6.851: Advanced Data Structures Spring 2012

Lecture 5 — March 1, 2012

Prof. Erik Demaine *Scribes: Brian Basham (2012), Travis Hance (2012), Jayson Lynch (2012)*

1 Overview

In the next two lectures we study the question of dynamic optimality, or whether there exists a binary search tree algorithm that performs "as well" as all other algorithms on any input string. In this lecture we will define a binary search tree as a formal model of computation, show some analytic bounds that a dynamically optimal binary search tree needs to satisfy, and show two search trees that are conjectured to be dynamically optimal. The first is the splay tree, which we will cover only briefly. The second will require us to build up a geometric view of a sequence of binary search tree queries.

2 Binary Search Trees

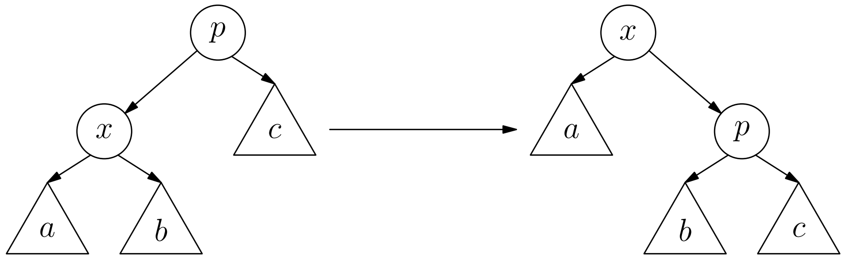
The question we will explore in this lecture is whether or not there is a "best" binary search tree. We know that there are self-balancing binary search trees that take $O(\log n)$ per query. Can we do better? In order to explore this question, we need a more rigorous definition of binary search tree.

In this lecture we will treat the binary search tree (BST) as a model of computation that is a subset of the pointer machine model of computation.

2.1 Model of Computation

A BST can be viewed is a model of computation where data must be stored as keys in a binary search tree. Each key has a pointer to its parent (unless it is the root) and a pointer to its left and right children, or a null pointer if they do not exist. The value of the key stored in the left child of a node must be less than or equal to the value of the key stored at the node, which is in turn less than or equal to the value of the key stored at the right child. The model supports the following unit-cost operations:

- Walk to left child.
- Walk to right child.
- Walk to parent.
- Rotate node x.



The first three operations only change a pointer into the tree. The last operation updates the tree itself.

These models support the query *Search*(x), which starts with a pointer to the root and can use the above operations, but must at some point visit the node with key x. In these two lectures we will assume for convenience that queries are always successful. We will also ignore insertions and deletions, for ease of definitions and proofs.

2.2 Is There a Best BST?

We already know of a number of self-balancing BSTs that take $O(\log n)$ per search. These include AVL trees and red-black trees.

Question: Is $O(\log n)$ the best possible?

Answer: Yes – in the worst case. Any tree on n items must have depth $\Omega(\log n)$, and an adversary could choose to search for the key located at the largest depth every round. However, in some cases we can do better. In general, we will consider a sequence of searches, and consider the best possible total time to complete the entire sequence.

2.3 Search Sequences

We will assume the n keys stored in the tree have values $\{1, 2, \ldots n\}$. We will consider sequences of search operations $x_1, x_2, \ldots x_m$, ordered by time. Some sequences are intuitively "easier" than other sequences. For example, If $x_1 = x_2 = \cdots = x_m = X$, then any search tree with X at the root can achieve constant time access per search.

We will investigate some possible properties of BST algorithms that guarantee certain access time bounds for specific input sequences.

2.4 Sequential Access Property

A BST algorithm has the *Sequential Access Property* if the search sequence $\{1, 2 \ldots n\}$ takes an amortized $O(1)$ time per operation.

This property seems easy to achieve, as it constitutes performing an in-order tree traversal in $O(n)$ time. It is slightly more complicated in this model of computation, as all searches must start at the root. However, starting at the last key can be simulated by rotating to the root, which we will not prove in this lecture.

2.5 Dynamic Finger Property

A BST algorithm has the *Dynamic Finger Property* if, for any sequence of operation $x_1, x_2, \ldots x_m$, the amortized access time for x_k is $O(|(x_k - x_{k-1})|)$.

This is a generalization of the Sequential Access Property. The Dynamic Finger Property tells me that as long as my queries remain close in space, the time needed will be small.

The Dynamic Finger Property can be achieved by a BST with some difficulty. In a more general pointer machine model, this is easy to achieve with a level-linked tree, a BST with pointers between adjacent nodes at every level.

2.6 Static Optimality/Entropy Bound

A BST algorithm is *Statically Optimal* if, given an input sequence where element k appears a p_k fraction of the time, the amortized access time per search is

$$O\left(\sum_{k=1}^{n} p_k \log \frac{1}{p_k}\right)$$

This is the information theoretic lower bound for the amortized access time for a static tree, hence the name static optimality.

2.7 Working Set Property

A BST algorithm has the *Working Set Property* if for a given search for x_i, if t_i distinct elements were accessed since the last access of x_i, the search takes an amortized $O(\log t_i)$.

The Working Set Property implies Static Optimality (although we will not prove it). If a few items are accessed often in some subsequence, the Working Set Property guarantees that these accesses are fast.

The Working Set Property says that keys accessed recently are easy to access again. The Dynamic Finger Property says that keys close in space to a key recently accessed are also easy to access. The following property will combine these.

2.8 Unified Property

A BST algorithm has the *Unified Property* [4] if, given that $t_{i,j}$ unique keys were accessed between x_i and x_j, then search x_j costs an amortized

$$O(\log(\min_{i<j}(|x_i - x_j| + t_{i,j} + 2)))$$

This is a generalization of both the Working Set Property and the Dynamic Finger Property, and implies both. If there is a key that was accessed somewhat recently and is somewhat close in space, this access will be cheap.

It is unknown whether or not there is any BST that achieves the Unified Property. This property can be achieved by a pointer machine data structure [4]. The best upper bound known is a BST that achieves an additive $O(\log\log n)$ factor on top of every operation.

2.9 Dynamic Optimality

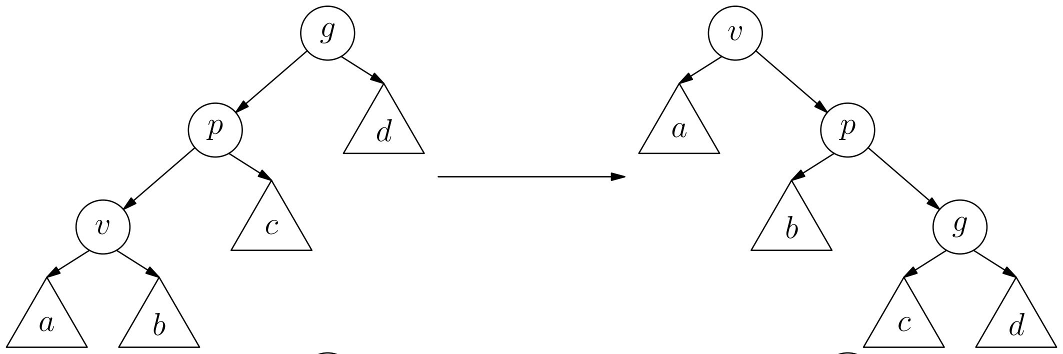
A BST algorithm is *Dynamically Optimal* if the total cost of a sequence of searches is within a multiplicative $O(1)$ factor of the optimal BST algorithm for that sequence. The optimal is the offline optimal, the min cost over all BST algorithms once a particular sequence is known. Dynamic Optimality implies that a particular online BST algorithm can achieve a constant factor approximation of the cost of the offline optimal algorithm.

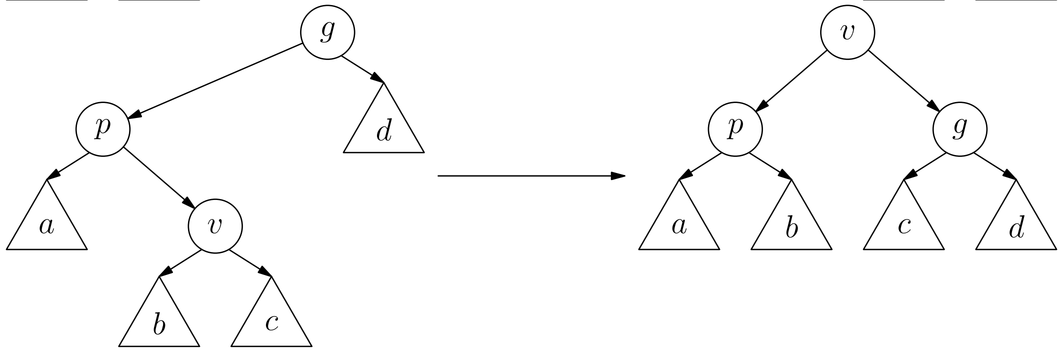
Open Questions: Is there an online dynamically optimal BST algorithm? Is there an online dynamically optimal pointer machine algorithm (either competitive with the best offline BST or offline pointer machine?)

Answer: We don't know any of the above. The best known is an online $O(\log\log n)$-competitive algorithm, which we will study next lecture. However, there are some candidates that are conjectured to be dynamically optimal, although a proof is not known.

3 Splay Trees

Splay trees were introduced by Sleator and Tarjan [5] in 1985. In a splay tree, a search for key x will start at the root and go down the tree until the key x is reached. Then, x is moved to the root using the following two "splay" operations, the *zig-zig* and the *zig-zag*:





In the zig-zig, we rotate y and then rotate x. In the zig-zag step, we rotate x twice. Splay trees differ from the "Move-to-root" algorithm because of the zig-zig operation, instead of simply rotating x twice. We perform these operations until x is either at the root of the tree, or it is a child of the root, in which case we perform one single rotation on x.

3.1 Analytic properties

Splay trees have been shown to have some of the properties discussed above. In particular, splay trees:

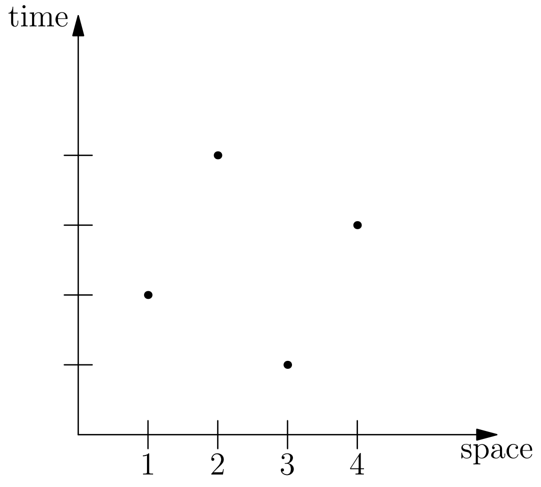
- are statically optimal, [5]
- have the Working Set Property, [5]
- have the Dynamic Finger Property. This was proven in 2000 by Cole et. al. [1][2]

It is not known whether splay trees are dynamically optimal, or even if they satisfy the Unified Property. They were conjectured to be dynamically optimal in the original paper [5], but this conjecture is still open.

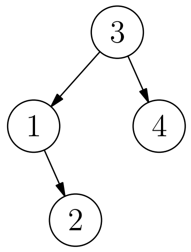
4 Geometric View

It turns out that there is a very neat geometric view of binary search tree algorithms, from [3]. Suppose on the key set $\{1, 2, \ldots, n\}$, we have an access sequence $x_1, \ldots, x_m$. We can represent this using the points (x_i, i).

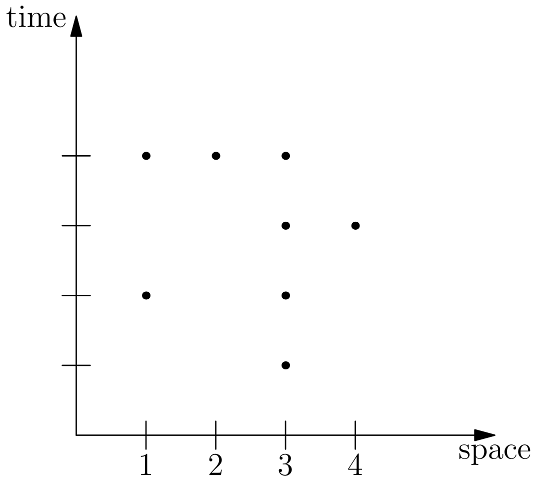
Example: If our key set is $\{1, 2, 3, 4\}$ and our access sequence is $3, 1, 4, 2$, we would draw this picture:



We could perform the access with this static binary tree:



In the first access, we only have to touch the 3. In the second access, we have to touch the 3 and the 1. In the third access, we have to touch the 3 and the 4. In the fourth access, we have to touch the 3, the 1, and the 2. We also represent these touches geometrically. If we touch item x and time t, then we draw a point at (x, t).



In general, we can represent a BST algorithm for a given input sequence by drawing a point for each item that gets touched. We consider the cost of the algorithm to be the total number of points; when running the BST, there is never any reason to touch a node more than a constant number of times per search. Thus, the number of points is within a constant factor of the cost of the BST.

Definition. We say that a point set is *arborally satisfied* if the following property holds: for any pair of points that do not both lie on the same horizontal or vertical line, there exists a third point which lie in the rectangle spanned by the first two points (either inside or on the boundary).

Note that the point set for the BST given in the example is arborally satisfied. In general, the following holds:

Theorem. A point set containing the points (x_i, i) is arborally satisfied if and only if it corresponds to a valid BST for the input sequence $x_1, \ldots, x_m$.

Corollary. The optimum binary search tree execution is equivalent to the smallest arborally satisfied set containing the input.

OPEN: What is the computational complexity of finding the smallest arborally satisfied set? Is there an $O(1)$-approximation?

Proof. First, we prove that the point set for any valid BST algorithm is arborally satisfied. Consider points (x, i) and (y, j), where x is touched at time i and y is touched at time j. Assume by symmetry that $x < y$ and $i < j$. We need to show that there exists a third point in the rectangle with corners as (x, i) and (y, j). Also let $\text{lca}_t(a, b)$ denote the least common ancestor of nodes a and b right before time t. We have a few cases:

- If $\text{lca}_i(x, y) \neq x$, then we can use the point $(\text{lca}_i(x, y), i)$, since $\text{lca}_i(x, y)$ must have been touched if x was.
- If $\text{lca}_j(x, y) \neq y$, then we can use the point $(\text{lca}_j(x, y), j)$.
- If neither of the above two cases hold, then we must have x be an ancestor of y right before time i and y be an ancestor of x right before time j. Then at some time k ($i \leq k < j$), y must have been rotated above x, so we can use the point (y, k).

Next, we show the other direction: given an arborally satisfied point set, we can construct a valid BST corresponding to that point set. Now we will organize our BST into a treap which is organized in heap-order by next-touch-time. Note that next-touch-time has ties and is thus not uniquely defined, but this isn't a problem as long as we pick a way to break ties. When we reach time i, the nodes touched form a connected subtree at the top, by the heap ordering property. We can now take this subtree, assign new next-touch-times and rearrange into a new local treap. Now, if a pair of nodes, x and y, stradle the boundary between the touched and untouched part of the treap, then if y is to be touched sooner than x then $(x, \mathit{now}) \to (y, \mathit{next} - \mathit{touch}(y))$ is an unsatisfied rectangle because the leftmost such point would be the right child of x, not y. ■

4.1 Greedy Algorithm

There is a simple greedy algorithm to construct arborally satisfiable sets. We consider the point set one row at a time. Now add any points to that row that would be necessary to make the current subset satisfiable. This is repeated until all rows of points are satisfied. It is conjectured that this greedy algorithm is $O(\mathit{Opt})$ or event $\mathit{Opt} + O(m)$.

Theorem. The online arborally satisfiable set algorithm implies an online BST algorithm with $O(1)$ slowdown.

Corollary. If the greedy algorithm is $O(\mathit{Opt})$, then we have an algorithm for dynamic optimality.

Proof. First, store the touched nodes from an access in a split tree. Split trees can move a node to the root, and then delete that node leaving two trees in amortized $O(1)$ time. This allows us to perform the reordering and separation based on touched nodes for all n nodes in only $O(n)$ time. Now we can essentially construct a BST which is essentially a treap of split trees ordered by the previously touched node. These trees allow us to efficently touch the predecessor and successor nodes in the parent tree when touching a node in the split tree. Thus we are able to simulate the decisions from the arborally satisfiable set algorithm with only only a constant factor slowdown.

References

[1] Richard Cole, Bud Mishra, Jeanette P. Schmidt, Alan Siegel: *On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences.* SIAM J. Comput. 30(1): 1-43 (2000)

[2] Richard Cole: *On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof.* SIAM J. Comput. 30(1): 44-85 (2000)

[3] Erik D. Demaine, Dion Harmon, John Iacono, Daniel M. Kane, Mihai Patrascu: *The geometry of binary search trees.* SODA 2009: 496-505

[4] John Iacono: *Alternatives to splay trees with O(log n) worst-case access times.* SODA 2001: 516-522

[5] Daniel Dominic Sleator, Robert Endre Tarjan: *Self-Adjusting Binary Search Trees* J. ACM 32(3): 652-686 (1985)