PRINCETON UNIV. F'08 COS 521: ADVANCED ALGORITHM DESIGN

Lecture 4: Competitive analysis of data structures

Lecturer: *Sanjeev Arora* Scribe:*Rong Ge*

1 Introduction

1.1 Competitive Ratio

In the last lecture we gave the definition of competitive ratio for data structures, and showed a simple Move-To-Front algorithm that achieves competitive ratio 2 in the linked list model. Recall that competitive ratio is defined as

DEFINITION 1 (COMPETITIVE RATIO) *An algorithm A has a competitive ratio of α if for all sequence of operations σ, we have*

$$COST_A(\sigma) \leq \alpha COST_{OPT}(\sigma) \tag{1}$$

where $COST_A(\sigma)$ is the cost of algorithm A on the sequence σ, and $COST_{OPT}(\sigma)$ is the cost of the best algorithm for the same sequence.

1.2 BST model

Today we are going to apply competitive analysis on binary search trees. In the binary search tree model (shown in figure 1), each node v has a key k, all nodes that are in the left subtree of v has keys smaller or equal to k, and nodes that are in the right subtree of v has keys larger than k. The cost of following pointers in the tree and performing rotations are all constants.

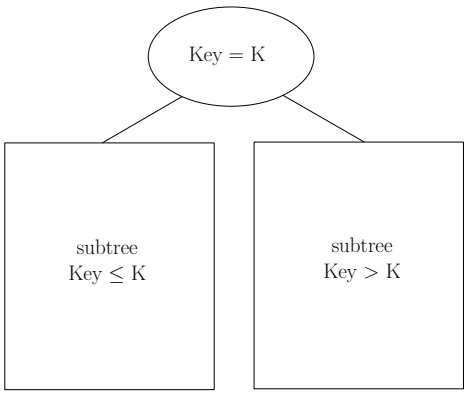

Figure 1: Binary Search Trees

The operations that binary search tree supports include $INSERT$, $DELETE$ and $FIND$.

Splay Tree [ST85] is conjectured to have competitive ratio $O(1)$, but the best bound known is only $O(\log n)$. Recently, [DHIP07] proposed a new data structure Tango Tree that has a competitive ratio $O(\log \log n)$.

1.3 Interleave Bound

One of the difficulties in proving competitive ratio upperbounds is that we do not know how the optimal algorithm works. To solve this problem, we first come up with some lowerbound $IL(\sigma)$ such that $\forall \sigma\ COST_{OPT}(\sigma) \geq IL(\sigma)$, then show that $COST_{TANGO}(\sigma) \leq O(\log \log n) \cdot IL(\sigma)$.

In this lecture, we assume for simplicity that the request sequence consists only of $FIND$'s. It is easy to extend the result when we allow $INSERT$ and $DELETE$. Now we may as well assume for notational ease that the set of keys is $\{1, 2, ..., n\}$ because the BST's structure only depends upon the comparison results between the keys. Let $\sigma = (\sigma_1, \sigma_2, \sigma_3, ...)$ be a sequence of $FIND$ operations.

We define P as the static complete binary search tree on the set $\{1, 2, ..., n\}$. For computing IL, we maintain a bit $MR[i]$ for each $i \in \{1, 2, ..., n\}$, where MR stands for "Most Recent".

$MR[i] = 1$ iff the most recent $FIND$ operation that was routed through node i in P went to its right subtree.

Since P is a complete binary search tree, for each single $FIND$ operation, there's at most $\lceil \log n \rceil$ changes in MR values.

Definition 2 (Interleave Bound) *IL(σ) = Total number of changes in MR bits while performing the $FIND$ operations in σ.*

Wilber [Wil89] showed the following algorithm that states IL is a lowerbound for $COST_{OPT}(\sigma)$.

Theorem 1

$$COST_{OPT}(\sigma) = \Omega(IL(\sigma)) \tag{2}$$

We will show the proof of this theorem later.

2 Tango Tree Description

2.1 Tango Tree Structure

Now we specify the structure of Tango Trees using the binary search tree P and MR bits. First we define the notion of Prefered Path

Definition 3 (Prefered Path) *Prefered Path (Figure 2) is the path descending from root to a leaf following the MR bits. (that is, going left when MR=0 and right when MR=1)*

It's easy to see that by removing the edges in Prefered Path from the tree P, we get a disjoint union of $\lceil \log n \rceil$ subtrees. Using these subtrees, we can define PP-Decomposition $PPD(T)$ for a tree T recursively

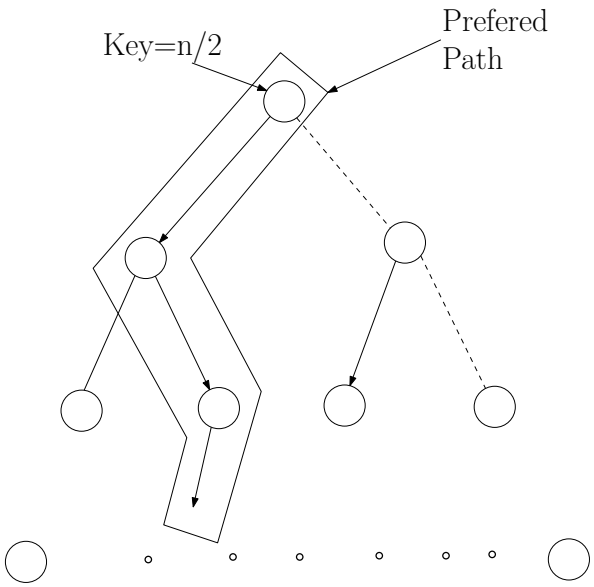

Figure 2: Prefered Path

Definition 4 (PP Decomposition)

$$PPD(T) = \{PreferedPath(T)\} \cup \textit{PP-Decomposition for trees in } \{T - PreferedPath(T)\} \tag{3}$$

Recall the fact that Red-Black Trees can support $INSERT$, $DELETE$, $FIND$ in $O(\log k)$ time, where k is the number of nodes in the tree. Using Red-Black Trees, we define Tango Tree recursively

Definition 5 (Tango Tree) *A Tango Tree (Figure 3) is a binary search tree in which we store the nodes in Prefered Path in a Red-Black Tree, and "Hang" Tango Trees for the* $\lceil \log n \rceil$ *subtrees in the appropriate places in this Red-Black Tree.*

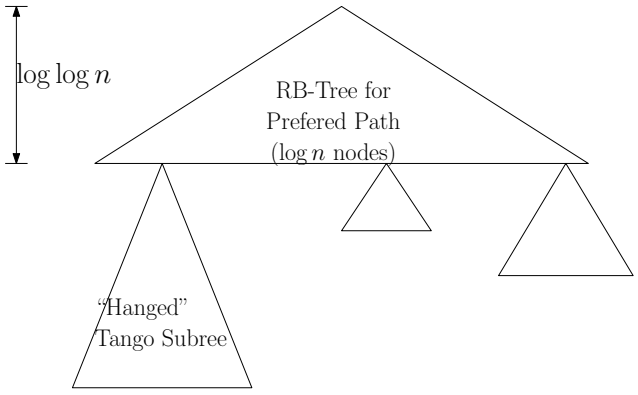

Figure 3: Tango Tree Structure

2.2 $FIND$ operations for Tango Tree

When a $FIND$ opperation happens, the MR bits may change. The changes in MR bits will cause PP-Decomposition to change. How do we update tango tree?

In fact, it is possible to do it in time $O(\log\log n)\cdot$ #changes in MR bits. To do this, we need to store auxiliary information Dep, $MinDep$ and $MaxDep$ in the nodes of Red-Black tree. The Dep value for a node is the depth of the node in the static tree P; the $MinDep$ value for a node is the minimum value of Dep among its children; the $MaxDep$ value is the maximum value of Dep among its children. Maintaining these auxiliary values do not effect the complexity of Red-Black Tree operations (see [CLRS01, chaptor 14]) We also claim that in Red-Black Trees, we can do the following SPLIT and MERGE operations in $O(\log k)$ time where k is the number of nodes (see [CLRS01, Problem 13-2]).

Definition 6 (SPLIT and MERGE) *$SPLIT(T, x)$, where T is a Red-Black Tree and x is a node in the tree, splits the tree into two Red-Black Trees where one tree includes all the nodes that has key $< x$; the other tree includes all the nodes that has key $> x$.*

$MERGE(T_1, T_2, x)$, where T_1 is a RB Tree whose nodes have key $< x$, T_2 is a RB Tree whose nodes have key $> x$, merge the two trees into a single RB Tree, which contains all nodes in T_1, T_2 and a node with key $= x$.

Observe that each prefered path in the PP-Decomposition involves a contiguous interval of depths (actually, it involves depths in the interval $[t, \log n]$ 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 $[l, r]$ (because they are the intersection of a subtree of P and the path). We can find the nodes with keys l and r following informations in $MinDep$ and $MaxDep$. Then we find the predecessor l' of l and the successor r' of r. All of these operations takes $O(\log k)$ time in Red-Black Tree.

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 is 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 4)

Join is similar with 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 falls 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.

Now for $FIND(\sigma)$, assume there are m changes in MR bits, we can break the path from root of P to σ in $m+1$ segments, each of which lies in the same path in the previous PP-Decomposition. Within each sigment, we can find the depth d that we should do cut and join in $O(\log\log n)$ time (just find adjacent l and r so that $l < \sigma < r$ and then d should be the larger value between $Dep[l]$ and $Dep[r]$), we perform the cut and then join with the Tango Tree previously "hanged" there (these steps also take $O(\log\log n)$ time). In this way we can find σ and change the structure of Tango Tree according to changes in MR bits in $O(\log\log n)\cdot(m+1)$ time, which infers

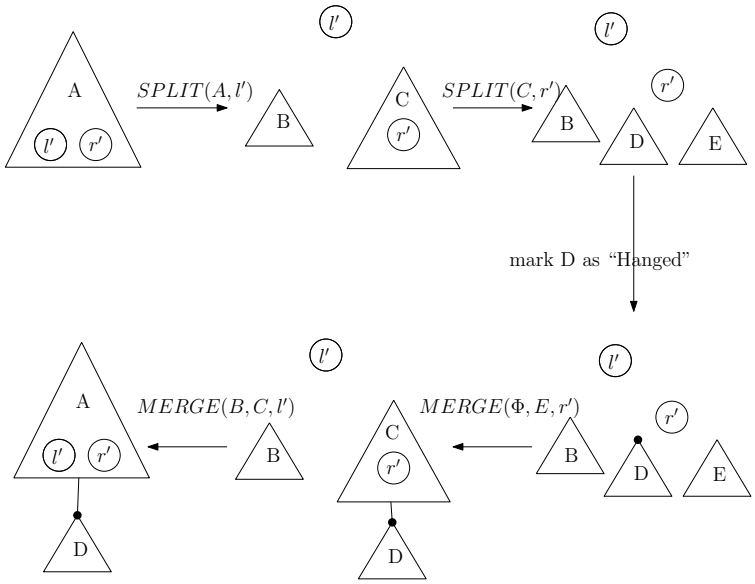

Figure 4: Implementing cut operation

THEOREM 2

$$COST_{TANGO}(\sigma) \leq O(\log\log n) \cdot IL(\sigma) \tag{4}$$

3 Proof of IL lowerbound

3.1 Prefix Sum Problem

To prove that IL is a lowerbound for the cost of maintaining binary search tree, we reduce the cost of maintaining binary search tree for operations σ to the cost of solving a certain instance σ' of Prefix Sum Problem.

We would like to show $COST_{OPT'}(\sigma') = O(COST_{OPT}(\sigma))$ and $COST_{OPT'}(\sigma') = \Omega(IL(\sigma))$ where σ' is an instance of Prefix Sum Problem created according to σ, and OPT' is the optimal algorithm for solving that problem.

New we define Prefix Sum problem

DEFINITION 7 (PREFIX SUM PROBLEM) *Maintain n registers $A[1..n]$ that support opperations $UPDATE$ and $PREFIXSUM$.*

$UPDATE(i, \Delta)$ updates $A[i] = \Delta$

$PREFIXSUM(i)$ returns the current value of $\Sigma_{1 \leq j \leq i} A[j]$.

To relate binary search tree problem with prefix sum, we construct σ' as follows

$$\sigma' = (UPDATE(\sigma_1, \Delta_1), PREFIXSUM(\sigma_1), UPDATE(\sigma_2, \Delta_2), PREFIXSUM(\sigma_2), ...) \tag{5}$$

Using binary search trees with auxiliary information (store the sum of Δ values in subtree), we can solve Prefix Sum problem. This infers

LEMMA 3

$$COST_{OPT'}(\sigma') = O(COST_{OPT}(\sigma)) \tag{6}$$

Δ_i are just variables; it is not important to assign specific values to them. Note that we can perform additions in a semigroup, so we have to carefully describe the computational model so that the algorithm gain bennefit by using tricks with this auxiliary computation.

DEFINITION 8 (SEMIGROUP MODEL) *Semigroup Model allows an arbitrary number of auxiliary variables $REG[j]$ and two kinds of operations:*
$REG[j] \leftarrow \Delta_t$ and $REG[k] \leftarrow REG[i] + REG[j]$
Each operation cost 1 unit.

Since we have an arbitrary number of REG, we can assume that each register is written only once. Also, following the definition of the model, we can see that at each step, every REG value is a sum of past Δ_t's.

DEFINITION 9 (SIGNATURE) *If $REG = \Delta_{t_1} + \Delta_{t_2} + \cdots + \Delta_{t_l}$, then the signature of this register $Sig[Reg] = (x, y)$, where*
x is the largest index among $\sigma_{t_1}, \sigma_{t_2}, \cdots, \sigma_{t_l}$
y is the largest time among $t_1, t_2, \cdots, t_l$

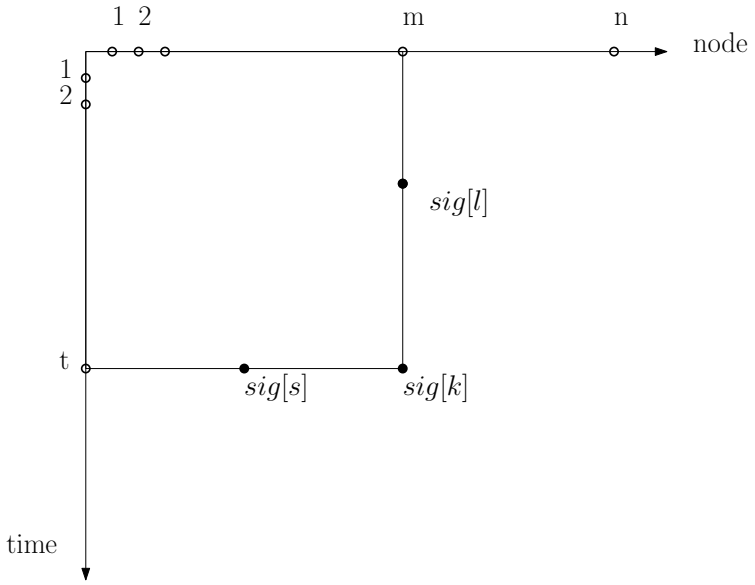

Figure 5: Query-Time Diagram and Signature

If $REG[k] \leftarrow REG[l] + REG[s]$, then $sig[l]$ and $sig[s]$ all have coodinates no larger than $sig[k]$. In the Query-Time Diagram (Figure 5), this means that $sig[s]$ and $sig[l]$ are all in the rectangle of $sig[k]$.

For an operation $REG[k] \leftarrow REG[l] + REG[s]$, we say that a node v in the static binary search tree P owns it if $sig[l]_1 \in [a, b]$ and $sig[s]_1 \in (b, c]$. Where $[a, b]$ is the range of keys of v's left subtree, and $(b, c]$ is the range of keys of v's right subtree. Notice that every such operation is owned by exactly one node in P (that is, the least common ancester of the two signature's first component). So we have

$$\#operations = \Sigma_v \#\text{operations owned by } v \tag{7}$$

So if we can prove the following lemma, we would be able to complete the proof.

Lemma 4
The number of operations owned by v is at least the number of times MR[v] changes from 0 to 1

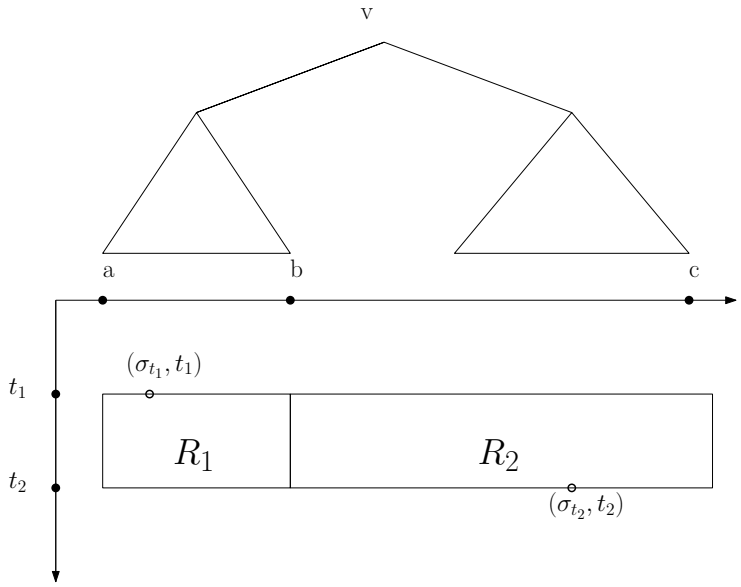

Figure 6: Query-Time Diagram for Proving Lemma 4

The proof is shown in Figure 6. t_2 is a time that $MR[v]$ changed from 0 to 1. t_1 is the last time when there is a query in $[a, b]$. We want to show that there exist an operation in time $[t_1, t_2]$ that is owned by v. Because of semigroup property, we observe that the answer to $PREFIXSUM(\sigma_{t_1})$ must come from a register whose signature is (σ_{t_1}, t_1); also the answer to $PREFIXSUM(\sigma_{t_2})$ must come from a register whose signature is (σ_{t_2}, t_2).

If there's no operation of the desired type during interval $[t_1, t_2]$, then there is no way that the answer of $PREFIXSUM(\sigma_{t_2})$ to be correct because the answer will be irrelevant to Δ_{t_1}.

In conclusion, we have proved $COST_{OPT'}(\sigma') = \Omega(IL(\sigma))$, together with Lemma 3, we proved Theorem 1.

References

[DHIP07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality—almost. *SIAM J. Comput.*, 37(1):240–251, 2007.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[Wil89] R. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM J. Comput.*, 18(1):56–67, 1989.