Splitting Interval Table

Zachary Kieda (zkieda@andrew.cmu.edu)

June 1, 2015

Abstract

We present a data structure that can map a real interval to an object. The splitting interval table (SIT) builds intervals by splitting an interval in two, and has amortized constant time lookup and expected $\log n$ insertion. This is a better runtime than BST interval structures, which have $\log n$ look up and insertion time. The worst case for an insertion in our structure is $O(n)$, and the best case is constant time. This data structure improves with certain inputs. For example, we have constant time expected insertion time when our inputs are uniformly randomly distributed.

Keywords. Data Structures

1 Proposed Solution

We have an intial interval is I, which goes from I_{lo} to I_{hi}, and has a length $length(I) = I_{hi} - I_{lo}$. We insert L, a sequence of splits, into I to create our splitting interval table. We assume I is fixed though all operations. When we insert a split into our data structure, the interval that we split becomes the left hand interval, and we create a new interval for the right hand interval.

The SIT data structure is very similair to the concepts employed to perform bucket sort. Bucket sort can achieve better expected runtime complexity by hashing each value into a bucket. Here, instead of sorting individual values, we store the intervals.

We keep a hash table H to store our intervals, and just perform a good hashing that will provide us with a good run time. We store k buckets that are uniformly spaced, and cover the range in I. Bucket i covers the range

$[I_{lo} + i(length(i)/k), I_{lo} + (i+1)(length(i)/k))$ for all $0 \leq i < k$. The only exception is the last bucket at $i = k - 1$, which stores the range inclusively. Note that $length(I)/k$ is the length of all buckets. Inside of each bucket, we store a BST that contains all of the intervals that are a part of this bucket. When we look up point p, we find the corresponding bucket, then traverse the BST to find its respective containing interval. After the number of intervals exceeds $2k$, we refactor the data structure.

Linked List. We store a singly linked list at each interval that points to its next largest interval (with respect to its minimum coordinate interval). This will allow us to traverse the sequence of intervals easily to perform operations.

To modify the linked list on insertion: We insert a split at position k in the interval I. I, by our invariant, has a node N linking to the next interval N' in our sequence. We split I in two, the left interval holds our old node. We create a new node N_0 for the right interval, such that the next item is N'. Then, we set the next node of N to N_0.

Hashing. We hash such that any point in a range corresponds to a single bucket. Suppose that we have a split at point $p \in I$. So, finding the correct one simply corresponds to $\lfloor p/(length(I)/k) \rfloor$. If $\lfloor p/((lengthI)/k) \rfloor = k$, we return $k - 1$.

Insertion. We insert L randomly into our data structure. We want to insert and keep our invariants in H. When we want to insert split p into our SIT, we first find p's bucket in H. Then, we find its corresponding interval in the bucket's BST. We modify this bucket to contain the two new intervals that are created. Then, we update the neighbor buckets to the right till we reach a new interval. This restores our invariant, the buckets to the left still point to the old interval, and the buckets to the right that are contained in this interval then point to the new node. We find the neighbors to the right using the linked list explained before.

Claim: The expected number of items in each bucket is constant.
Proof: When we have k buckets, we have $k \leq n < 2k$ items. In expectation, there are n/k items per bucket, which is in the range $1 \leq n/k < 2$. So, when we update an individual bucket by an insertion, we have an expected constant time to find the correct bucket.

Claim: Insertion is in expected $O(\log n)$.
Proof: Note that finding the correct bucket is expected constant time. Updating each neighbor to the right individually is also in constant time. So, the time to update all the right hand side neighbors after an insertion is our limiting factor. Since we inserted all of our elements in L into our data structure randomly, this improves our cost bound.

In the worst case, we have n items and n buckets. All n items are in the first bucket, then we make another $n-1$ insertions to the right of all the items in the first bucket. In the worst case ording, we insert all of the new items from left to right. We update all of the $n-1$ buckets to the right n times. This is in $O(n^2)$, and $O(n)$ per insertion. In the best case, we make insertions right to left. The first insertion will update all n right neighbors, but all of the next insertions will only update the current bucket.

We use backwards analysis to find the expected cost. Suppose we remove an item when there are $n < n+i < 2n$ items on our interval. The expected number of neighbors we have to update is in $O(n/(n+i))$. This gives us the expected runtime over n insertions

$$\sum_{i=1}^{n-1} \frac{n}{n+i} \leq cnH_{n-1} \in O(n \log n)$$

Which gives us $O(\log n)$ per insertion, our target bound.
Refactoring. When we refactor, we take all ordered n intervals by performing a range query over all of I. This query is explained in *Section 1.3: Extensions*. Using this list of intervals, we insert them one-by-one into a new H with n buckets from right to left.

Claim: Refactoring is in $O(n)$.
Proof: Note that we insert from right to left. This implies that we never modify a neighbor bucket more than once, since we only modify neighbor buckets that are to the right. This means all bucket modifications are in $O(n)$, and each basic insert is expected $O(1)$. We conclude that the refactoring is in $O(n)$.

Note that we refactor when $n = 2k$, and when this occurs we have $k := n$ buckets. The cost of refactoring is amortized $O(1)$ over each insertion. We use the accounting method. Allocate a token for refactoring for every insertion. Suppose we have zero tokens for for refactoring immediately after we

have just refactored. We have $k = n$. We insert n more splits, which gives us n tokens. This will pay for our refactoring, which is $O(n)$, and we're back to zero tokens.

1.1 On Uniform Inputs

Suppose the list of inputs we provide are uniformly randomy distributed through our interval. We will show that in this case we have expected constant time insertion.

With n intervals, the minimum ratio of items to buckets is $n/n \in O(1)$. Assume that the intervals are already uniformly randomly distributed, and we insert n more uniformly randomly distributed intervals into our structure. The ratio of items to buckets on insertion i is $1 \leq (n+i)/n \leq 2$. Note that the only part of an insertion that performed above the constant runtime was when we changed the neighbor buckets. However, since the intervals are uniformly distributed, the expected number of intervals already inside of our bucket is between 1 and 2. Implying, in expectation, we will only have to change the current bucket.

Formally, the expected cost of these n insertions is

$$\sum_{i=0}^{n} \mathrm{E}[\textit{nearby buckets}] = \sum_{i=0}^{n} \frac{n+i}{n}$$

$$= \frac{3(n+1)}{2}$$

Which gives us $O(1)$ per insertion.

1.2 On Other Inputs

The reason this paper was written was to aide in my other paper on the Multi Spectator Problem. The insertions we perform from handling the multi spectator problem have a unique property that also allows us to achieve constant time insertions.

Because of the order we insert each interval, we only insert the nearest spectators first, and that the initial interval maps to the background (nothing in sight). Note that the largest interval is either (1) the background or (2) another spectator close to the viewer. Since we insert spectators from nearest

to furthest, we will never overwrite an interval that already contains a spectator. This implies that a spectator in front will have a larger range (span more buckets), and will never be overwritten. This implies that we can leave background space alone when a specator is inserted – there's a limit to the number of buckets in the background space we need to update when we make an insertion - only the buckets that this spectator covers. This means we only need to update a neighbor bucket at most once over all insertions. This bounds our n insertions to a total $O(n)$ runtime for updating neighbor buckets. This makes the cost of insertion amortized $O(1)$.

1.3 Extensions

Suppose we wanted to perform a range query (getting all intervals in some search interval). The current structure will only search using a point as a key, and will find its containing interval. We hope to support other operations as wells.

We extend this data structure to give range queries $O(r)$ time, where r is the number of items that are in our search interval. We use our linked list to find the items in the interval $[i, j]$. When performing the range query, we only need to find i in our interval table. Then, we can traverse the linked list till we reach an interval's right hand side that exceeds j.