COMS 6998 Advanced Data Structures **Spring 2019**

Lecture 3: Feb 7

Lecturer: Omri Weinstein *Scribe: Justin Whitehouse, Trung Vu*

Overview

(1) Geometric characterization of the dynamic optimality conjecture

(2) Greedy binary search tree algorithm

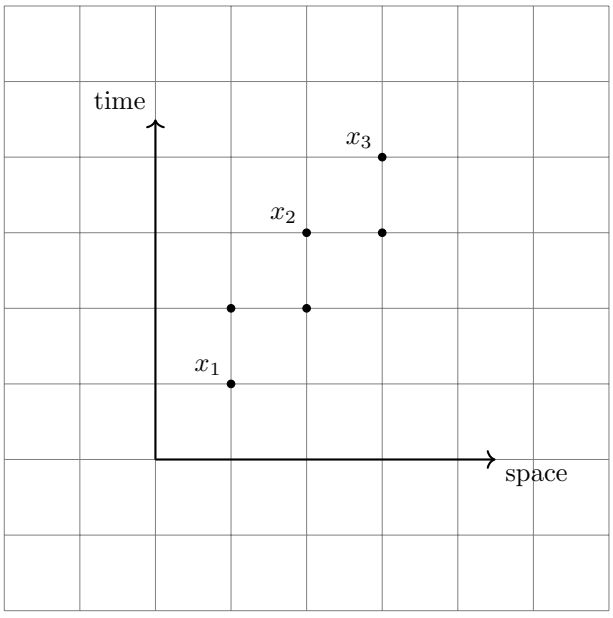
(3) Wilber's interleave lower bound

(4) Tango Trees

1 Geometric View of Dynamic Optimality

Given some access sequence $X = \{x_1, \ldots, x_n\}$, we can provide a geometric view of all nodes touched in performing the search sequence as $GV(X) := \{(i, j) \in [n]^2 : i \text{ is touched in the BST when accessing } x_j\}$.

Visually, $GV(X)$ on $[4]^2$ may look something like as follows:



This geometric view of nodes touched whilst searching for elements x_i in some access sequence X gives rise to a very natural geometric question, which can be phrased as follows:

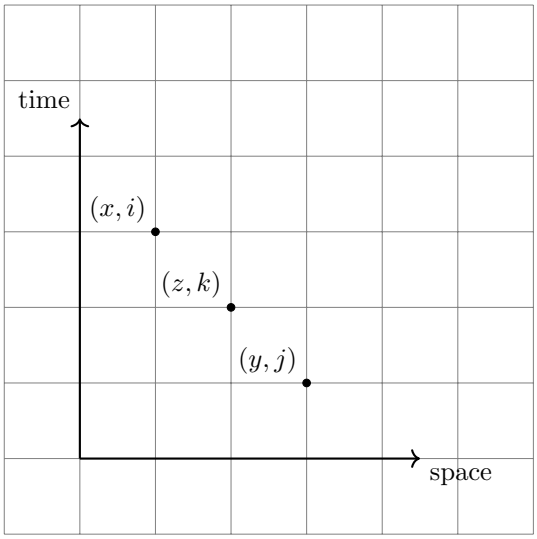
Question: Given an arbitrary subset $K \subset [n]^2$, does K correspond to a valid binary search tree execution for some access sequence X?

The answer to the above question is captured in the following theorem.

Theorem: $K \subset [n]^2$ corresponds to a valid BST execution of an access sequence $X = \{x_1, \ldots, x_n\}$ if and only if for any two points $x = (p, i), y = (q, j) \in K$ that do not lie on the same horizontal or vertical line, the rectangle spanned by x and y, denoted R_{xy}, contains another point. In particular, we call such a set **arborly satisfied**.

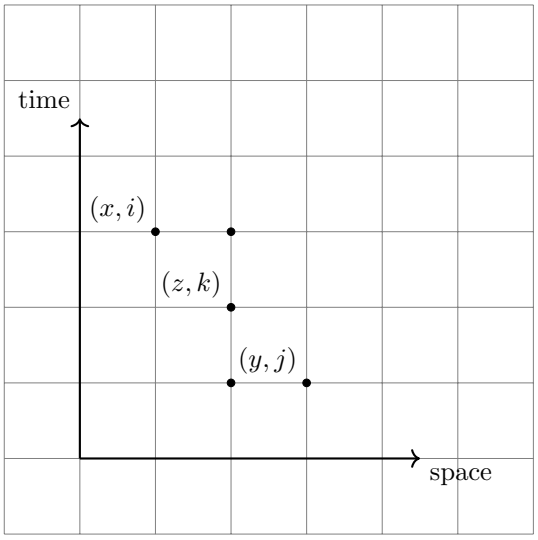
Before we provide a proof, we note that we can strengthen the statement to be "There exists a third point $z \in \partial(R_{xy})$." where ∂ is used to denote boundary. To provide a recursive explanation for this, supposed that $z \in R_{xy} \setminus (R_x y)$. Then, we see R_{xz} and R_{zy} are also valid rectangles of strictly smaller size, which, in order to be in a valid construction of a access sequence on a BST, must each respectively contain another point.

We also illustrate this with a visual example. Consider the following three points, $(x, i), (y, j), (z, k)$, arranged as follows in the plane:



Clearly, the rectangle R_{xy} satisfies the outline property, as z is contained within. However, the rectangles R_{xz} and R_{zy} do not contain any further points, so the outlined example does not correspond to a valid access sequence.

However, through adding in two more points, we can get a valid access sequence on some BST illustrated as follows:

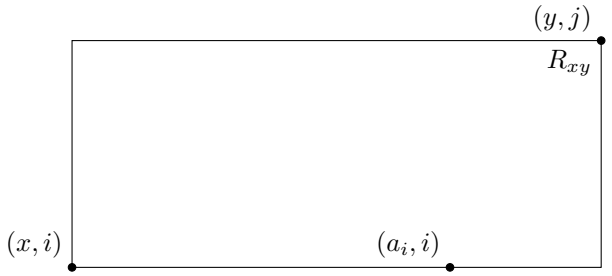


Now that we have provide examples and counterexamples of arborly-satisfied $K \subset [n]^2$, we now prove the theorem.

Proof. ($\Rightarrow$) Suppose we are given $K \subset [n]^2$ which corresponds to an access sequence on some BST. This may look like the following:

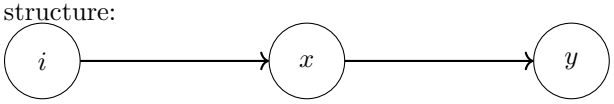
We will show R_{xy} contains a third point on its boundary somewhere. In particular, let $a_t :=$ *least-common-ancestor*$_t(x, y)$ or $lca_t(x, y)$ on the t^{th} BST, T_t. We have several cases to consider.

Case 1: If $a_i \neq x$, then, since x is touched, a_i must also be touched as well. Given the BST structure, this implies that $x < a_i \leq y$. Thus, we must have (a_i, i) on the boundary of R_{xy}, the desired result. Visually, this appears as follows:

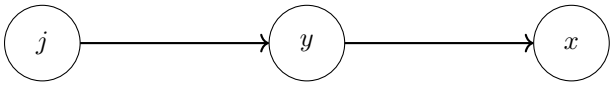


Case 2: In this situation, we have $a_j \neq y$, and so by an identical argument, we must have (a_j, j) on the boundary of R_{xy}.

Case 3: In this situation, we have $a_i = x$ and $a_j = y$. Visually, this means, on the ith tree, we have the structure:



And likewise, on the jth tree, this means we have the following structure:

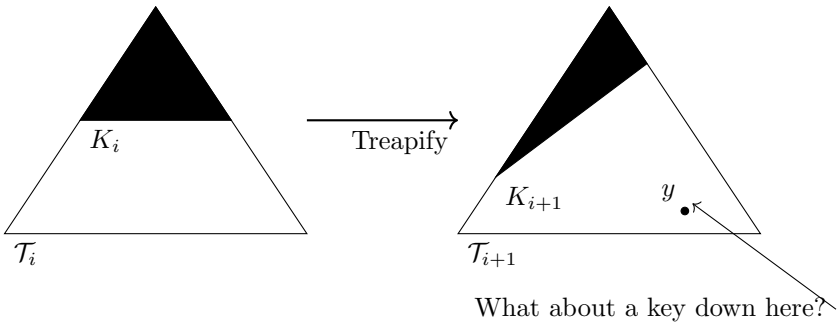


From this information, we see that there must have been some rotation involving the node containing x, which precisely means that (x, k) is touched, $i < k \leq j$. But this is a new point on the boundary of R_{xy}, so we are done with this case. Thus, we have completed the forward direction of the proof.

($\Leftarrow$)We pause before continuing with the reverse direction of the proof in order to introduce a necessary construction: the Treap. At a high level, a Treap is exactly what it sounds like. That is, it is a combination of a binary search tree and min-heap. It is a two dimensional tree where the ordering on the first coordinate obeys the standard BST ordering and the ordering on the second coordinate obeys the min heap ordering.

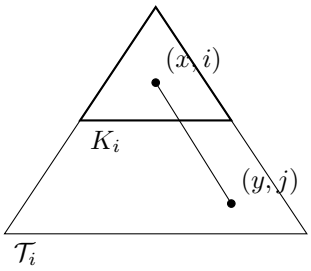
It is somewhat straightforward to see, for an arbitrary set of points $S = \{(x_i, p_i)\}_{i=1}^n$, that a Treap exists, as we can construct one by simply inserting S in order of priorities p_i and not rebalance. The question of rebalancing, however, is a more difficult one.

We first note that the theorem, as stated above, is existential, so it suffices to construct a Treap in an offline manner. We will assign priorities to x_i based on the next touch time of x_i, which we will denote as $NTT(x_i)$. More formally, if we let K_i be the set of nodes touched at time i, at the start of any search, K_i will have the lowest priority, and thus will be a local BST at the top of the Treap. After touching each node k, we reassign its priority to be $NTT(k)$. Following the end of our search, we perform the standard local heapify, or "treapify", which will only take $O(|K_i|)$, and will position K_{i+1} at the top of the tree. Visually, this looks the following:

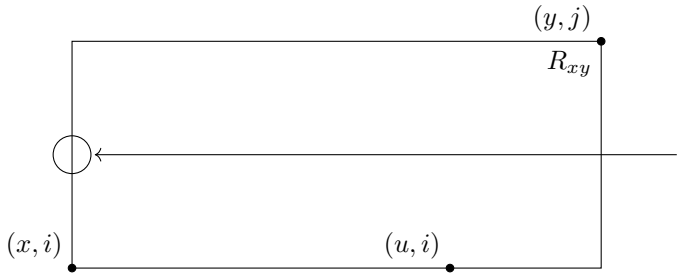


The main situation that seems concerning for the above picture is as follows. Consider $(x, i) \in K_i$ and (y, j) its child, where $i < j$. What seems to be an issue in the heapifying process is if $NTT(x) > NTT(y) = j$.

We show now that this is impossible.



We use the diagram of R_{xy} below as an aid for our argument. In particular, since $NTT(x) > j$, then the vertical face of the rectangle above x must be empty. Likewise, the vertical face of R_{xy} below y must be empty. What remains is to argue that both horizontal faces are empty as well. We argue in the case of the lower face. Suppose there was some u on the aforementioned face. Then, we would have $x < u < y$, so thus u is a left descendant of y by the BST ordering property. But then y must be touched at time i, which is a contradiction. Thus, this worrisome scenario cannot occur, finishing the proof.



■

Corollary: Finding the Optimal BST for an access sequence X is equivalent to finding the minimum ABS for X.

Open Problems

* What is the complexity of computing an ABS? NP-Hard? Is it hard to approximate within a constant factor? [Good final project topic]

2 Greedy binary search tree algorithm

There is a simple greedy offline algorithm that constructs an ABS: just do a line sweep and add points necessary to make the set ABS. This algorithm is conjectured to be $O(Opt)$.

In fact, we can show that we can simulate a greedy offline algorithm within a constant factor with a greedy online algorithm. The corollary of this is:

Corollary 1. *If the greedy offline algorithm is $O(Opt)$, then there exists an online algorithm that is $O(1)$-competitive.*

The theorem is as follows:

Theorem 2. *The greedy offline algorithm can be simulated with a greedy online algorithm with loss up to a constant factor.*

Proof. (sketch) Instead of heapifying nodes right away, we will delay that decision by putting them in a split tree. A split tree allows us to search for a node x, move it to root, and then split it to its left and right subtrees in $O(1)$ time. Our BST will be treap of split trees, and then we can do search and move-to-front in $O(1)$ time with this data structure.

■

3 Wilber's interleave lower bound

3.1 Definition of the lower bound

Our setting will be as follows. Fix any reference tree P on $[n]$. For all access sequence $x = x_1, \ldots, x_n$, $y \in P$, we define $\mathscr{L}_x(y)$ as the sequence of left/right access for y over the access sequence x. Every time an access touches y or its left child, we count that as a left access. Every time an access touches y's right child, we count that as a right access. Next, we define $alt_x(y)$ as the number of left/right alternations in y. For example, we can look at the access sequence $x = \{0, 1, 2, 3, 4, 2\}$ in the following tree:

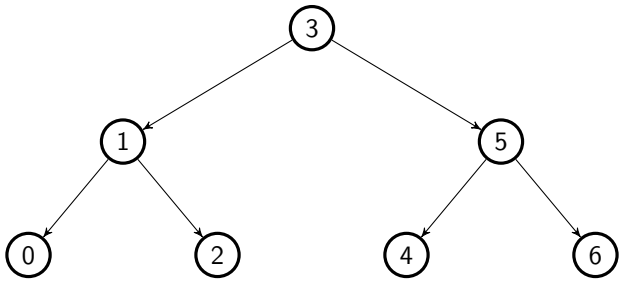


Figure 1: Reference tree P

In this case, $\mathscr{L}_x(y) = \{L, L, L, R, L\}$ and $alt_x(y) = 3$. We have:

$$IB_P(x) = \sum_{y \in P} alt_x(y)$$

Then, we can define the lower bound as follows:

Theorem 3. *(Wilber's interleave lower bound)*

$\forall$ *reference tree* P, $\forall$ *access sequence* x*:*

$$OPT(x) \geq \frac{IB_P(x)}{2} - n$$

3.2 Proof of the lower bound

The proof of the lower bound relies crucially on the notion of a *transition point*. Let the configuration of the optimal tree at time i be T_i. For every node $y \in P$, at any time point i, the *transition point* $tp_i(y)$ is the shallowest node z in the tree T_i whose path to the root contains both elements in the left subtree of y, $L(y)$, and the right subtree of y, $R(y)$.

The ingredients to the proof of the lower bound are the following three observations about transition points.

1. Transition points are well-defined, i.e there exists a unique transition point $tp_i(y)$ for each point $y \in P$ at each time point i.
2. Transition points are stable, i.e if the transition point of y does not change from i to $i+1$ unless it is touched by a search during this period.
3. Transition points are distinct for each node, i.e for $x, y \in P$, $x \neq y$, $tp_i(x) \neq tp_i(y)$.

We will prove these observations below.

Lemma 4. *$\forall y \in P$, at each time point i, $tp_i(y)$ is unique.*

Proof. We start with the observation that in a binary of keys $[n]$, any subtree forms an interval of consecutive numbers. Define l_i as the lowest common ancestor of all nodes in $L(y)$ and r_i as the lowest common ancestor of all nodes in $R(y)$. Our next observation is that in a binary search tree, the lowest common ancestor of two nodes is non-strictly between those two nodes (this can be proven via a case analysis). This second observation implies that l_i and r_i are in between the values in $L(y)$ and $R(y)$, respectively, and since the first observation implies that $L(y)$ and $R(y)$ form an interval of consecutive numbers, we know that l_i and r_i must be in $L(y)$ and $R(y)$, respectively. Now, we see that the lowest common ancestor of all nodes in $L(y)$ and $R(y)$ must also be either in $L(y)$ and $R(y)$, by the same argument as before, so it must either be l_i or r_i, whichever one is shallower. Assume that the lowest common ancestor is l_i, then the transition point is r_i: its path to root include nodes from both $L(y)$ and $R(y)$, and any node shallower than it does not have that property. Thus we can see that $tp_i(y)$ is well-defined as the deeper node between the lowest common ancestor of $L(y)$ and $R(y)$ in T_i. ■

Lemma 5. *If the transition point $tp_i(y)$ is not touched during period i, then $tp_i(y) = tp_{i+1}(y)$.*

Proof. Suppose l_i is the ancestor of r_i, where l_i and r_i are defined as in lemma 3 (the other case follows symmetrically). Since r_i is not touched and consequently no node in $R(y)$ is touched, it will still be the lowest common ancestor of all nodes in $R(y)$. The lowest common ancestor $L(y)$ might change, but we observe that it will still be shallower than $R(y)$. Initially during time i there were nodes in $L(y)$ shallower than r, and we did not touch r, so we cannot alter the subtree of r by inserting any of these nodes into r, and therefore we know these nodes must still remain shallower than r after time $i+1$. Thus, the lowest common ancestor l_{i+1} of $L(y)$ can only be among these nodes, and thus l_{i+1} is still the ancestor of $r_i = r_{i+1}$. Thus $tp_i(y) = tp_{i+1}(y)$. ■

Lemma 6. *For $x, y \in P$, $x \neq y$, $tp_i(x) \neq tp_i(y)$*

Proof. Suppose x and y are not ancestors of each other in the reference tree P, then the left and right subtree for each node is disjoint, thus the transition point must be different.

Suppose x and y are ancestor/descendant of each other. We assume without loss of generality that x is the ancestor of y. We can examine l_y, r_y, where l_y and r_y are the lowest common ancestor for node $L(y)$, $R(y)$.

If $tp_i(x)$ is not in y's subtrees, then the transition point of y must differ from r_x. Otherwise, the transition point $tp_i(x)$ must be the lowest common ancestor of all nodes in y's subtree, and thus is the shallower node between l_y and r_y. We know that $tp_i(y)$, on the other hand, must be the deeper node between l_y and r_y. Thus, $tp_i(x) \neq tp_i(y)$.

■

With the three lemmas above, we can prove Wilber's lower bound.

Theorem. *(Wilber's interleave lower bound)*

$\forall$ *reference tree* P, $\forall$ *access sequence* x*:*

$$OPT(x) \geq \frac{IB_P(x)}{2} - n$$

Proof. The time of the optimal tree is simply the number of times all nodes are touched during its operations. The number of times all nodes are touched is surely larger than the number of times transition points are touched. Thus, we only need to lower bound the number of transition points touched.

By lemma 3 and lemma 5, we can calculate the number of times transition points are touched but the number of times the transition points of each node $y \in P$ are touched in sum across them. In order words:

$$\begin{aligned}
&\#\text{ of times transition points are touched} \\
=&\sum_{i=1}^{m} \sum_{y \in P} \#\text{ of times } tp_i(y) \text{ is touched during } i \\
=&\sum_{y \in P} \sum_{i=1}^{m} \#\text{ of times } tp_i(y) \text{ is touched during } i \\
=&\sum_{y \in P} \#\text{ of times transition points of } y \text{ is touched during the access sequence } x
\end{aligned}$$

Now, suppose $alt_x(y) = p$, which means that the access sequence x alternates between $L(y)$ and $R(y)$ $p-1$ times. Now we define the sequence of time points $j_1, j_2, j_3, \ldots, j_p$, where j_1 is the time point at which y is first accessed and $j_2, \ldots, j_p$ is the time point at which y's accesses switch (left to right, or right to left). We want to charge each of the alternation with at least 1 touch of y's transition point. We can focus on two consecutive even-odd time points in the sequence, j_k and j_{k+1} where k is even. Suppose without loss of generality that all even time points are left access and all odd time points are right access. Thus, j_k is a left access and j_{k+1} is a right access. If either the search in j_k or j_{k+1} touches y's transition point, then we get to charge one of these points, say j_k to be consistent, with 1 touch. If neither searches in j_k or j_{k+1} touches y's transition point, then we examine what happens in between. Suppose that between l and r (the other case can be proven by symmetry), as define in lemma 3, l is the ancestor and r is the transition point. We claim that the algorithm must have touched r between j_k and j_{k+1}. Suppose not, then by 4, $tp_{j_k}(y) = tp_{j_{k+1}}(y)$. But then this means that at time point $tp_{j_{k+1}}(y)$, $tp_{j_{k+1}}(y)$ must have been touched, since the $tp_{j_k}(y) = tp_{j_{k+1}}(y) = r$ and j_{k+1} is a right access. Thus we get a contradiction. Therefore, the transition point must have been touched in between j_k and j_{k+1}. Thus, for each even access we can charge 1 touch of the transition point. This yields $\lfloor \frac{p}{2} \rfloor \geq \frac{p}{2} - 1$ times where touch the transition point of y during the entire access sequence. Summing over all y's, this yields us $\sum_{y \in P} \frac{alt_x(y)}{2} - n = \frac{IB_P(x)}{2} - n$.

■

3.3 Application: a sequence that is hard for all trees

As an application of Wilber's lower bound, we will construct a hard access sequence that is simultaneously hard for any tree. Consider the monotonic access sequence, but with their bits reversed. So, the sequence $0, 1, 2, 3, 4, 5, 6$, which has binary representation $000, 001, 010, 011, 100, 101, 110$. The bit-reverse sequence is $000, 100, 010, 110, 001, 101, 011$, which is $0, 4, 2, 6, 1, 5, 3$. Now assume that our tree is the perfect binary tree in figure 3.1. Observe that this sequence will force each node to alternate between its left and right child every time one of its children is touched. This means that:

$$alt_x(y) = \#\text{ of children of } y$$

$$IB_p(x) = \sum_{y \in P} alt_x(y) = \sum_{y \in P} \#\text{ of children of } y = \Theta(n \log n)$$

The Wilber lower bound implies that $OPT(x) \geq= \frac{IB_p(x)}{2} - n = \Theta(n \log n) - n = \Omega(n \log n)$. Thus this sequence will take $\Omega(n \log n)$ time for any trees.

4 Tango trees

4.1 Construction of Tango trees

The main idea of tango trees is to simulate the Wilber lower bound. Integral to this is the idea of a *preferred child*. At any time point, a *preferred child* of a node y is the left or the right child, depending on whether the left or right subtree was last accessed. If the element has not been accessed, then it has no preferred child. The following examples illustrates this idea:

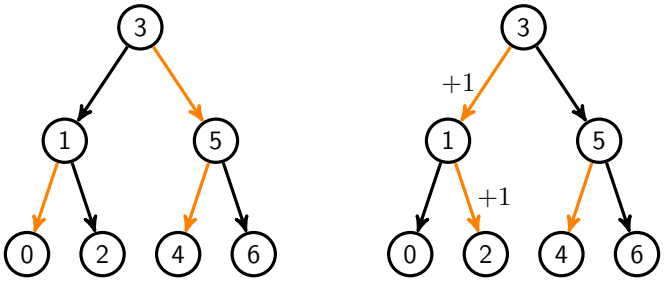


Figure 2: A tree T, before (left) and after (right) the key 2 is accessed.

We see that the switch in the preferred child is exactly the same as the alternations between left and right in the Wilber lower bound. Above, we see how the preferred children changes as key 2 is accessed. The key observation is that the Wilber lower bound also incurs a cost of 1 on the interleaving at key 3 and key 1 when key 2 is accessed. Thus, we can traverse across non-preferred edge essentially "for free," since whenever we are incurring a cost for traversing a non-preferred edge, Wilber's lower bound is also incurring the same cost. Since we only have to pay for traversing along preferred paths (which is a series of incident preferred edges), we only need to minimize this cost.

A simple solution is to throw the nodes on the preferred paths into balanced auxiliary trees, more specifically, red-black trees. Since there are at most $\log n$ nodes on the path, the red-black tree will have depth $\log \log n$.

If we can achieve this then the cost of our tree will be within $\log \log n$ factor of the Wilber bound, since we incur a cost of $\log \log n$ for each alternation, since we have to traversed through $\log \log n$ to get to the next non-preferred edge.

4.2 Dealing with auxiliary trees

An issue we need to deal with, however, is the fact that each access query will change our preferred paths. What we need to do each time the preferred path changes is cut off the old preferred path at the node where we jumped off and then link that node to our new preferred path.

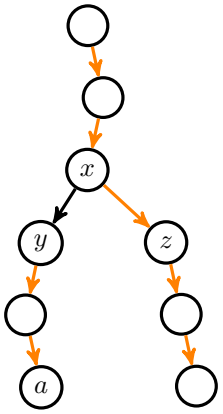


Figure 3: Preferred path changes when we query a.

For example, in figure 4.2 above, we see that when we access key a, we will jump off our old preferred path at x. This requires us to cut off the auxiliary tree at x into two auxiliary trees: one containing nodes above x and the other containing nodes below x. Then, we can concatenate the tree containing nodes above x to the tree auxiliary for the path starting from node y.

We can use the split-concatenation operation from red-black trees to do this in $O(\log \log n)$ time. The split operation allows us to split a tree into nodes smaller than a key x and nodes larger than a key x. It seems like this does not exactly correspond to our situation, since we are trying to split by depth rather than value. But we note that since we are working with a BST, the nodes in the preferred path rooted at y forms an interval. This means that we can cut this path out with two split operations: 1) split with value equal to the min of the interval and 2) split with value equal to the max of the interval. We can join the subtree at y with one concatenation operation. Both split and concatenate take logarithmic time in the size of the red-black tree. Thus, we can do all of these operations in $O(\log \log n)$ time.