The Interval Skip List: A Data Structure for Finding All Intervals That Overlap a Point*†

Eric N. Hanson
Theodore Johnson
Computer and Information Sciences Department
University of Florida
Gainesville, FL 32611

UF-CIS-92-016
16 June 1992

Abstract

A problem that arises in computational geometry, pattern matching, and other applications is the need to quickly determine which of a collection of intervals overlap a point. Requests of this type are called *stabbing queries*. A recently discovered randomized data structure called the *skip list* can maintain ordered sets efficiently, just as balanced binary search trees can, but is much simpler to implement than balanced trees. This paper introduces an extension of the skip list called the *interval skip list*, or IS-list, to support interval indexing. The IS-list allows stabbing queries and dynamic insertion and deletion of intervals. A stabbing query using an IS-list containing n intervals takes an expected time of $O(\log n)$. Inserting or deleting an interval in an IS-list takes an expected time of $O(\log^2 n)$ if the interval endpoints are chosen from a continuous distribution. Moreover, the IS-list inherits much of the simplicity of the skip list – it can be implemented in a relatively small amount of high-level language code compared with dynamic interval indexes based on balanced trees.

1 Introduction

An important problem that arises in a number of computer applications is the need to find all members of a set of intervals that overlap a particular point. Queries of this kind are called *stabbing queries* [Sam90]. This paper introduces a data structure called the *interval skip list* (IS-list), which is designed to handle stabbing queries efficiently. The IS-list is an extension of the randomized list structure known as the *skip list* recently discovered by Pugh [Pug90]. In Section 2, other methods for solving stabbing queries are discussed. Section 3 describes the interval skip list data structure and methods for searching and updating it. Section 4 gives an analysis of the complexity of algorithms for manipulating IS-lists. Finally, Section 5 presents conclusions.

2 Review of Stabbing Query Solution Methods

Formally, we describe the stabbing query problem as the need to find all intervals in the set $Q = \{i_1, i_2, ...i_n\}$ which overlap a query point X. We will use a notation for intervals that indicates a pair of values with inclusive boundaries by square brackets, and non-inclusive boundaries by parentheses. Open intervals have one boundary at positive or negative infinity, and points have both boundaries equal. Examples of intervals are [17,19), [12,12], [-inf,22]. Several different approaches to solving the stabbing query problem have been developed. The most trivial solution is to place all n intervals in Q in a list and traverse the list sequentially, checking each interval to see if it overlaps the query point. This algorithm has a search complexity of $O(n)$.

A more sophisticated approach is based on the *segment tree* [Sam90]. To form a segment tree, the set of all end points of intervals in Q is formed, and an ordered complete binary tree is built that has the end points as its leaves. To index an interval, the identifier of the interval is placed on the uppermost nodes in

*This work was supported in part by the Air Force Office of Scientific Research under grant number AFOSR-89-0286.
†A preliminary version of this paper appeared in the *Proceedings of the 1991 Workshop on Algorithms and Data Structures*, Ottawa, Canada, Springer Verlag.

the tree such that all values in the subtrees rooted at those nodes lie completely within the interval. In this way, an interval of any length can be covered using $O(\log n)$ identifiers. Hence, the segment tree requires $O(n \log n)$ storage. In order to solve a stabbing query using a segment tree, the tree is traversed from the root to the location the query value X would occupy at the bottom of the tree. The interval identifiers on all nodes visited are returned as the answer to the query. A query takes $O(\log n)$ time. The segment tree works well in a static environment, but is not adequate when it is necessary to dynamically add and delete intervals in the tree while processing queries.

Another data structure that can be used to process stabbing queries is the *interval tree* [Ede83a, Ede83b]. Unfortunately, as with the segment tree, all the intervals must be known in advance to construct an interval tree.

A data structure that can index intervals dynamically is the R-tree [Gut84]. R-trees are a multi-dimensional extension of B-trees in which each tree node contains a set of possibly overlapping n-dimensional rectangles. Subtrees of each index node contain only data that lies within a containing rectangle in the index node. Since rectangles in each node may overlap, on searching or updating the tree it may be necessary to examine more than one subtree of any node. An important part of the R-tree algorithm involves use of heuristics to decide how to partition the rectangles in a subtree to determine the best set of index rectangles for an index node. Due to its generality, and the indexing heuristics required, the R-tree is challenging to implement. A useful property of R-trees is that they require only $O(n)$ space. Their performance should be good for rectangles (or intervals in the 1-dimensional case) with low overlap, but when there is heavy overlap, search time can degenerate rapidly.

Another data structure which solves the stabbing query problem efficiently (among others), and does allow dynamic insertion and deletion of intervals is the *priority search tree* [McC85]. An advantage of the priority search tree is that it requires only $O(n)$ space to index n intervals. However, the priority search tree in its balanced form is very complex to implement [Wir86]. In addition, for a priority search tree to handle a set of intervals with non-unique lower bounds, a special transformation must be used to transform the set of intervals into one where the intervals have unique lower bounds. This transformation is not trivial, and it must be created for each different data type to be indexed.

The *interval binary search tree* (IBS-tree) can handle stabbing queries, and can be balanced more easily and is easier to implement than the priority search tree, although it requires $O(n \log n)$ storage [HC90]. We conjecture that balanced IBS-trees require $O(\log n)$ time for searching and $O(\log^2 n)$ average time for insertion and deletion, though a definitive performance analysis has not been done. A data structure closely related to the IBS-tree called the *stabbing tree* has been developed to find the *stabbing number* for a point given a collection of intervals [GMW83]. The stabbing number is the number of intervals that overlap a point. In contrast, the IBS-tree and the IS-list return a *stabbing set* containing all the intervals overlapping the query point, not just the number.

The interval skip list is quite similar in principle to the IBS-tree, but it inherits the simplicity of skip lists, making it much easier to implement than balanced IBS-trees. In the next section, we present the details of the IS-list.

3 Interval Skip Lists

In this section we introduce a method for augmenting a skip list with information to make it possible to rapidly find all intervals that overlap a query point. The IS-list can accommodate points as well as open and closed intervals with inclusive and exclusive boundaries. We will review the skip list data structure [Pug90] and then discuss the extensions needed to index intervals.

3.1 Review of Skip Lists

The skip list is similar to a linked list, except that each node on the list can have one or more forward pointers instead of just one forward pointer. The number of forward pointers the node has is called the *level* of the node. When a new node is allocated during a list insertion, its level is chosen at random, independently of

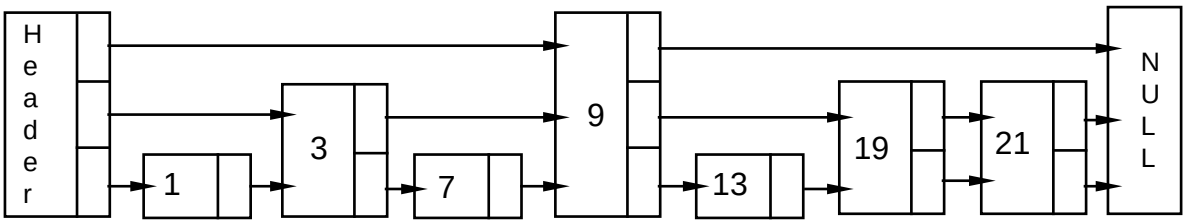


Figure 1: Example of a skip list.

the levels of other nodes. The probability a new node has x levels is

$$P(x = k) = \begin{cases} 0 & \text{for } k < 1 \\ (1-p) \cdot p^{k-1} & \text{for } k \geq 1 \end{cases}$$

where $p \in (0, 1)$ parameterizes the skip list. With $p = 1/2$, the distribution of node levels will allocate approximately $1/2$ the nodes with one forward pointer, $1/4$ with two forward pointers, $1/8$ with three forward pointers, and so on.

A skip list is normally organized with values in increasing order. A node's pointer at level l points to the next node with l or more forward pointers. An example of a skip list is shown in figure 1. Searching in a skip list involves "stair-stepping" down from the beginning of the list to the location of the search key. The process of searching a skip list for a search key K begins at the list header at the level i equal to the maximum level of a node in the list. Assume that the current node being visited is called y (y initially is the header). If the value of the key of the node pointed to by the level i pointer of y is $\geq K$, i is set to $i - 1$. Otherwise, y is set to be the node pointed to by the level i forward pointer of the current node. The search continues in this fashion until $i = 0$, at which point the node immediately after y is either has a key equal to K, or else K is not present in the list and it would be located immediately after y.

Insertion and deletion in skip lists involves simply searching and splicing. The splicing operation is supported by maintaining an array of nodes whose forward pointers need to be adjusted. For a full description of the algorithms for maintaining skip lists and skip lists extended to support additional capabilities such as searching with fingers, efficient merging, finding the kth item in a list etc. the reader is referred to [Pug90, Pug89].

The performance of skip lists is quite similar to that of balanced binary search trees. The expected value of times for searching, insertion and deletion in a skip list with n elements are all $O(\log n)$. The variance of the search times is also quite low, making the probability that a search will take significantly longer than $\log n$ time vanishingly small. Comparing actual implementations of skip lists and AVL trees [AVL62], skip lists perform as well as or better than highly-tuned non-recursive implementations of AVL trees, yet programmers tend to agree that skip lists are significantly easier to implement than AVL trees [Pug90]. A discussion of extensions to skip lists to support interval indexing is given below.

3.2 Extending Skip Lists to Support Intervals

The basic idea behind the interval skip list is to build a skip list containing all the end points of a collection of intervals, and in addition to place markers on nodes and forward edges in the skip list to "cover" each interval. The placement of markers on edges and nodes in an interval skip list can be stated in terms of the following invariant:

Interval skip list marker invariant: Consider an interval $I = (A,B)$ to be indexed. End points A and B are already inserted into the list. Consider some forward edge in the skip list from a node with value X to another node with value Y. The interval represented by this edge is (X,Y). A marker containing the identifier of I will be placed on edge (X,Y) if and only the following conditions hold:

1. **containment:** I contains the interval (X,Y).

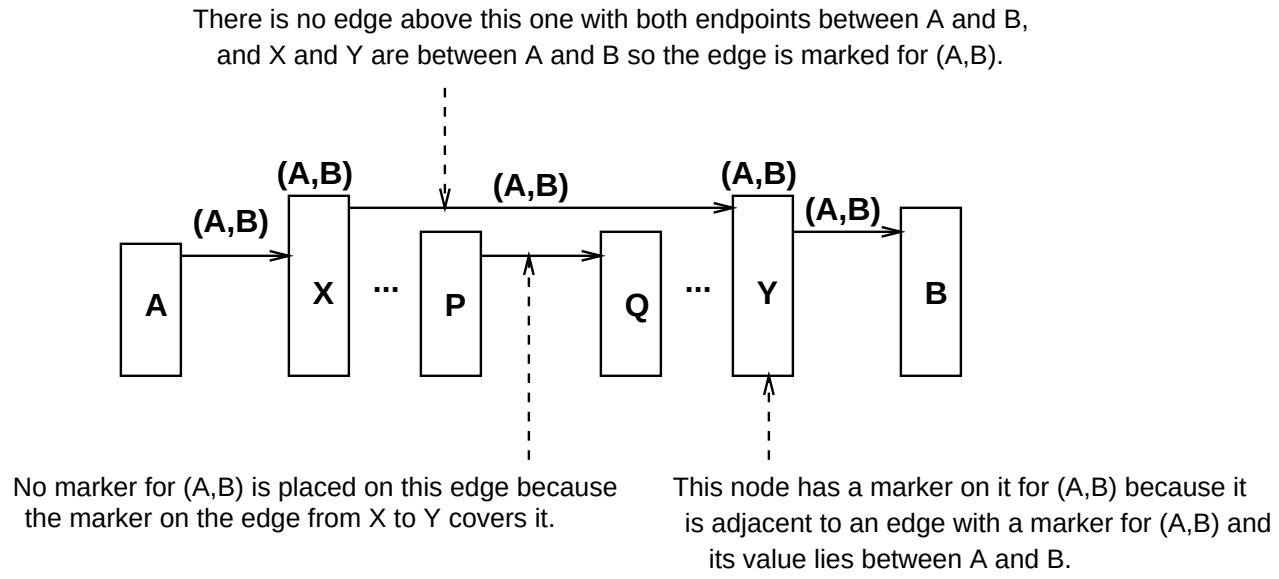


Figure 2: An example illustrating the interval skip list marker placement invariant.

2. **maximality:** There is no forward pointer in the list corresponding to an interval (X', Y') that lies within I and contains (X,Y).

In addition, if a marker for I is placed on an edge, then the nodes that are the endpoints of that edge and have a value contained in I will also have a mark placed on them for I. Open intervals are represented as closed intervals with one inclusive boundary at infinity, e.g., $(7,\infty]$. Hence without loss of generality only closed intervals are considered. A diagram illustrating the application of the invariant is shown in 2.

An example of a set of intervals and the IS-list for those intervals is shown in figure 3. Searching an IS-list to find all intervals that overlap a search key can be done efficiently given a skip list with markers on it satisfying this invariant. The challenge in inserting and deleting intervals into an interval skip list is to perform the operations efficiently while maintaining the invariant. The remainder of this section describes the procedures for searching, insertion and deletion in IS-lists.

3.3 Searching

The procedure to search an IS-list L to find all intervals that overlap a search key K, and return those intervals in a set S, is to search along the same path that would be visited by the standard skip list search procedure, and add markers to S as the search proceeds. Whenever the procedure drops from level i to $i-1$ during the search, it adds to S the markers on the forward pointer at level i of the current node. This is valid since markers on the forward pointer at level i must belong to an interval that contains K. At the final destination, if K is present in the list, the procedure adds the markers on node K to S. Otherwise, (when K is not present) it adds the markers on the lowest pointer of the current node to S. When the search terminates, exactly one marker for every interval that overlaps K will be in S. No duplicates will be found. Each node of level i in an interval skip list contains the following:

key: a key value,

forward: an array of forward pointers, indexed from 0 to $i-1$, as in a regular skip list,

markers: an array of sets of markers, indexed from 0 to $i-1$,

owners: a bag (multi-set) of identifiers of the intervals that have an endpoint equal to the key value of this node (one interval identifier can appear twice here if the interval is a point),

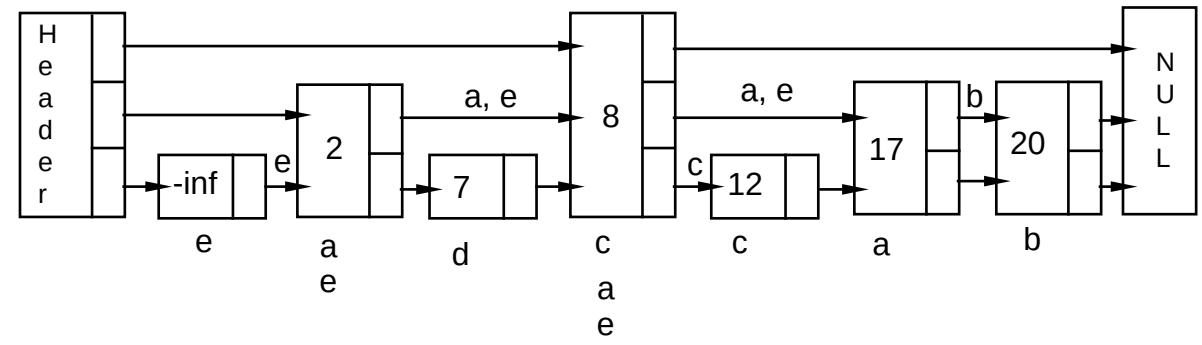


Figure 3: Example of an interval skip list for intervals shown.

eqMarkers: a set of markers for intervals that have a marker on an edge that ends on this node, and which contain the key value of this node.

An outline of an implementation of this search algorithm is shown as the procedure findIntervals(K,L,S) below.

```
procedure findIntervals(K,L,S)
    x := L.header; S := φ
    // Step down to bottom level.
    for i:=maxLevel down to 1 do
        // Search forward on current level as far as possible.
        while (x→forward[i] ≠ null and x→forward[i]→key < K) do
            x := x→forward[i]
        // Pick up interval markers on edge when dropping down a level.
        S := S ∪ x→markers[i]
    end

    // Scan forward on bottom level to find location where search key will lie.
    while (x→forward[0] ≠ null and x→forward[0]→key < K) do
        x := x→forward[0]

    // If K is not in list, pick up interval markers on edge,
    // otherwise pick up markers on node with value = K.
    if (x→forward[0] = null or x→forward[0]→key ≠ K)
        S := S ∪ x→markers[0]
    else
        S := S ∪ x→forward[0]→eqMarkers
end findIntervals
```

In an actual implementation, the set S of matching intervals can be constructed by building a list of pointers to sets of markers that reside on

1. individual forward pointers and
2. perhaps the final node visited.

The union operations in findIntervals require simply appending a single value to the list representing S. This value is a pointer to a mark set being added to S. This operation requires only $O(1)$ time per level in the IS-list. Duplicates do not have to be removed from S, because it is not possible to add a marker for the same interval to S more than once. This is true since descending past two edges with a marker for the same interval on them during a search would imply that the IS-list marker invariant was violated, which is a contradiction. The last **if** statement in the search algorithm prevents adding any duplicate markers to S at the bottom level of the IS-list.[‡] We now turn to a discussion of the algorithm for inserting an interval into an IS-list.

3.4 Insertion

To insert an interval (A,B) into an IS-list, the first step is to insert A and B separately if they are not already in the list and adjust existing markers as necessary. The next step is to start at A, search for B, and place markers for (A,B) in a way that satisfies the marker invariant.

To place an interval end-point A into the list, the first step is to use the standard interval skip list insertion algorithm [Pug90] to insert A. During this step one must save a pointer to the new IS-list node containing A (call this N) and save the **updated** array containing pointers to the nodes with pointers to N that had to be adjusted when A was inserted. The next step is to adjust the markers so that the IS-list marker invariant is maintained. An important observation is that markers can only stay at the same level or go up to a higher level after an insertion. They never move down. The procedure shown below adjusts markers to maintain the invariant after insertion of a node. It first places markers on the outgoing edges from N, raising them to higher levels as necessary. Then it raises markers on edges leading into N as necessary. In the procedure, the function level(x) returns the number of forward pointers of node x.[§]

procedure adjustMarkersOnInsert(L,N,updated)
// Update the IS-list L to satisfy the marker invariant.

// Input: IS-list L, new node N, vector 'updated' of nodes with updated pointers.
// The value of updated[i] is a pointer to the node whose level i edge was changed to point to N.

// Phase 1: place markers on edges leading out of N as needed.

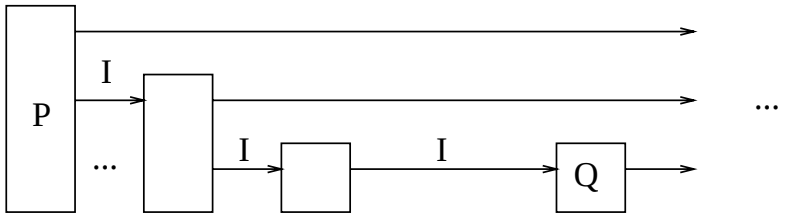
// Starting at bottom level, place markers on outgoing level i edge of N.
// If a marker has to be promoted from level i to i+1 or higher, place
// it in the promoted set at each step.

promoted := ϕ // make set of promoted markers initially empty
newPromoted := ϕ // temporary set to hold newly promoted markers

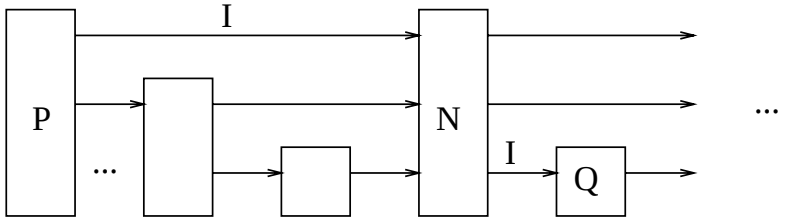
[‡] If preferred, one may choose to implement the construction of S by traversing each set of markers added to S and adding the markers individually to a list of markers representing S. This method will add additional time $O(|S|)$ to the total search time. Since most applications would have to traverse the list of markers returned anyway, constructing S this way would not normally affect the order of growth of the running time of the application, and it might be more convenient from a software engineering perspective.

[§] In the algorithms for insertion and deletion that follow, for simplicity we do not explicitly state how the eqMarkers are manipulated. It is assumed that when a marker is placed on an edge, it will be placed in the eqMarkers sets of a node on either end of the edge if the interval for the marker covers the node. Similarly, when a marker is removed from an edge markers will be removed from eqMarkers sets on nodes adjacent to the edge.

```
for i := 0 to level(N) - 2 do
begin
    for m in updated[i]→markers[i] do
    begin
        if the interval of m contains (N→key,N→forward[i+1]→key)
        then // promote m
            remove m from the level i path from N→forward[i] to N→forward[i+1]
            and add m to newPromoted
        else
            place m on the level i edge out of N
    end

    for m in promoted do
    begin
        if the interval of m does not contain (N→key,N→forward[i+1]→key)
        then // m does not need to be promoted higher
            place m on the level i edge out of N and remove m from promoted
        else // continue to promote m
            remove m from the level i path from N→forward[i] to N→forward[i+1]
    end

    promoted := promoted ∪ newPromoted
    newPromoted := φ

end

// Combine the promoted set and updated[level(N)-1]→markers[level(N)-1]
// and install them as the set of markers on the top edge out of N.
LN := level(N)-1
N→markers[LN] := promoted ∪ updated[LN]→markers[LN]

// Phase 2: adjust markers to the left of N as needed.

// Markers on edges leading into N may need to be promoted as
// high as the top edge coming into N, but never higher.

promoted := φ
newPromoted := φ

for i := 0 to level(N)-2 do
begin
    for each mark m in updated[i]→markers[i] do
        if m needs to be promoted (i.e. m's interval contains (updated[i+1]→key,N→ key))
        then begin
            place m in newPromoted.
            remove m from the path of level[i] edges between
                updated[i+1] and N (it will be on all those edges
                or else the invariant would have previously been violated).
        end

    for each mark m in promoted do
        if m belongs at this level, (i.e. m's interval covers (updated[i]→key,N→key)
                but not (updated[i+1]→key,N→key))
```



Markers placed for interval I=(P,Q) on sample IS-list.



Markers placed for I after insertion of node N and marker adjustment.

Figure 4: Example of node insertion and marker promotion

then place m on the level i edge between updated[i] and N,
and remove m from promoted.
else strip m from the level i path from updated[i+1] to N.

promoted := promoted ∪ newPromoted
newPromoted := ϕ
end

// Put all marks in the promoted set on the uppermost edge coming into N.
top := level(N)-1
updated[top]→markers[top] := updated[top]→markers[top] ∪ promoted

end adjustMarkersOnInsert

An example of insertion of a node N and the corresponding promotion of markers in an example IS-list that would be accomplished by adjustMarkersOnInsert is shown in figure 4. A key feature of this procedure is that the time taken to examine a marker that is not promoted is $O(1)$. This fact is important for the overall performance of the insertion operation, as will be discussed in section 4.

Placing markers to cover the inserted interval (A,B) is accomplished by following forward pointers from A to B along the path defined by the IS-list marker invariant. In general this will involve stepping up several levels in the list from A and then stepping back down to B. An example of the general case is shown in figure 5. There are also special cases in which it is only necessary to step down or up or proceed on the same level from A to B. Let I be an interval with lower and upper endpoints A and B respectively. The procedure to place markers for I on an interval skip list L that already contains endpoints A and B is shown below.

procedure placeMarkers(L,I)
begin
// mark non-descending path
x := search(L,I.left)

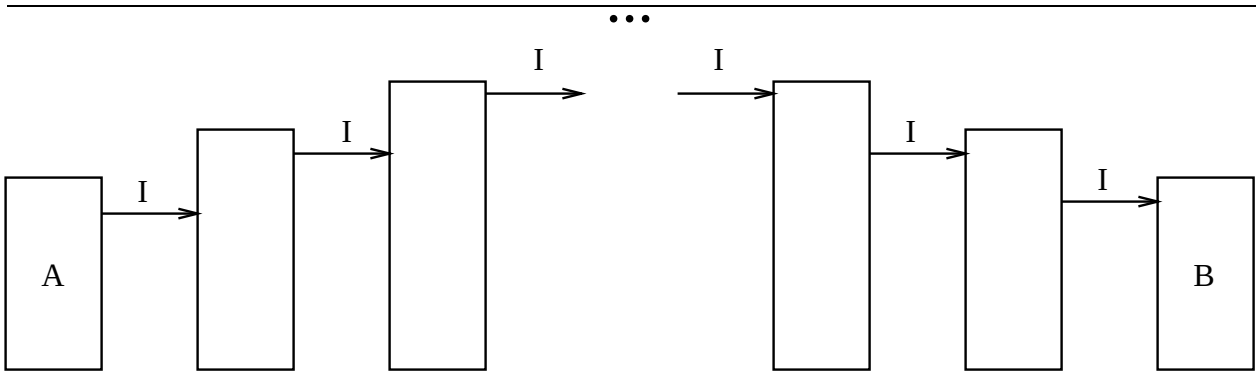


Figure 5: Placement of markers in IS-list to cover interval (A,B)

```
        if I contains x→key then add I to x→eqMarkers
        i := 0    // start at level 0 and go up
        while (I contains (x→key,x→forward[i]→key)) do
        begin
            // find level to put mark on
            while(i ≠ level(x)-1 and I contains (x→key,x→forward[i+1]→key) do
                i := i + 1
            // Mark current level i edge since it is the highest edge out of x that contains I.
            add I to x→markers[i]
            x := x→forward[i]
            // Add I to eqMarkers set on node unless currently
            // at right endpoint of I and I doesn't contain
            // right endpoint.
            if I contains x→key then add I to x→eqMarkers
        end

        // mark non-ascending path
        while(x→key ≠ I.right) do
        begin
            // find level to put mark on
            while(i ≠ 0 and I does not contain (x→key,x→forward[i]→key) do
                i := i - 1
            // At this point, we can assert that i=0 or I contains (x→key,x→forward[i]→key.
            // In addition, x is between A and B so i=0 implies I contains (x→key,x→forward[i]→key.
            // Hence, the interval must be marked.
            add I to x→markers[i]
            x := x→forward[i]
            if I contains x→key then add I to x→eqMarkers
        end
    end
```

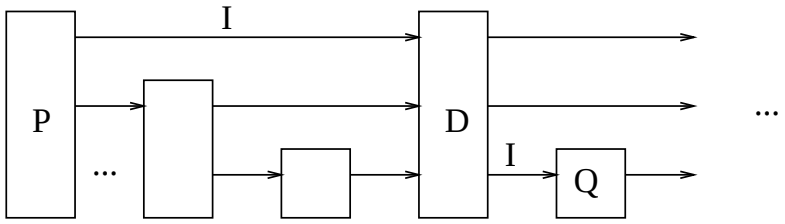
The procedure for deleting an interval from an IS-list discussed below is analogous to the insertion procedure.

3.5 Deletion

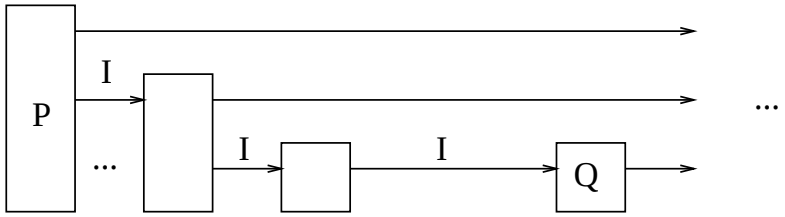
To delete an interval (A,B) the first step is to remove its markers. This is done by searching for the node containing A, and then scanning forward in the list for B, following a staircase pattern, which in general will contain an ascending path followed by a descending path. The approach used is very similar to that of the placeMarkers procedure for placing markers for a new interval, so we will not show a detailed algorithm for removing markers for (A,B).

The next step in deletion of (A,B) is to remove the IS-list nodes containing the endpoints A and B, and to adjust any markers affected so that the IS-list marker invariant is still satisfied. Affected markers will always either stay at the same level or move down. They will never move up. (If deletion of a node would make them go up, they would have already been placed higher, contradicting the IS-list marker invariant.) This forms the basis for an incremental algorithm for adjusting markers after deletion of a node that is similar to the one used after insertion of a node. The algorithm, which we call adjustMarkersOnDelete, is implemented by the procedure below. The parameters to the procedure are the IS-list L, the node to be deleted D, and a vector **updated** that contains pointers to the nodes with pointers into D that must be updated after the deletion. The **updated** vector can be constructed during a standard IS-list search for D.

```
procedure adjustMarkersOnDelete(L,D,updated)
demoted := φ
newDemoted := φ

// Phase 1: lower markers on edges to the left of D as needed.

for i := level(D)-1 down to 0 do
begin
    Find marks on edge into D at level i to be demoted, (which means they don't cover
    the interval (updated[i]→key,D→forward[i]→key)),
    remove them from that edge, and place them in newDemoted.

    // Note: no marker will ever be removed from a level 0 edge
    // because any interval with a marker on the incoming level 0
    // edge must have a marker on an edge out of D. Hence the
    // interval for any mark into D on level 0 always contains
    // (updated[0]→key,D→forward[0]→key).


    for each mark m in demoted set do
    begin     // the steps below won't execute for i=level(D)-1 because demoted is empty.
        Let X be the nearest node prior to D that has more than i levels.
        Let Y be the nearest node prior to D that has i or more levels
        (Y is updated[i], X is updated[i+1]).
        Place m on each level i edge between X and Y (this may not
        include any edges if X and Y are the same node).
        If this is the lowest level m needs to be placed on (i.e. m covers the interval
        (Y→key,D→forward[i]→key)) then place m on the level i edge out of Y and
        remove m from the demoted set.
    end
    demoted := demoted ∪ newDemoted
    newDemoted := φ
end

// Phase 2: lower markers on edges to the right of D as needed.
```



Markers for I=(P,Q) before deletion of node D.



Markers for I=(P,Q) after marker demotion & removal of D.

Figure 6: Example of node deletion and marker demotion.

```
demoted := φ
newDemoted := φ

for i := level(D)-1 down to 0 do
begin
    for each marker m on the level i edge out of D do
        if the interval of m does not cover (updated[i],D→forward[i])
        then add m to newDemoted.

    for each marker m in demoted do
    begin
        Place m on each edge on the level i path from D→forward[i] to
        D→forward[i+1].
        If the interval of m contains (updated[i]→key,D→forward[i]→key)
        then remove m from demoted.
    end
    demoted := demoted ∪ newDemoted
    newDemoted := φ

end adjustMarkersOnDelete
```

An example showing the demotion of markers for an interval $I = (P, Q)$ that would be done for one possible IS-list after deletion of a node D appears in figure 6. The incremental marker adjustment algorithms discussed above are important to the overall performance of insertion and deletion operations, which is analyzed in the next section.

4 Performance Analysis

The expected time to search an IS-list to find all intervals that overlap a key K is $O(\log n)$, since $O(1)$ time is spent per level and there are $O(\log n)$ levels in the list. The cost of insertion and deletion are determined

below.

Components of the insertion cost include the time required to

- insert the left and right endpoints of the interval,
- adjust markers for intervals already in the IS-list, and
- place markers for the new interval.

At the end of the operation the IS-list marker invariant must again hold. Inserting the left and right end points into the skip list requires $O(\log n)$ time. We assume that the interval markers on the edges are stored in a data structure that allows $O(\log n)$ inserts and deletes. This assumption implies that the time required to place markers for the new interval is $O(\log^2 n)$, because there are $O(\log n)$ levels and markers are placed on $O(1)$ edges of a level, at a cost of $O(\log n)$ per edge. The remaining cost that must be calculated is that of promoting markers due to the insertion of the endpoints of the new interval. We say that a node *disturbs* an interval if the node cuts an edge that contains a marker for the the interval. For our analysis, we define the following:

$P(i)$: probability that the inserted node has i levels,

$D(i,o)$: set of intervals disturbed (with markers on levels 1 through i) by operation o which inserts an i level endpoint,

$R(i,o,s)$: cost to promote the markers for interval s when operation o inserts a level i endpoint,

$A(i,o)$: cost to adjust the markers for operation o which inserts a level i endpoint.

In addition, using $E[v]$ to indicate the expected value of v, we define $D(i) = \mathrm{E}[|D(i,o)|]$, $A(i) = \mathrm{E}[A(i,o)]$, and $A = \mathrm{E}[A(i)]$.

Theorem 1 *If $D(i) = O(p^{-i})$, then the expected time required to adjust the existing markers in an IS-list when an endpoint is inserted is $O(\log^2(n))$.*

Proof: The value that we are trying to calculate is A. If we know the expected adjustment cost of an operation that inserts a level i endpoint, $A(i)$, we can calculate A by taking expectations. If the underlying skip list is parameterized by p, then the probability that a node has i levels is:

$$P(i) = (1-p)p^{i-1} \tag{1}$$

Therefore,

$$A = \sum_{i=1}^{\infty} (1-p)p^{i-1}A(i) \tag{2}$$

In order to find $A(i)$, we start by examining the cost to adjust the markers for the intervals that operation o disturbs when it inserts a level i endpoint. Operation o disturbs intervals in $D(i,o)$, and each of these intervals requires $R(i,o,s)$ time to adjust its markers. Therefore

$$A(i,o) = \sum_{s \in D(i,o)} R(i,o,s) \tag{3}$$

We consider the algorithm for updating the markers of the disturbed intervals. If no marker for that interval is promoted to a higher level, then processing that interval requires $O(1)$ time. If a marker for the interval is promoted from level j to level k, then all markers for that interval between the new node and the previous (next) k-level node must be deleted, (see figure 7). There are $O(1)$ consecutive s-level nodes between $(s+1)$-level nodes, so promoting a marker by l levels requires that $O(l)$ markers be deleted, which requires $O(l \log n)$ time.

We consider an interval that is disturbed when the new endpoint is inserted. The number of nodes that the level j markers for the interval cover is $O((1/p)^j)$. If an i-level node is inserted and disturbs the interval,

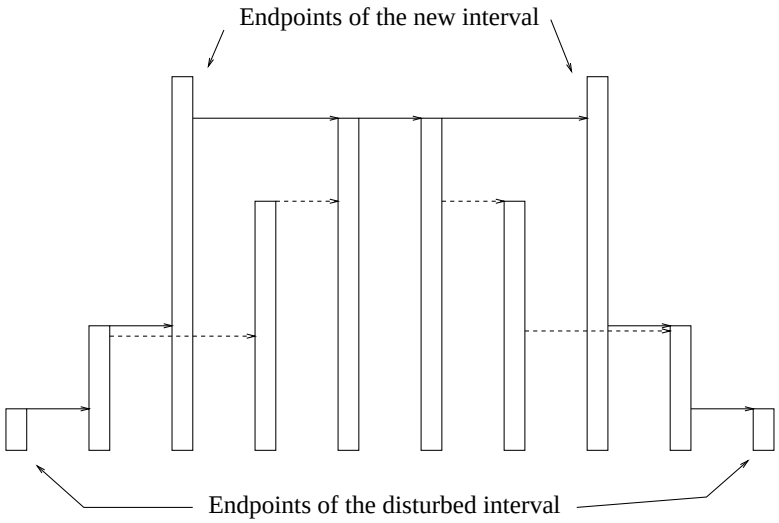


Figure 7: The dashed lines represent edges from which markers for the old interval are removed when the new interval is inserted.

then the marker for the interval is on a level $i-l$ edge with probability $O(p^l)$. We assume that if a marker is promoted when an i-level node is inserted, it is promoted to level i. Therefore, a marker for a disturbed interval is promoted an expected $\sum_{l=1}^{i-1} l \cdot O(p^l) = O(1)$ levels.

The expected amount of work to adjust markers for a disturbed interval when an i-level endpoint is inserted is therefore

$$\begin{aligned} E[R(i,o,s)] = &\quad O(1) + O(1) \cdot O(\log n) \\ = &\quad O(\log n) \end{aligned} \tag{4}$$

Therefore,

$$\begin{aligned} A(i) = &\quad E[A(i,o)] \\ = &\quad E[\textstyle\sum_{s \in D(i,o)} O(\log(n))] \\ = &\quad O(\log(n) \cdot E[|D(i,o)|]) \\ = &\quad O(\log(n) \cdot D(i)) \\ = &\quad O(min(\log(n)p^{-i}, \log(n) \cdot n)) \end{aligned} \tag{5}$$

In the last step, we use our assumption that $D(i) = O(p^{-i})$, and the fact that no more than n intervals can be disturbed. Putting equation (5) into equation (2), we get

$$\begin{aligned} A = &\quad \textstyle\sum_{i=1}^{\infty} P(i)A(i) \\ = &\quad \textstyle\sum_{i=1}^{\infty} p^i \cdot min(\log(n)p^{-i}, \log(n) \cdot n) \\ = &\quad O(\textstyle\sum_{i=1}^{\lceil \log n \rceil} \log(n) + \log(n) \sum_{i=\lceil \log n \rceil + 1}^{\infty} p^i \cdot n) \\ = &\quad O(\log^2 n + \log(n) \textstyle\sum_{i=\lceil \log n \rceil + 1}^{\infty} p^i p^{-\log n}) \end{aligned}$$

To finish, we make the substitution $r = i - \lceil \log n \rceil$,

$$\begin{aligned} A = &\quad O(\log^2 n + c \log(n) \textstyle\sum_{r=1}^{\infty} p^r) \\ = &\quad O(\log^2 n) \end{aligned} \tag{6}$$

The last step follows because the $p < 1$, so the sum is $O(1)$. •

Corollary 1 *If $D(i) = O(p^{-i})$, then the expected time to perform an insert is $O(\log^2 n)$.*

Proof: The time to to insert both endpoints is $O(\log^2 n)$. The remaining work is to add the markers for the new interval, which is $O(\log^2 n)$ •

Corollary 2 *If $D(i) = O(p^{-i})$, then the expected time to perform a delete is $O(\log^2 n)$.*

Proof: the analysis is the same as for the insert operation.

Our analysis of the IS-list assumes that an operation that inserts a level i node disturbs $O(p^{-i})$ intervals. We feel that most interesting distributions that we will encounter will satisfy this assumption.

We consider the following arguments: An interval in the IS-list follows a staircase pattern, and so places at most $O(1)$ markers on every level. Therefore, there are at most $O(n)$ markers placed on every level. The probability that a node has i or more pointers is $\sum_{j \geq i} P(j) = \sum_{j \geq i} (1-p)p^{j-1} = p^{i-1}$, so the expected number of forward level i edges is np^{i-1}. If every level i forward edge is equally likely to be cut when a node with at least i levels is inserted, then $D(i) = O(p^{-i})$.

The assumption that the expected number of level i forward edges is np^{i-1} is safe, because the skip list algorithm explicitly randomizes the node levels. The assumption that we are making about the underlying distribution is that every level j edge is equally likely to span the next insertion of a level i node, $j \leq i$. We next show that a large class of distributions are actually biased towards choosing edges with few markers on them.

Theorem 2 *If the endpoints of the distribution are chosen independently and identically distributed (iid) from a continuous distribution, then $D(i) = O(p^{-i})$.*

Proof: We consider, without loss of generality, that the endpoints are chosen iid from the uniform random distribution on $[0,1]$ (other continuous distributions can be mapped to a uniform $[0,1]$ distribution). Let us count $M(w)$, the expected number of markers placed on a level i edge of length w (the distance between the endpoints is w). We will call this edge e_w, and call the level $i+1$ edge that covers w, e_s, and we will say that e_s is of length s. A marker for the interval (a,b) is placed on e_w if and only if the endpoints of e_w are contained within (a,b), but the endpoints of e_s are not.

Let us first determine the probability that a marker would be placed on e_w if e_s does not exist. We consider a randomly chosen interval, (a,b), and the probability, M_w, that its marker is placed on e_w. The endpoints of the interval are uniformly randomly chosen, so that the joint distribution of (a,b) has the distribution of a two element order statistic. The theory of order statistics [Fel70] tells us that the density of the joint distribution $g(a,b)$ is a constant 2 in the region $b \in [0,1]$, $a \in [0,b]$. Let us define w_1 and w_2 to be the lower and higher endpoints of e_w. We can also show that the density function for the w_1, $h(w_1)$ is a constant $1/(1-w)$ in the region $[0, 1-w]$. Let $f(a,b,w_1)$ be the joint density of a, b, and w_1. Since w_1 is chosen independently of (a,b), $f(a,b,w_1) = g(a,b)h(w_1)$. A marker for (a,b) is placed on e_w iff. $a \leq w_1$ and $b \geq w_2 = w_1 + w$, so that:

$$\begin{aligned} M_w &= \int_{w_1=0}^{1-w} \int_{a=0}^{w_1} \int_{b=w_1+w}^{1} f(w_1, a, b)\, db\, da\, dw_1 \\ &= \int_{w_1=0}^{1-w} \int_{a=0}^{w_1} \int_{b=w_1+w}^{1} 2/(1-w)\, db\, da\, dw_1 \\ &= (1-w)^2/3 \end{aligned} \tag{7}$$

We see that longer edges (in terms of the difference in the endpoint keys) are less likely be spanned by a random interval, and so are less likely to carry a marker for that interval (see figure 8). Similarly, the number of intervals that cover e_s and so are not placed on e_w is $(1-s)^2/3$. Let $M_{w,s}$ be the probability that a marker is placed on an edge of length w whose parent edge is of length s. Then:

$$M_{w,s} = [(1-w)^2 - (1-s)^2]/3 \tag{8}$$

The lengths of e_w and e_s are not independent. However, since endpoints and node levels are chosen iid, the length of e_s is w plus an increment, a_i, that depends only on the level, i. So, the expected number of

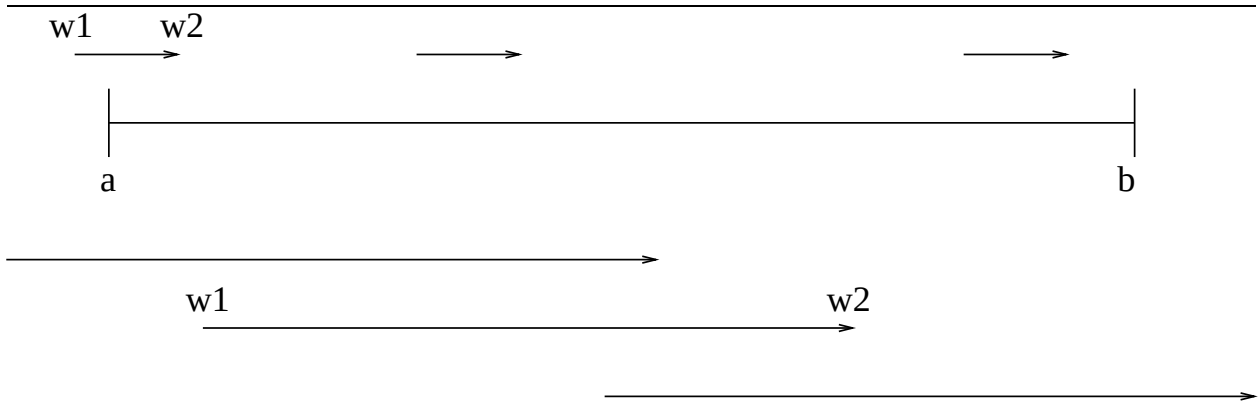


Figure 8: A long edge is less likely to fit in an interval (a, b) than is a short edge.

markers placed on e_w is n times the probability that an interval's marker is placed on the edge:

$$\begin{aligned} M(w) &= nM_{w,w+a_i} && (9)\\ &= n[(1-w)^2 - (1-w-a_i)^2]/3 && (10)\\ &= na_i(2-2w-a_i)/3 && (11) \end{aligned}$$

When a new endpoint is chosen, the probability that it is covered by the level i edge e is proportional to the length of e. But the expected number of markers on edge e decreases with the length of e. Therefore, when an i-level endpoint is inserted, the level i edge that spans the endpoint is likely to have fewer than average markers on it, so that $D(i) = O(p^{-i})$ •

This leads to

Theorem 3 *If the endpoints of the intervals are chosen iid from a continuous distribution, then the time required to perform an insert or a delete is* $O(\log^2 n)$.

In summary, the IS-list allows stabbing queries to be done in $O(\log n)$ time, and updates in $O(\log^2 n)$ time, while using $O(n \log n)$ storage.

5 Conclusion

The interval skip list is an efficient and relatively simple *dynamic* data structure for indexing intervals to handle stabbing queries efficiently. Unlike the commonly used segment tree, it can process insertions and deletions efficiently on-line, requiring $O(\log^2 n)$ time for each operation. We have implemented IS-lists in about 700 lines of C++ code, which is about one-fourth the amount of C++ code required in our implementation of interval binary search trees. No other known interval index that is based on a self-balancing data structure and supports both stabbing queries and dynamic updates can match the simplicity of implementation of the IS-list. This simplicity is in large part inherited from the skip list used as the basis for the IS-list. The main drawback of IS-lists is their potentially large storage utilization of $O(n \log n)$. If interval overlap is very low, less space is required. In fact, if intervals do not overlap, only $O(n)$ storage is needed. Thus, the IS-list may have advantages for applications that have intervals with limited overlap. The IS-list's storage cost may be well worth paying for applications that require a simple, efficient interval index that can be updated dynamically.

References

[AVL62] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, 3, 1962.

[Ede83a] H. Edelsbrunner. A new approach to rectangle intersections: Part I. *International Journal of Computer Mathematics*, 13(3-4):209–219, 1983.

[Ede83b] H. Edelsbrunner. A new approach to rectangle intersections: Part II. *International Journal of Computer Mathematics*, 13(3-4):221–229, 1983.

[Fel70] W. Feller. *An Introduction to Probability Theory and Its Applications, Vol. II.* John Wiley, 1970.

[GMW83] Gaston H. Gonnet, J. Ian Munro, and Derick Wood. Direct dynamic strucutures for some line segment problems. *Computer Vision, Graphics, and Image Processing*, 23, 1983.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, June 1984.

[HC90] Eric N. Hanson and Moez Chaabouni. The IBS tree: A data structure for finding all intervals that overlap a point. Technical Report WSU-CS-90-11, Wright State University, April 1990.

[McC85] Edward M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–278, 1985.

[Pug89] William Pugh. A skip list cookbook. Technical Report CS-TR-2286, Dept. of Computer Science, Univ. of Maryland, July 1989.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), June 1990.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addision Wesley, 1990.

[Wir86] Nicklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1986.