(T) & \text{if } n > q \\ \varphi & \text{Otherwise} \end{cases}$$

2.1 Insertion

For inserting a segment, say S, one has to search for a node, whose range overlaps with S, from root like in the binary search tree based on the relation given in definition 2. As soon as, an overlapping range is found in any node v, the insertion process starts as follows. Let R be the range of v. Now, S U R is partitioned into sub-ranges as described in definition 5. Now, range of v will be changed as C(S,R) and the segment S will be included in the list of v. Further, insertion of L(S,R) (if not empty) continues with the inorder predecessor of v, and insertion of R(S,R) (if not empty) continues with the inorder successor of v. However, in these two cases, the list of segments to be inserted may change, as in definition 6. Note that initially, the list of segments to be inserted contains only S, and later may be changed according to definition 6. At one point of time, the insertion may reach a null pointer (not always). If so, the balancing range (sub-range of segment not yet considered) will be considered as a range for a node, by creating a new one, with necessary thread pointers and having the list currently to be inserted. As the tree is balanced, creation of a new node may require necessary rotation to get the tree to be balanced. The rotations we performed here is only the AVL-tree based rotations [1]. The formal description of the insertion procedure is given in algorithm 1. The CreateNode() function used in algorithm 1, create a node as in threaded binary tree [6, 7], however here the threads are bidirectional.

For example, consider insertion of segment d = [2,7] in the BITS-tree given in figure 1. The search starts with the root node, and stops in the node with range r = [5,10] as this range overlaps with d, and d U r results in [2,5] U [[5, 7] U [7,10], as in definition 5. Here, L(d,r) = [2,5], C(d,r) = [5,7] and R(d,r) = [7,10]. Now, the range [5,7] is retained at that node by including d in the list of this node, and further insertion takes with respect to the ranges [2,5] and [7,10] at the inorder predecessor and successor, respectively.

Algorithm 1: BITS-Insert(T,S,L)

Input: T - the pointer to the root of the tree, S - the segment to be inserted; and L -Set of segments, initially L = {S}.

P← φ ;

[p,q] ← S;

if T ≠ NULL and T ≠ HEAD **then**

> **repeat**
>
> > [m,n] ← RANGE(T);
> >
> > C ← [m,n] ∩ [p,q];
> >
> > P← T;
> >
> > > **if** C = φ **then**
> > >
> > > > **if** [p,q] < [m, n] **then** T← LCHILD(T);
> > > >
> > > > **else** T←RCHILD(T);
> > > >
> > > **end**
> >
> > **until** C ≠ φ or T = NULL;
>
> **if** C ≠ φ **then**
>
> > RANGE(T) ← C;
> >
> > R ← LIST(T);
> >
> > LIST(T) ← LIST(T) U {[p,q]};
> >
> > **if** p ≠ m **then**
> >
> > > **if** p < m **then** R ←{[p,q]};
> > >
> > > BITS-Insert(PRED(T), [min{p,m}, max{p,m}],R);
> >
> > **end**
> >
> > **if** q ≠ n **then**
> >
> > > **if** q > n **then** R ←{[p,q]};
> > >
> > > BITS-Insert(SUCC(T), [min{q, n}, max{q,n}],R);
> >
> > **end**
>
> **end**

end

if T = NULL or T = HEAD **then**

> N ← CreateNode();
>
> RANGE(N) ← [p,q];
>
> LIST(N) ←L;
>
> > **if** P ≠ φ **then**
> >
> > > **if** RANGE(P) < RANGE(N) **then** RCHILD(P) ← N;
> > >
> > > **else** LCHILD(P) ← N;
> >
> > **end**
>
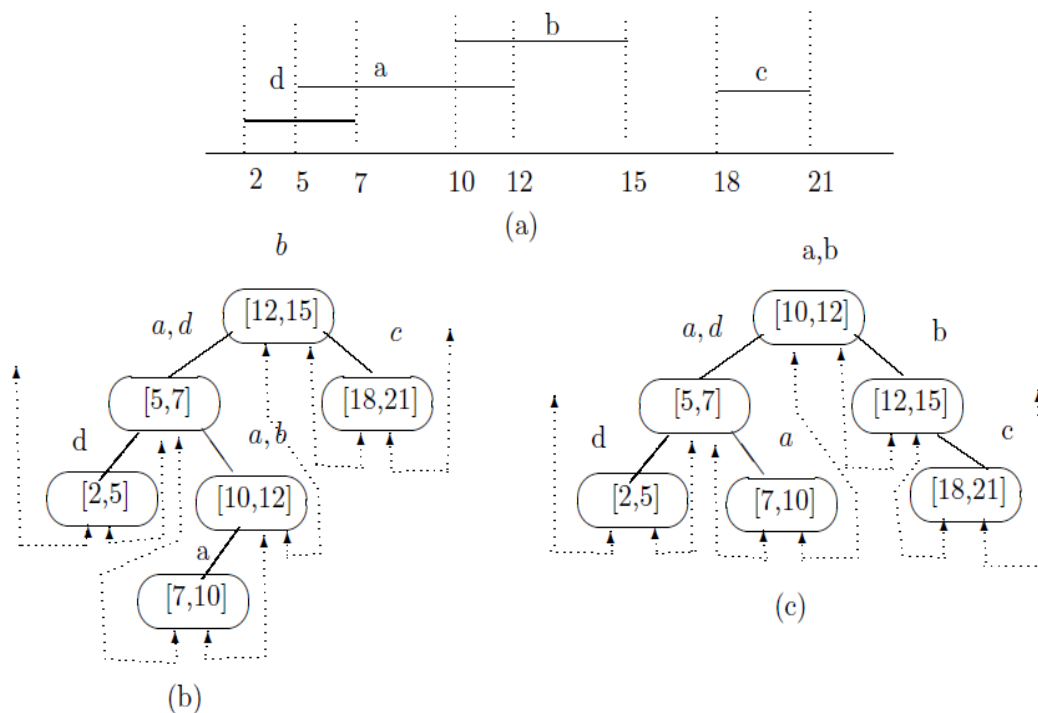> Perform rotation if required;

end

Figure 2: Insertion: (a) Set of segments including new segment d. (b) Insertion of segment d in BITS-tree in figure 1 before rotation. (c) BITS-Tree after rotation

This process in turn creates two new nodes in the tree given in figure 1, one with range [2,5] and other with range [7,10]. This is shown in figure 2(b). As creation of new node with range [2,5] does not affect the balancing factor, no rotation is required. However, creation of new node for the [7,10] affects the balancing factor (figure 2(b)), and hence necessary rotation to be carried out, here it is left-right rotation. The effect of this rotation is shown in figure 2(c). Note here that the thread pointers are not affected due to rotation. Whichever be the rotation, the thread pointer cannot change, as the rotation does not affect the inorder sequence. Note here that the usage of threads obviously reduces the time for inserting a segment, as compared to the standard dynamic version of the segment tree.

The time required for inserting a node is O(log n+k), where the first factor log n stands for locating first overlapping node, and k be the number of nodes in which the new segment get inserted. Note that, a segment tree contains at most 2n-1 nodes, where n is the number of segments in the tree.

2.2 Deletion

Let S = [p,q] be the segment to be deleted. First step of the deletion process is to locate a node v with range [p,-q], where -q denotes a value less than or equal to q. If the list of the node v does not contain the segment S, then deletion does not require, and the process of deletion can be terminated. Otherwise, the segment S can be removed from the list of v.Now, after removal of S, if LIST(v) is empty then convert the RANGE(v) as φ (empty),and v should be merged with PRED(v). Note that, during merge the node appears at lower level will be joined at the node at the higher level, and also it performs necessary rotation if required to make the tree balanced. In case, the LIST(v) is not empty after removal of S then compare the LIST(v) with the LIST(PRED(v)) and if found matching then merge the node v with PRED(v). Now, the deletion process can be continued with the SUCC(v), and that to be continued till the node w with range [p- q], where p- denotes a value greater than equal to p. Then, to complete the deletion, compare the new range of w with the SUCC(w), and if found matching then w and SUCC(w) must be merged. Each time when a node is merged with the inorder predecessor or successor, necessary rotation need to be performed as in the case of AVL trees.

Algorithm 2: BITS-Deletion

Input: T - the pointer to the root of the tree, and [p,q] - the segment to be deleted.

begin

> **while** RANGE(T) ≠ [p,-q] **do**
>
>> **if** RANGE(T) < [p, q] **then** T ← RCHILD(T);
>>
>>> **else** T ← LCHILD(T);
>
> **end**
>
> **repeat**
>
>> LIST(T) ← LIST(T) - {[p,q]};
>>
>> **if** LIST(T) = φ **then** RANGE ← φ ;
>>
>>> **if** LIST(T) = LIST(PRED(T)) or LIST(T) = φ **then**
>>> merge(T,PRED(T));
>>
>> **end**
>>
>>> **else** T ← SUCC(T);
>
> **until** RANGE(T) = [p˙, q];
>
>> **if** LIST(T) = LIST(SUCC(T)) **then** merge(T,SUCC(T));

end

As an example, suppose segment a=[5,12] is removed, as in figure 3(a). This process starts with node having range [5,7]. Now, after removal of segment a from this node, the list of this node matches with its inorder predecessor, thus to be merged with its inorder predecessor and the result is shown in figure 3(c). Next, the process continues with the node having range [7,10]. Now by removing segment a from this node, its list becomes empty and so its range to be modified as empty(φ) and required merging is to be done with its inorder predecessor, and the result is shown in figure 3(d). Finally, the segment a must be removed from the list of the node having range [10,12]. Here, the list of this node does not match with its inorder predecessor, and hence merging with the predecessor is not required, rather as a last step of the deletion process, its range is to be compared with the inorder successor, and as it is matching it should be merged with its inorder successor. The final result is shown in figure 3(e).

Like insertion, deletion also takes $O(\log n + k)$ time, where k is the number of nodes in which the segment to be deleted is present.

2.3 Query Processing

2.3.1 *Stabbing Query*

Given a point (one dimensional) p, it reports the segments containing p. On the other hand, it reports set of all segments intersects with the line x = p. In standard segment trees, the answer to this query lies in several nodes and one has to take union of lists of all those nodes to report the final answer. However, in BITS-tree, the answer lies in only one node. Here, the process is only to find the node whose range contains p, and the list of that node is the answer to the query. As the ranges of the nodes do not overlap, a simple binary search will do for locating the range that contains p. Since, BITS-tree is height balanced, $O(\log n)$ time is sufficient for locating a node with such range. Due to reduction in number of nodes and height as compared to the standard segment tree, our approach certainly outperforms considerably as compared to the existing one.

2.3.2 *Range Query*

Knowing set of segments that overlap with a given segment is an important problem in computational geometry. In BITS-tree, such queries can be answered very efficiently as compared to the standard segment trees. Here, given a segment s = [p,q], finding set of overlapping segments are determined by first locating a node u having
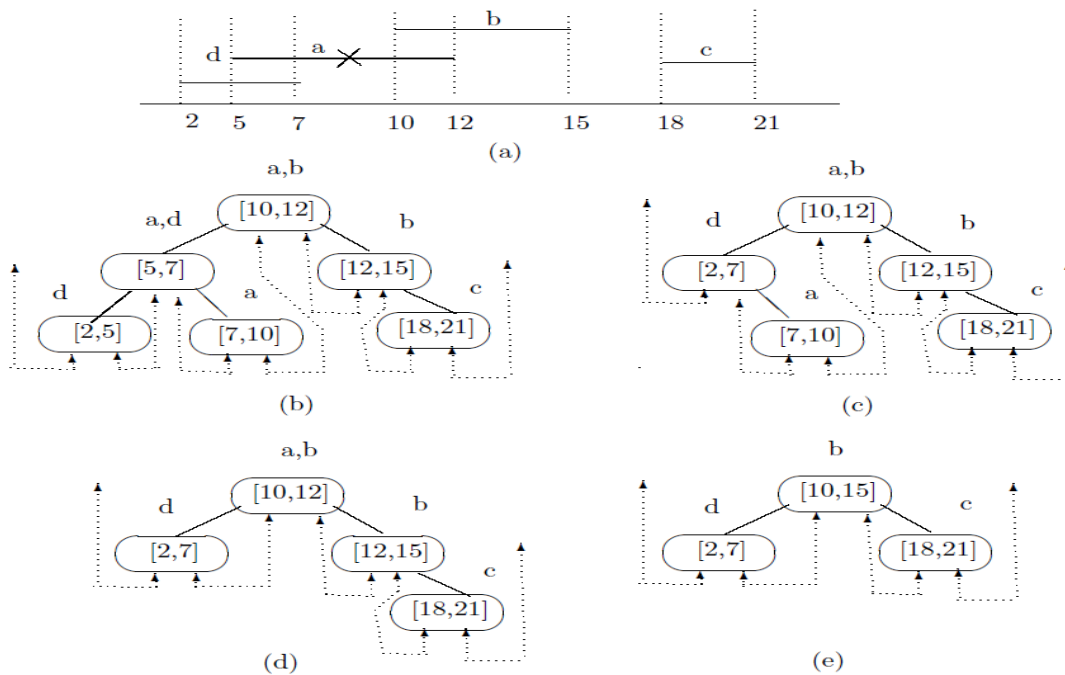
Figure 3: Deletion: (a)Set of segments including the segment a to be deleted. (b) Original BITS-tree. (c) After removal of segment a from node having range [5,7]. (d) After removal of segment a from node having range [7,10]. (e) After removal of segment a from node having range [10,12].

range which perfectly contains p, and traverse through the inorder successor till reaching the node v with range that perfectly contains q. Now the answer is the union of list of nodes traversed from u to v. The time required for the same is O(log n + k), where k is the number of nodes containing answer to the query.

3. COMPARISON

The static version of the standard segment tree of the segments, given in figure 1, is shown in figure 4. Similarly, the best dynamic version of the standard segment tree for the same set of segments is shown in figure 5. For this particular example, the comparison is shown in table 1. The maximum stabbing query time for the static segment tree comes when the query point is 15, and in the dynamic segment tree the maximum query time comes when the query point is 13. However, this analysis is not sufficient, and thus the factors related to the theoretical analysis with respect to the number of segments as n is shown in table 2, which are due to the following lemmas. For the following lemmas, let us assume SST and DST respectively represent the static and dynamic versions of the standard segment trees. Also, the terms trees and BITS trees are used interchangeably in rest of the paper.
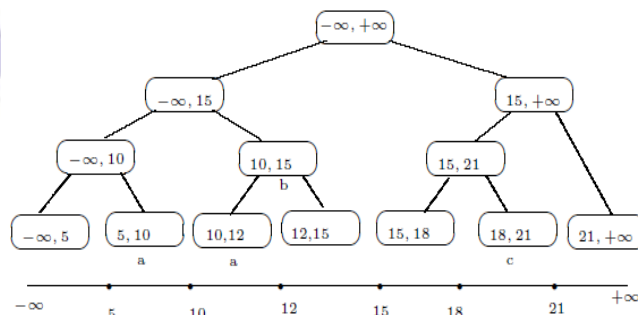

Figure 4: Static version of the standard segment tree for the segments shown in figure 1.
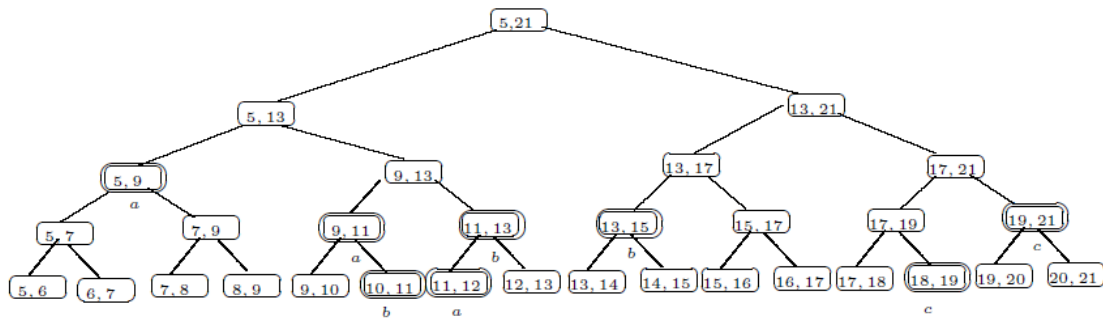
Figure 5: Dynamic version of the standard segment tree for the segments shown in figure 1.

Description	SST	DST	BITS
Number of nodes	13	31	4
Cumulative list size (for segment storage)	4	8	5
Height of the tree	3	4	2
Maximum stabbing query time	7	9	4

Table 1: Comparison of trees given in figures 1, 4, 5. SST denotes Static Segment Tree, DST denotes Dynamic Segment Tree and BITS denotes BITS Tree.

Lemma 1: The maximum number of nodes required for the BITS tree is $2n - 1$, where n is the number of segments.

Proof: The range of nodes in BITS trees are due to the end points of the segments, and these ranges are not overlapped with the ranges of other nodes in the tree. As n segments partition the range $[-\infty,+\infty]$ into $2n+1$ sub-ranges, the BITS tree can have the maximum of $2n-1$ nodes, since the BITS tree cannot have the nodes with range either $[-\infty, x]$ or $[y,+ \infty]$, where x and y are respectively $\min_{1 \leq i \leq n}\{x_i \mid s_i = [x_i,y_i]\}$ and $\max_{1 \leq i \leq n}\{y_i \mid s_i = [x_i,y_i]\}$ for the set of segments $\{s_1, s_2,\dots s_n\}$.

Lemma 2: Maximum storage required for maintaining the lists on all the nodes together in BITS tree is n^2.

Proof: The maximum storage required for maintaining the lists arises only when one of the node's (say, N) list with n segments and two nodes, other than N, each with $n-1, n-2,\dots,1$ segments. Thus, a total of $n + 2((n -1) + (n - 2) + \dots + 1)$, which is n^2, segments (may be with replication across the lists) stored altogether in the BITS tree.

Lemma 3: The height of the BITS trees does not exceed $\lceil \log n \rceil$.

Proof: The proof follows from the fact that the height of the AVL tree is less than $1.441 \lceil \log n \rceil$ [1].

Description	SST	DST	BITS
Maximum number of nodes	$4n+1$	$\geq 2(n_2- n_1) -1$	$2n-1$
Maximum cumulative list size (for segment storage)	$2n\lceil\log(2n+1)\rceil-1$	$2n\lceil\log(n_2- n_1)\rceil-1$	n^2
Maximum height of the tree	$\lceil\log(2n+1)\rceil$	$\lceil\log(n_2- n_1)\rceil$	$1.441\lceil\log(n)\rceil$
Maximum stabbing query time	$2\lceil\log(2n+1)\rceil-1+k$	$2\lceil\log(n_2- n_1)\rceil-1+k$	$1.441\lceil\log n\rceil+1+k$

Table 2: Theoretical comparison of static segment trees, dynamic segment trees and the proposed BITS trees. SST denotes Static Segment Trees, DST denotes Dynamic Segment Trees and BITS denotes BITS Trees.

Lemma 4: Stabbing query on BITS tree can be addressed within $\lceil \log n \rceil +1+k$ time, where k is the number of output segments.

Proof: As the height of the tree is $\lceil \log n \rceil$, to locate a node whose range contains the query point requires $O(\lceil \log n \rceil)$ time. Once such a node is found, the segments that exist in the list of that node becomes the answer, and thus, we add $\varTheta (k)$ time to output the segments.However, if the query point is exactly one of end points of a segment, it is required to access

either the inorder successor or the inorder predecessor of the node to list out the segments.In this case, it requires an additional $\Theta(1)$ time to visit that node using the threads.Thus, the stabbing query can be addressed within the time of $\lceil \log n \rceil + 1 + k$.

Lemma 5: Let [n1,n2] be the range of root of a DST T, and n be the number of segments stored in T. Then, $n \le n_2 - n_1$.

Proof: Let k be the number of leaf nodes in T. Let the range of i-th leaf be $[a_i, b_i]$, where $1 \le i \le k$. It is clear that the range $[a_i, b_i]$ of the leaf in DST satisfies $b_i - a_i = 1$. Also, the number of leaf nodes in T is $n_2 - n_1$, which is the disjoint union of ranges of unit length across the range of the root.Thus, if $n > n_2 - n_1$, then there should at least one segment, say $s_i = [x_1, x_2]$, exists by satisfying $x_2 - x_1 < 1$, which is impossible.Thus, $n \le n_2 - n_1$.

Lemma 6: BITS trees outperforms SST and DST in case of range search.

Proof: Given a range, say $r = [x_1, x_2]$, the range search has to find the number of segments overlapped with r. In BITS tree, this is done by locating the node N_1, whose range contains x_1, then by visiting the nodes using inorder successor threads till reaching the node N_2 with the range containing x_2. Here, the segments stored in the list of nodes between N_1 and N_2 in the inorder sequence will be the answer. This process needs to visit $O(\log n + p)$ nodes, where p is the number of nodes lying between N_1 and N_2 in the inorder sequence. Note that, $\log n + p \le m$, where m is the number of nodes in the tree, as a node visited once cannot be visited again.

However, it is impossible to use inorder sequence in standard segment trees due to the following two reasons.

- The SST or BST does not have inorder thread to efficiently access the next node in the inorder sequence.
- The range of the nodes are overlapped with its ancestors.

Thus, to do range search on SST or DST, one has to use a known binary tree traversal method to answer the query, and that takes the time of $O(m)$, where m is the number of nodes in the tree. This is due to the fact that the properties of segment trees are not useful, in this case, to find the range search.Thus, the BITS trees outperforms both SST and DST in case of answering the range query.

The above analysis shows that BITS trees are better, as compared to the standard segment trees, except in list size requirements. Asymptotically, it is $O(n^2)$ for BITS trees and $O(n \log n)$ for the SST, and may be more for DST as the number of segments does not determine this. However, this is not a major issue, as in any search tree, the time taken to query processing is more important than the space requirements. Once the segments are stored, the overall complexity relies on the number of queries performed. Moreover, in BITS tree the answer (for stabbing query) lies in only one node, but in SST or DST it may spread in several nodes. Due to reduction of height and number of nodes in the tree, it is sure that the BITS trees are much better compared to the standard segment trees.

4. CONCLUSION

The BITS-tree, proposed in this paper, is a balanced two-way inorder threaded segment tree, which is useful for storing geometric data objects available in one dimension. It handles both stabbing and range queries very efficiently than the standard segment tress. The BITS-trees are not restricted to any bound in range, unlike in the standard segment trees, and that is the main advantage of this structure. Thus, one can insert a segment of any range without knowing the overall range of the segment tree. Moreover, it is quite possible to generalize this to higher dimension with slight modifications in the data representation.

REFERENCES

[1] G. Adelson-Velskii and E.M. Landis 1962. An algorithm for the organization of information,Soviet Mathematics.

[2] P.K. Agarwal, L. Arge, H. Kaplan, E. Molad, R.E Tarjan, and K. Yi.,2012. An optimal dynamic data structure for stabbing-semigroup queries. SIAM J. Comput.41(1),104-127.

[3] J.L. Bentley. 1977.Solutions to klee's rectangle problems. Technical report, Carnegie-Mellon Univ., Pittsburgh, PA.

[4] M.D. Berg, O. Cheong, M.V. Kreveld, and M. Overmars. 2008. Computational Geometry:algorithms and applications. Springer-Verlag, New York,USA, third edition.

[5] A. Feldman and S. Muthukrishnan. 2000. Tradeoffs for packet classification. In Proc. of the IEEE Computer and Communication Societies, 3,1193-1202.

[6] D. Knuth. 1973. Art of Computer Programming: Fundamental Algorithms. Addison Wesley.

[7] A. Perlis and C. Thronton. 1960. Symbol manipulation by threaded lists. Communications of ACM, 3(4),195-204.

[8] F.P. Preparata and M.L. Shamos. 1985. Computational Geometry. An Introduction. Springer-Verlag, New York.

[9] M. Thorup.2003. Space efficient dynamic stabbing with fast queries. In Proc. of STOC'03, 3,649-658.