6.851: Advanced Data Structures Spring 2010

Lecture 1 — 2 February, 2010

Prof. Erik Demaine *Scribe: Kevin Kelley*

1 Administrivia

Homework is subject to a one-page-in, one-page-out rule; that is, assignments will be kept to one page, but you may only turn in a single page in response. Your single page must be typeset with LaTeX. A template is available on the course website.

2 Overview

Today's topic is the question of dynamic optimality; that is, of whether there is a single "best" binary search tree. (A more rigorous definition of "best" follows.)

3 Binary search trees

The binary search tree is typically viewed as a comparison data structure: smaller things are placed to the left, and larger things are placed to the right. However, the binary search tree can also be treated as a model of computation.

This model defines several unit-cost operations. From a given position in the tree, it takes $O(1)$ time to walk to either the left or right child, to walk to the parent, or to rotate a node and its parent. (Rotation is a simple tree operation in which a node x takes the place of its parent y, with y becoming a child of x. The two subtrees of x and the other subtree of y are redistributed such that the structure of the binary search tree is preserved.)

3.1 Types of binary search trees

There are many types of data structures which meet these criteria. (All except tango trees provide $O(\lg n)$ lookups, insertions, and deletions.)

- *Simple binary search trees* – The naïve implementation.
- *AVL trees* – The first self-balancing tree.
- *Red/black trees* – A self-balancing alternative to the AVL tree which provides the same asymptotic bound but can be faster for workloads heavy in insertions and deletions.
- *Treaps* – A hybrid between a binary search tree and a heap (hence the name). Similar to a randomized binary search tree. Provides amortized guarantees.

- *Weight-balanced trees* – Similar to a treap, but constructed to that frequently-accessed elements are towards the root.
- *Splay trees* – Uses a particular method of moving accessed elements to the root of the tree in order to provide fast lookups for recently-accessed keys.
- *Tango trees* – To be covered next lecture.

3.2 Beating $O(\lg n)$

The $O(\lg n)$ limit may seem like a strict lower bound. After all, if we have a set of n elements, we need $\lg n$ bits of information to uniquely identify each, and each pairwise comparison gives us a single bit. However, some data structures, such as tango trees, manage to do so. We will revisit this topic next lecture.

4 Properties of binary search trees

There are a number of variously-well studied properties that individual binary search tree data structures can be said to possess. Some imply others (that is, they are strictly stronger properties).

Before considering some of those properties, we begin with some definitions. Let the members of the set $\{1, 2, \ldots n\}$ be *keys* stored in the binary search tree. Then, let the x_i in the *access pattern* $\{x_1, x_2, \ldots x_m\}$ be keys. We ignore insertions and deletions for the purpose of simplicity.

It should be intuitively obvious that some access patterns are very simple. For example, if we had $x_i = k$ for all i, then placing k at the root would give us $O(1)$ access time across the board. In-order traversals are correspond to a particularly nice access pattern which can easily be satisfied in amortized constant time.

4.1 Static optimality (the entropy bound)

Static optimality. Let k be a key which appears in the access sequence with probability p_k. Then, the per-operation cost of access is

$$O\left(\sum p_i \lg \frac{1}{p_i}\right)$$

This is because an element which appears with probability p_i can be stored at height $\lg p_i$. This forms a strict lower bound if the tree is *static* (that is, cannot be modified as part of the lookup operation); this is where the term *static optimality* comes from.

It should be intuitive that we might be able to do better if we relax this restriction. (Indeed, of course, we can.)

4.2 Dynamic finger property

Dynamic finger property. If the last access was to key x_{i-1} and the present access is to key x_i, then the cost of the present access is $O(\lg |x_i - x_{i-1}|)$.

This is roughly equivalent to "close is cheap"– a data structure with this property performs well on access patterns which exhibit spatial locality.

4.3 Sequential access property

Sequential access property. An in-order traversal of the keyspace is achieved in amortized constant time per operation.

This is related to the dynamic finger property.

4.4 Working-set property

Working-set property. Let i be the largest index into the access pattern such that $x_i = x_j$ and $i < j$. Let t_j be the number of distinct keys in $\{x_i, \dots x_j\}$. Then the cost of accessing x_j is

$$O(\lg t_i + 2)$$

Less formally, let t_j the number of distinct keys between the present operation and the last time the same key was accessed. This is roughly equivalent to "recent is cheap"–a data structure with this property performs well on access patterns which exhibit temporal locality.

Furthermore, this property implies the entropy bound; that is, data structures with this property are statically optimal.

4.5 Unified property

Unified property. Let t_{ij} be the number of distinct keys in $\{x_i, \dots x_j\}$. Then the cost of accessing x_j is

$$O\left(\lg \min_i \left(|x_i - x_j| + t_{ij} + 2\right)\right)$$

It is more difficult to discover an intuitive explanation of this property by inspection. Broadly speaking, this property mandates that "close to recent is cheap"–access to keys close to keys which have been recently accessed is cheap. This is convenient for certain access patterns that (perhaps) resemble those that may be encountered in real-world applications.

There are a number of problems related to this property, the first and foremost of which is that it is unknown whether any binary search trees satisfy the property. Even so, this is still not equivalent to a strict lower bound on the performance of a binary search tree–we can do better.

4.6 Dynamic optimality

Dynamic optimality. Accesses are $O(1)$-competitive with an optimal offline, dynamic binary search tree.

(For those not familiar with the notation, we could rephrase the above by stating that, in order to be dynamically optimal, a data structure must provide bounds within a constant factor of those provided by an optimal offline, dynamic binary search tree.)

Whether or not any online binary search tree achieves this property is an open question, although tango trees are $O(\lg \lg n)$-competitive.

5 Splay trees

As mentioned briefly above, splay trees are a type of binary search tree data structure. They have many practical advantages. They are self-balancing, but are much simpler (from an algorithmic or implementation perspective) than other self-balancing options, such as AVL trees or red/black trees. Despite this, they require no additional bookkeeping information at each node and provide comparable average-case performance. They also provide a healthy subset of the properties we have previously discussed.

5.1 Manipulating the tree

At the highest level, splay trees are designed to bring an accessed node x to the root of the tree so that subsequent accesses are fast. However, doing so naïvely (by simply rotating x until it is at the root) is undesirable since it may unbalance the tree.

Instead, we perform one of two composite operations–the *zigzig* step, in which we rotate y and then rotate x, or the *zigzag* step, in which we rotate x twice–depending on the structure of the tree around x. Each of these steps moves x up twice; if x winds up a direct child of the root, we simply rotate it once more.

5.2 Analytic properties

- Entropy bound (static optimality)
- Working set property
- Dynamic finger property

Whether or not splay trees are dynamically optimal is an open question.

6 A geometric perspective

Let us treat the access sequence $\{x_1, \ldots x_n\}$ as a set of points $\{(x_j, i\}$ where the x_j are the nodes "touched" by the operation performed at time i (that is, the nodes traversed during the access to

x_i).

Arborally satisfied set. A set S of points is considered arborally satisfied iff $\forall x, y \in S$ s.t. the rectangle defined by x and y has nonzero area, that rectangle is nonempty.

6.1 Equivalence

Theorem. The point set defined above is a valid binary search tree execution iff it is an arborally satisfied set (ASS).

6.1.1 A forward proof

(Here we prove that if we have a valid binary search tree execution, the resulting set is arborally satisfied.)

Imagine that we have an arborally satisfied set of points, from which (without loss of generality) we pick x_i and x_j. Let a be the least common ancestor of nodes x_i and x_j at time i. Then, a must have been touched at time i, since the tree was traversed from its root to x_i, which necessarily involves passing over a.

This node a, then, must be within the rectangle formed by x_i and x_j–unless, of course, $x_i = a$. However, in this case, a must be touched again at time j when the tree is traversed from its root to x_j, and since $x_i \neq x_j$, this must be in the rectangle formed by x_i and x_j.

6.1.2 (A sketch of) a reverse proof

(Here we prove that if we have an arborally satisfied set, the corresponding binary search tree execution is valid.)

In very general terms, the reverse proof involves a treap whose heap key is the time at which the node will next be touched. However, a full explanation was postponed until the next lecture.

6.2 To what end?

Why should we care about this perspective on the problem?

Imagine, for a moment, a greedy, offline algorithm which rearranges the nodes it touches in order of the time at which they will next be accessed. Geometrically, this is equivalent to adding a minimal number of points per line in a time-space plot of the point set.

Conjecture. This approach is within a constant factor of dynamic optimality.

This conjecture is obviously exciting, but is unproven.

Theorem. The online arborally satisfied set algorithm can be modified to become an online BST algorithm with only a constant-factor slowdown.

This theorem was not proven.

References

[1] M. Fredman, J. Komlós, E. Szemerédi, *Storing a Sparse Table with $O(1)$ Worst Case Access Time*, Journal of the ACM, 31(3):538-544, 1984.