Advanced Data Structures Jan-Apr 2012

Lecture 4 — January 27, 2012

Lecturer: Venkatesh Raman *Scribe: Shion Samadder Chaudhury*

1 Overview

In the last lecture we saw Splay trees and the different bounds proved on them like the working set bound, the static finger bound, the sequential access bound and the dynamic finger bound. These bounds show that Splay trees execute certain classes of access sequences in $o(m\lg n)$ time, but they all provide $O(m\lg n)$ upper bounds on access sequences that sometimes take $\Theta(m)$ time to execute on Splay trees. Splay trees are conjectured to be $O(1)$-competitive for all access sequences.

In this lecture we discuss Tango tree, an online BST data structure that is $O(\lg\lg n)$-competitive against the optimal offline BST data structure on every access sequence. This reduces the competitive gap from the previously known $O(\lg n)$ to $O(\lg\lg n)$. Tango Tree originates in a paper by Demaine, Harmon, Lacono and Patrascu [1]. We also discuss a variation of the lower bound of Wilber [2], called the *interleave bound* (*IB*) on which today's results are based.

THEOREM (Wilber '89): $COST_{OPT}(\sigma) = \Omega(IB(\sigma))$

Today we will prove the following theorems:-

THEOREM 1. *$IB(\sigma)/2 - n$ is a lower bound on $COST_{OPT}(\sigma)$, the cost of the optimal offline BST that serves access sequence σ.*

Using *theorem 1* we shall prove the following main result on Tango trees:

THEOREM 2 *The running time of the Tango BST on a sequence σ of m accesses over the universe* {1,2,...,n} *is* $O((COST_{OPT}(\sigma) + n)(1 + lglgn))$.

(In view of *Wilber's theorem and theorem 1* it will be enough to show that: *The running time of the Tango BST on a sequence is* $O((IB(\sigma) + n)(1 + lglgn))$.)

We shall prove *theorem 1* later.

2 Interleave Bound

In this lecture, we assume for simplicity that the request sequence consists only of *FIND*'s.(Although it is easy to extend the result when we allow *INSERT* and *DELETE*). We also assume that the set of keys is {1,2,...,n}. We maintain a perfect binary tree P on the set {1,2,...,n}. This tree has a fixed structure over time. For each node y of P, assume that y is contained in the left subtree of y in P. Let $\{x_1, x_2, ...\}$ be a sequence of *FIND* operations.

For each node y in (P) the *preferred child* of y, either *left* or *right*, is the one whose subtree contains the most recently accessed descendent of y. If the most recently accessed element is y, the preferred child is *left.*

For each node y in P, we label each access x_i in the access sequence σ by whether x_i is in the left or right subtree of y, discarding all accesses outside y's subtree in P.

The *amount of interleaving* through y, *IC(y)* is the number of child preferences that would change in this sequence.If no preferences change, we incur a cost of 1.

Definition :- The *interleave bound IB(σ)* is the sum of these interleaving amounts over all nodes y of P.

$$IB(\sigma) = \sum_{y \epsilon P} IC(y)$$

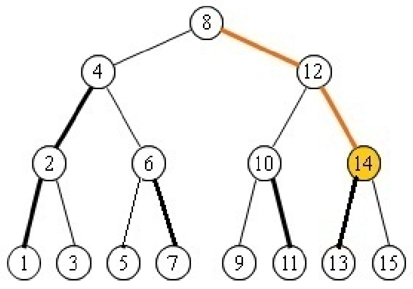

Figure 1: Access Element 14

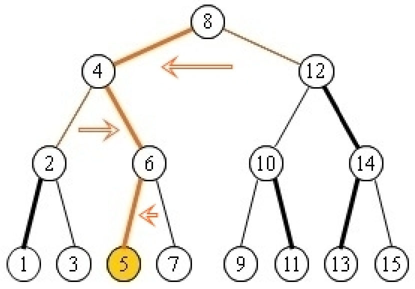

Figure 2: Access Element 5

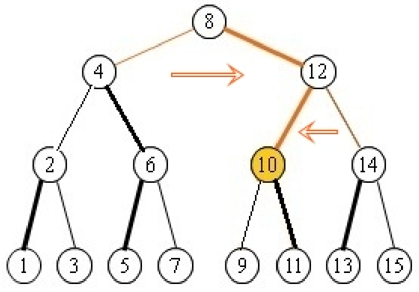

Figure 3: Access Element 10

Let us consider the above tree and the access sequence (....8, 14, 5, 10) (Figures 1, 2 and 3.) (the bold black lines denote the preferences prior to access 8. Note the change from old preferences to new preferences[black to yellow].)

3 Tango Tree

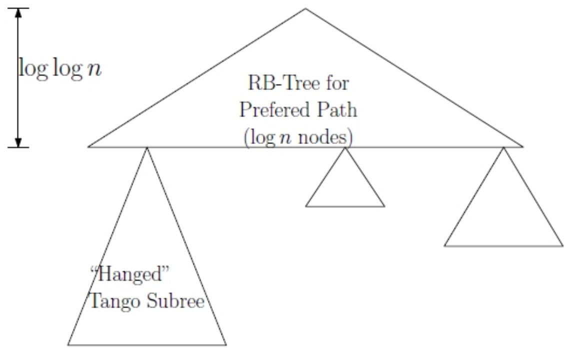

Figure 4: Tango Tree Structure

3.1 Tango Tree Structure

Now we specify the structure of *Tango Trees.* Let T_i be the state of the *Tango tree* after executing the first i accesses $x_1, x_2, ..., x_i$.We define T_i using the tree P, *preferred children* as defined before and *Preferred Paths*, which we will describe shortly. The following steps are taken:

1. Augment P to maintain for each internal node y of P, a *preferred child.* If no node whithin y's subtree has been accessed (or if y is a leaf), y has no *preferred child.*

2. Let P_i be the state of the augmented tree P after i accesses.(P_i is uniquely determined by $x_1, x_2, ..., x_i$ in dependent if the *Tango tree.*) Start at the root of P and repeatedly proceed to the *preferred child* of the current node until reching a node whithout a *preferred child.* The nodes traversed by this process, including a root form a *Preferred Path.*

3. Compress this *preferred path* in to an *auxiliary tree* (defined in the next section) R.

4. By removing the path from P, it has been split into several pieces. Recurse on each piece and "hang" the resulting BST's as child subtrees of the *auxiliary tree R.*

3.2 Auxiliary Tree

Definition:- The *auxiliary tree* data structure is an augmented BST that stores a subpath of a root-to-leaf path in P (in this case, a preferred path) *ordered by key value*, with each node it stores the fixed *depth* in P and supports each of the following operations in time $O(\lg k)$ (k is the total number of nodes involved in the operation) :-

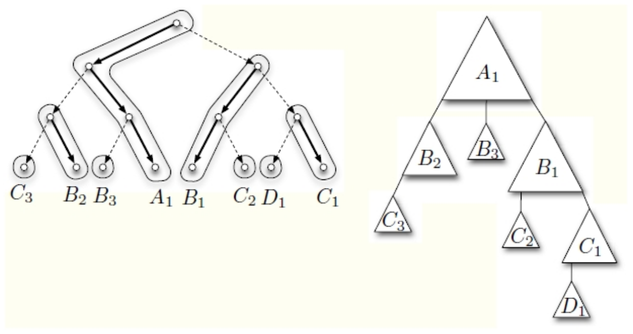

Figure 5: On the left, reference tree P with its preferred paths. On the right, Tango Tree representation of P.

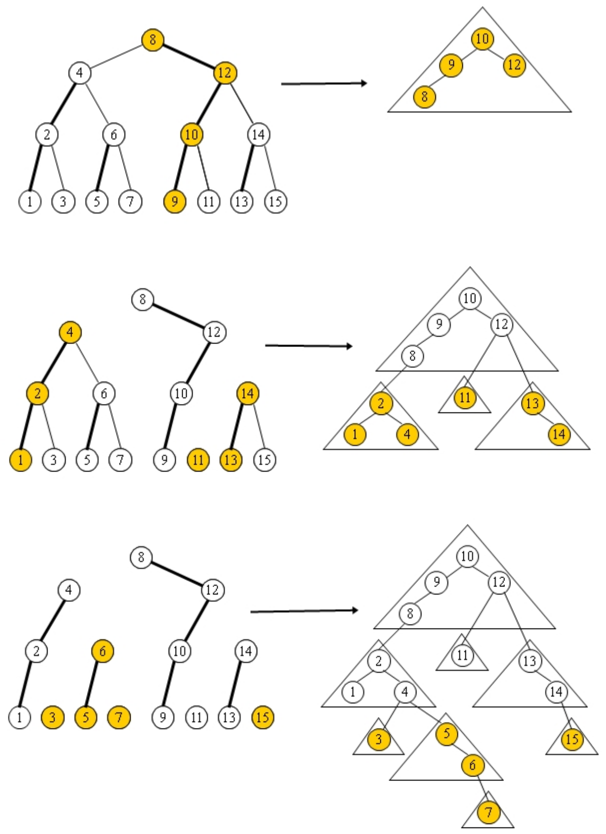
Figure 6: Step by step construction of a Tango Tree

1. Searching for an element by key.

2. Cutting it into two auxiliary trees, one storing the path of all nodes at most at a given depth d and the other storing the path of all nodes greater than d.

3. Joining two auxiliary trees that store two disjoint paths where the bottom of one path is the parent of the top of the other path.

We call the shallowest node the *top* and the deepest node the *bottom* of the path.

An auxiliary tree is implemented as an augmented *red-black tree*. In addition to storing the key value and the depth (*Dep*), each node stores the maximum depth (*MaxDep*) which is the maximum value of *Dep* among its children and similarly the minimum depth (*MinDep*).Maintaining these auxiliary values do not effect the complexity of *red-black tree* operations (see [3], chapter 14]). We also know that in *red-black trees*, we can do the following *SPLIT* and *MERGE* operations in $O(\lg k)$ time where k is the number of nodes (see [3],Problem 13-2]).

Definition:- $\hat{\text{S}}$PLIT(T, x):- splits the *red-black tree* T at the node x into two *red-black trees* where one tree includes all the nodes that has $key < x$ and the other tree includes all the nodes that has $key > x$.

Definition:- $MERGE(T_1, T_2, x)$:- merges the two *red-black trees* T_1 and T_2 where T_1 has all its nodes with $key < x$ and T_2 has all its nodes with $key > x$, into a single *red-black tree* which contains all nodes in T_1 and T_2 and a node with $key = x$.

We observe that each *prefered path* involves a contiguous interval of depths (actually, it involves depths in the interval $[t, \lg n]$ where t is the minimum depth).Using the *SPLIT* and *MERGE* operations, we claim that given a depth$d$, we can cut the nodes whose $Dep > d$ in a *red-black tree* . Also, we can join two *red-black trees* where one only contains nodes with $Dep > d$, and we have performed a cut to the other tree so that its $Dep > d$ nodes are all lost.

The key observation here is in *red-black tree* of any path, the keys of nodes that have $Dep > d$ form an interval $[t, r]$. We can find the nodes with keys l and r following informations in *MinDep* and *MaxDep*.(we find l by starting at the root and repeatedly walking to the leftmost child whose subtree has $MaxDep > d$ and symmetrically we find r.) Then we find the predecessor l' of l and the successor r' of r. All of these operations take $O(lgk)$ time in *red-black trees*.

To do *cut*, we do a *SPLIT* at l' and then SPLIT at r'. Then we have a tree whose nodes have $l' < key < r'$ and therefore all the nodes with $Dep > d$. We mark this tree has "hanged" and then do MERGE at r' and l' respectively to finish cut operation. (The whole procedure is shown in Figure)

Join is similar to cut. Suppose A is the tree with nodes $Dep > d$, B is the tree that do not have nodes with $Dep > d$. Observe that the key values in A must fall in between two adjacent keys l' and r' in B, we can do *SPLIT* at this two points, and then do two *MERGE*s to join A and B.

3.3 Tango Algorithm for $FIND(\sigma)$

We construct the new state T_i from the given previous state T_{i-1} and the next access x_i. In T_{i-1} we walk toward x_i. Accessing x_i changes the preferred children to make a preferred path from the root to x_i and sets the preferred child of x_i to the left. Except for the last change to x_is preferred

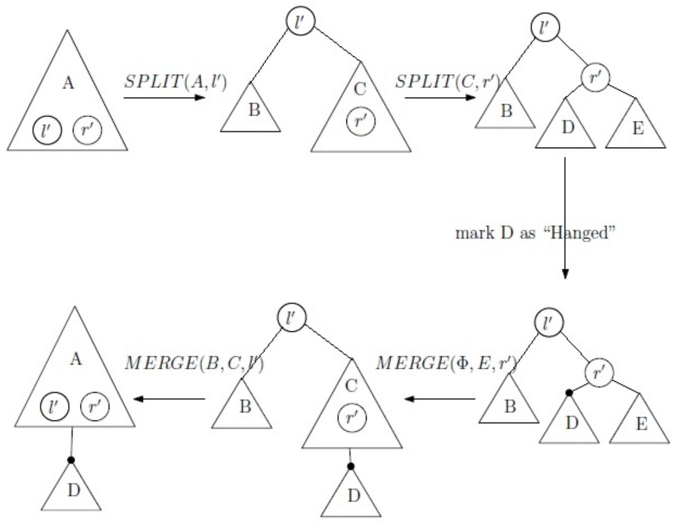

Figure 7: Implementing cut.

child, the points of change in preferred children correspond exactly to where the BST walk in T_{i-1} crosses from one augmented tree to the next. Thus, when the walk visits a marked node x, we cut the auxiliary tree containing the parent of x, cutting at a depth one less than the *MinDep* of nodes in the auxiliary tree at x; then we join the resulting top path with the augmented tree rooted at x. Finally, when we reach x_i, we cut its auxiliary tree at the depth of x_i and join the resulting top path with the auxiliary tree rooted at the preceding marked node of x_i.

3.4 Analysis and proof of Theorem 2

Lemma 3.1. *The running time of an access x_i is $O((k+1)(1+\lg\lg n))$, where k is the number of nodes whose preferred child changes during access x_i.*

Proof. The search visits a root-to-x_i path in T_{i-1}, which we partition into subpaths according to the auxiliary trees visited. Clearly the search path in T_{i-1} partitions into at most k+1 subpaths in k+1 auxiliary trees. Hence the cost of the search within a single auxiliary tree is $O(\lg\lg n)$ because each auxiliary tree stores $O(\lg n)$ elements, corresponding to a subpath of a root-to-leaf path in P. Therefore the total search cost for x_i is $O((k+1)(1+\lg\lg n))$. The update cost is the same as the search cost up to constant factors. For each of the at most $k + 1$ auxiliary trees visited by the search, we perform one cut and one join, each costing $O(\lg\lg n)$. We also pay $O(\lg\lg n)$ to find the preceding marked node of x_i. The total cost is thus $O((k+1)(1+\lg\lg n))$.

Definition:- Define the *interleave bound of access x_i* to be

$$IB_i(\sigma) = IB(x_1, x_2, ..., x_i) - IB(x_1, x_2, ..., x_{i-1})$$

.

Observation:- The number of nodes whose preferred child changes from left to rightor from right to left during an access x_i equals $IB_i(\sigma)$.This is because the preferred child of a node y in P changes from left to right when the previous access within ys subtree in P was in the left region of y and the next access x_i is in the right region of y and symmetrically for right to left. Both of these events correspond exactly to interleaves.

THEOREM 2: *The running time of the Tango BST on a sequence σ of m accesses over the universe* {1,2,...,n} *is* $O((COST_{OPT}(\sigma) + n)(1 + lglgn))$.

Proof. There can be at most n first preferred child settings (i.e., changes from no preferred child to a left or right preference). Therefore the total number of preferred child changes is at most $IB(\sigma)+n$. Combining this bound with Lemma 3.1, the previous observation and theorem 1.1 we get the result.

4 Proof of Interleave Bound (Theorem 1)

4.1 Idea of the proof and steps taken

1. Let T_i denote the state of a fixed arbitrary BST after the execution of accesses x_1, x_2,..., x_i.

2. **Definition**:- For a node y in P the *transition point* for y at time i is defined to be to be the *minimum-depth* node z in the BST T_i such that the path from z to the root of T_i includes a node from the left subtree of y and a node from the right subtree of y. (We ignore nodes not from ys subtree in P.)

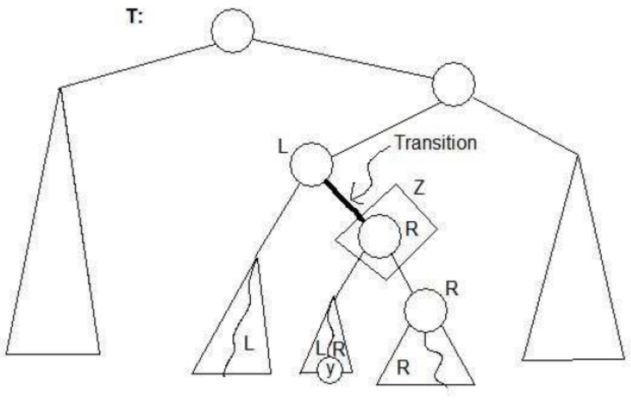

Figure 8: Transition.

3. Intuitively, any BST access algorithm applied both to an element in the left subtree of y and to an element in the right subtree of y must touch the transition point for y at least once.

4. Transition point is well-defined.

Lemma 4.1.: *For any node y in Pand any time i, there is a unique transition point for y at time i.*

5. Transition point is stable.

Lemma 4.2.: *If the BST access algorithm does not touch a node z in T_i for all i in the time interval [j, k], and z is the transition point for a node y at time j, then z remains the transition point for node y for the entire time interval [j, k].*

6. Transition points are different over all nodes in P.

Lemma 4.3.: *At any time i, no node in T_i is the transition point for multiple nodes in P.*

Using the above steps we will prove theorem 1.

Proof of Lemma 4.1.:Let l be the *lowest common ancestor* of all nodes in T_i that are in the left subtree of y. $l$is in the left region of y. (since we are considering BST here) Thus l is the unique node of minimum depth in T_i among all nodes in the left subtree of y. Symmetrically for r, *the lowest common ancestor* in right subtree Also, since the *lowest common ancestor* in T_i of all nodes in the subtree of y must be a unique node of minimum depth, it must be either l or r (one with smaller depth).

Let it be l(symmetrically). Then l is an ancestor of r. The path in T_i from the root to r visits at least one node (l) from the left subtree of y in P, and visits only one node (r) from the right subtree of y in P because it has minimum depth among such nodes. Also, any path in T_i from the root must visit l before any other node in the left or right subtree of y, because l is an ancestor of all such nodes, and similarly it must visit r before any other node in the right subtree of y. Hence r is the unique transition point for y in T_i.

Proof of Lemma 4.2.: Let l and r be as in the proof of the previous lemma. Let l be an ancestor of r in T_j. Then r is the transition point for y at time j. The BST access algorithm does not touch r. Hence it does not touch any node in the right subtree of y, and thus r remains the *lowest common ancestor* of these nodes.

The algorithm may touch nodes in the left subtree of y, and in particular the *lowest common ancestor* $l = l_i$ of these nodes may change with time (i). But still, l_i remains an ancestor of r. Since nodes in the left subtree of y cannot newly enter rs subtree in T_i, and y is initially outside this subtree, some node l_i' in the left subtree of y must remain outside this subtree in T_i.Hence, the *lowest common ancestor* a_i of l_i' and r cannot be r itself, so it must be in the left region of y. Thus l_i must be an ancestor of a_i, which is an ancestor of r, in T_i.

Proof of Lemma 4.3.: Consider any two nodes y_1 and y_2 in P, and define l_j and r_j in terms of y_j as in the proof of Lemma 4.1. We have two cases:-

1. y_1 and y_2 are not ancestrally related in P: then their left and right subtrees are disjoint from each other and thus, l_1 and r_1 are distinct from l_2 and r_2, and hence the transition points for y_1 and y_2 are distinct.

2. (Symmetrically) y_1 is an ancestor of y_2 in P: If the transition point for y_1 is not in y_2s subtree in P then it differs from l_2 and r_2 and thus the transition point for y_2. Otherwise, the transition point for y_1 is the lowest common ancestor of all nodes in y_2s subtree in P, and thus it is either l_2 or r_2, whichever is less deep. On the other hand, the transition point for y_2 is either l_2 or r_2, which ever is deeper.

Therefore the two transition points differ in all cases.

We will now prove theorem 1.

Proof of Theorem 1.: We count the number of transition points the (optimal offline) BST touches. We count the number of times the BST touches the transition point for y, separately for each y, and then sum these counts(by lemma 4.3). Let l and r be as in the proof of Lemma 4.1.(transition point for y is always either l or r, whichever is deeper). Let $x_{i_1}, x_{i_2}, ..., x_{i_p}$ be a *maximal ordered subsequence* of accesses to nodes that alternate between being in the left and right subtrees of *y*. So p is the *amount of interleaving through y*. Let (symmetrically)the odd accesses $x_{i_{2j-1}}$ be nodes in the left subtree of *y*, and the even accesses $x_{i_{2j}}$ be nodes in the right subtree of y. Consider each j with $1 \leq j \leq \lfloor p/2 \rfloor$. Any access to a node in the left subtree of y must touch *l,* and any access to a node in the right subtree of y must touch *r*. Thus, for both accesses $x_{i_{2j-1}}$ and $x_{i_{2j}}$ to avoid touching the transition point for *y*, the transition point must change from r to l in between, which requires touching the transition point for y(lemma 4.2). Thus the BST access algorithm must touch the transition point for y at least once during the time interval $[i_{2j-1}, i_{2j-1}]$. Summing over all j, the BST access algorithm must touch the transition point for y at least $\lfloor p/2 \rfloor$ times. Summing over all y, the amount p of interleaving through y adds up to the interleave bound $IB(\sigma)$; thus the number of transition points touched adds up to at least $IB(\sigma)/2$ - n.

5 References

[1] E. Demaine, D. Harmon, J. Iacono, and M. Patrascu. Dynamic optimality almost. *SICOMP*, 37(1):240251, 2007. Also in *Proc. FOCS 2004*.

[2] R. Wilber. Lower bounds for accessing binary search trees with rotations. *SICOMP*, 18(1):5667, 1989.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[4] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652686, 1985.