A Survey on the Dynamic Optimality

Yiding Zhang zhangyd19@mails.tsinghua.edu.cn
Institute for Interdisciplinary Information Sciences
Tsinghua University
Beijing, China

Abstract

Dynamic optimality is an area that focuses on whether there is a "best" binary search tree (BST). In 1985, Sleator and Tarjan (1985) presented the dynamic optimality conjecture - the running time of Splay, a self-adjusting BST, is within a constant factor of the optimal offline BST on any sufficiently long sequence. In this survey, we present the progress in dynamic optimality that have been made since the conjecture was put forward.

Keywords: binary search tree, Splay tree, Tango tree, dynamic optimality

1. Introduction

A binary search tree (BST) is a rooted binary tree that can represent an ordered sequence as its in-order traversal. This data structure can support insertion, deletion, and query operations efficiently if we wisely adjust the tree structure by rotation, rebuilding, or something else. In this survey, we focus on the problem that whether there is a "best" BST. We first introduce some basic definitions related to dynamic optimality in this section, and then some upper bounds will be shown in Section 2. After that, we will introduce a neat geometric view of BST algorithms in Section 3, and then show some lower bound from this view in Section 4.

1.1 Model of computation

To define what is a "best" BST, we first need to formally define the model of computation - binary search tree model. In this model, data we need to maintain can only be stored as keys of the nodes in the BST. Each node has a pointer to its parent (null if it is the root), and a pointer to its left and right child respectively (null if it does not have such child). This model requires that for each node, its left child must have a smaller key value, and its right child must have a larger key value (suppose that keys are distinct).

The BST model supports the following unit-cost operations:

- Go to the left child.
- Go to the right child.
- Go to the parent.
- Rotate a node.

The model needs to support query operation **search**(x), which starts at the root of the tree, uses the above operations arbitrarily, and touches the node with key value x at some point.

In addition, each node can store $O(1)$ extra information (e.g., the color of each node in a red-black tree), and the algorithm can determine what to do next according to the extra information. Although there are BST algorithms that are not rotation-based (e.g., scapegoat trees), a series of rotations can change the structure of a BST arbitrarily, and thus the model can also fit these algorithms.

1.2 Dynamic optimality

In the dynamic optimality problem, we consider only query operations (i.e., access the node with the given key value), and ignore other operations like insertions and deletions for convenience. Therefore, the operations we need to deal with are just a access sequence $X = q_1, q_2, \ldots, q_m$ that asks **search**(q_i) in the i-th query. We define $R_A(X)$ as the total running time of BST algorithm A on the access sequence X.

For any access sequence X, define $OPT(X)$ as the minimum cost over all the offline BST algorithms (i.e., operations can depend on future queries) executing on X. Then we say that a BST algorithm A (of course an online algorithm) is **dynamically optimal** if it performs as well as the best offline BST on any access sequence. More precisely, the definition is as follows:

Definition 1 *(dynamic optimality) We say that an online BST algorithm A is dynamically optimal if for any access sequence X, we have $R_A(X) = O(OPT(X))$.*

For simplicity, in the following sections, we always suppose that elements are numbered from 1 to n in increasing order, and the access sequence is $X = q_1, q_2, \ldots, q_m$.

2. Upper bounds better than $O(\log n)$/op

In a BST consisting of n elements, the deepest element always has depth $\Omega(\log n)$. Therefore, $O(\log n)$ time per access is optimal in the worst case (since we can adaptively choose the deepest element). Almost all of the self-balancing BSTs can achieve such worst-case optimality. However, we can see that some access sequences are "easier" than others. For example, accessing the same value m times only requires $O(m)$ time. In fact, we have upper bounds that are better than $O(\log n)$ per operation on some special access sequences (although none of them are tight in general).

2.1 Dynamic finger bound

Intuitively, dynamic finger bound implies the spatial locality, i.e., an access can be fast if its key value is close to the previous access. More precisely, A BST algorithm has the dynamic finger property if the the (amortized) access time for q_i is $O(\log(|q_i - q_{i-1}| + 1))$ ($+1$ here is to avoid the $\log(0)$ case).

Dynamic finger property implies the sequential access property, which says that accessing $1, 2, \ldots, n$ in order takes $O(1)$ amortized time per operation. If we allow searching from the last visited element, any BST can achieve this because an in-order traversal always takes $O(n)$ time. But it is a little hard to implement under BST model.

Now we present some BSTs with such property and some related applications.

2.1.1 Splay

The dynamic finger property first appears as a conjecture in Sleator and Tarjan (1985). The conjecture is that the total access time is

$$R_{Splay}(X) = O(m + n + \sum_{i=2}^{m} \log(|q_i - q_{i-1}| + 1)).$$

Sleator and Tarjan (1985) also presents the proof of an easier version called static finger property:

Theorem 2 (static finger property) *For any fixed element f, we have*

$$R_{Splay}(X) = O(n \log n + m + \sum_{i=1}^{m} \log(|q_i - f| + 1)).$$

Proof We assign the weight of i as $w_i = 1/(|i-f|+1)^2$, and then apply the potential analysis. Note that the total weight of the tree is $\sum_{i=1}^{n} 1/(|i-f|+1)^2) \leq 2\sum_{k=1}^{+\infty} 1/k^2 = O(1)$. So the amortized cost of accessing i is $3(\mu(S) - \mu(i)) + 1 = O(\log(|i-f|+1))$. Note that the net potential of the tree is at least $\sum_{i=1}^{n} \log(1/(|i-f|+1)^2) = -2\sum_{i=1}^{n} \log(|i-f|+1)$. So the potential drop is bounded by $O(n \log n)$, and then we can get the total access time $O(n \log n + m + \sum_{i=1}^{m} \log(|q_i - f| + 1))$. ■

The dynamic finger property is proved by [Cole et al. (2000)][Cole (2000)], but with a higher initialization cost. The total access time is proved to be

$$R_{Splay}(X) = O(m + n \log\log n + \sum_{i=2}^{m} \log(|q_i - q_{i-1}| + 1)).$$

This is a very complicated proof of over 40 pages, and we will not go into details in this survey.

2.1.2 Treap

In the BST model, searches must start from the root. If we remove this constraint and allow searching from the last accessed element, Treap also has dynamic finger property.

Suppose that i is the last accessed element and j is the element we want to access now (i is the root if j is the first element of the access sequence). We implement $\mathbf{search}_i(j)$ (accessing j from i) in the following way: starting from i, going upward along the parent link to the least common ancestor of i and j (this can be done by recording the range of elements in each subtree), and then going to j. Now we prove that Treap has the dynamic finger property:

Lemma 3 *Each $\boldsymbol{search}_i(j)$ operation takes $O(\log(|j-i|+1))$ time in a Treap.*

Proof Suppose that the priority of parent node is higher in the Treap, and no two elements have the same priority. Then y is an ancestor of x if and only if y has the highest priority among the elements in $[\min\{x, y\}, \max\{x, y\}]$.

W.l.o.g, suppose that $i < j$. Consider the expected length of the path P from i to $lca(i, j)$, which consists of elements that are ancestors of i but not ancestors of j. For each element k ($k \neq i, j$), there are three possible cases:

1. $k < i$: $k \in P$ iff k has the highest priority in $[k, i]$, and some element $k' \in (i, j]$ has higher priority than k. So we have $\Pr[k \in P] = \frac{1}{i-k+1} - \frac{1}{j-k+1}$.
2. $i < k < j$: $k \in P$ iff k has the highest priority in $[i, k]$, and some element $k' \in (k, j]$ has higher priority than k. So we have $\Pr[k \in P] = \frac{1}{k-i+1} - \frac{1}{j-i+1}$.
3. $j < k$: If k is the ancestor of i, then it is also the ancestor of j. So $k \in P$ is impossible.

Then we can get

$$\begin{aligned}
\mathbb{E}[|P|] &\leq 2 + \sum_{k=1}^{i-1} \left(\frac{1}{i-k+1} - \frac{1}{j-k+1} \right) + \sum_{k=i+1}^{j-1} \left(\frac{1}{k-i+1} - \frac{1}{j-i+1} \right) \\
&= 2 + \sum_{k=2}^{i} \frac{1}{k} + \left(\sum_{k=2}^{j-i+1} \frac{1}{k} - \sum_{k=2}^{j} \frac{1}{k} \right) + \sum_{k=2}^{j-i} \frac{1}{k} - \frac{j-i-1}{j-i+1} \\
&= 2 + \sum_{k=2}^{j-i+1} \frac{1}{k} + \sum_{k=2}^{j-i} \frac{1}{k} - \sum_{k=i+1}^{j} \frac{1}{k} - \frac{j-i-1}{j-i+1} \\
&\leq 2 + \sum_{k=2}^{j-i+1} \frac{1}{k} + \sum_{k=2}^{j-i} \frac{1}{k} \\
&= O(\log(j-i+1))
\end{aligned}$$

Similarly we can get the expected length of the path from $lca(i, j)$ to j. So the expected running time is $O(\log(|j-i|+1))$.

■

2.1.3 Application - merging BSTs

Now we consider the problem of merging two BSTs. Note that we do not guarantee that the elements in one tree are all smaller than the elements in the other (in this case merging can be done in $O(\log n)$ time by Splay or Treap).

BSTs do not intrinsicly support merging two trees quickly like some heaps (e.g., Fibonacci heap). But we can implement merging in a heuristic way: inserting elements in the smaller tree into the larger one. Suppose that we finally get a BST with n nodes after some mergings. For each node, every time we insert it into another BST, the size of the BST containing it is at least doubled. So each node is inserted $O(\log n)$ times, and thus the total time complexity is $O(n \log^2 n)$.

The time complexity of merging can be improved by dynamic finger property. We only need to modify the heuristic algorithm a little bit: every time we merge two BSTs T_1 and

T_2 with $|T_1| < |T_2|$, insert elements of T_1 **in increasing (or decreasing) order** into T_2. Now we prove that this algorithm takes only $O(n \log n)$ time if we use BSTs with dynamic finger property. Each time we merge two trees T_1 and T_2 with size n_1, n_2 ($n_1 < n_2$), the time complexity (without the initialization cost) is

$$O\left(n_1 + \sum_{i=1}^{n_1} \log(d_i + 1)\right) = O\left(n_1 + n_1 \log\left(\frac{n_1 + n_2}{n_1}\right)\right)$$

(note that the last step is due to the concavity of log function and Jensen's inequality), where d_i denotes the distance of the i-th and $(i-1)$-th element of T_1 in the resulting tree. In such a merging operation, we count $O(1+\log((n_1+n_2)/n_1))$ on each node of T_1. Suppose that we finally get a tree with n nodes; for each element i, let $s_1 < s_2 < \cdots < s_k = n$ be the size of the trees after each insertion of i during the merging process. Then all the insertions of i cost

$$O(\sum_{i=2}^{k}(1 + \log(s_i/s_{i-1}))) = O(k + \log(s_k/s_1)) = O(\log n).$$

So the whole algorithm takes $O(n \log n)$ time. Note that the $O(n \log \log n)$ initialization cost can be ignored if we regard all these mergings as a whole process.

2.2 Working-set bound

The working-set bound says that searching an element that has been recently searched can be fast. More precisely, each access q_i requires only $O(\log(t_i + 1))$ time, where t_i is the number of distinct elements being accessed since the previous access of element q_i (or since the beginning of the sequence if q_i is the first access of this element).

Sleator and Tarjan (1985) proves that Splay can satisfy this bound:

Theorem 4 (Working-set theorem of Splay)

$$R_{Splay}(X) = O(m + n \log n + \sum_{i=1}^{m} \log(t_i + 1)).$$

Proof We can also prove this by setting the weight of each element properly. First, we sort the elements in increasing order of their first access (put the elements that have never been accessed at the end), and assign weight $1/i^2$ to the i-th element. Then after each access, we permute the weights of elements in the following way: suppose the root (i.e., the element having been accessed) has weight k; we change the weight of the root to 1, and assign the new weight $1/(k'+1)^2$ to each element with original weight $1/k'^2$ ($k' < k$). Note that the total weight is $\sum_{i=1}^{n} 1/i^2 = O(1)$. So each access operation has amortized cost $O(\log(t_i + 1))$. Now we consider the amortized cost of permuting the weights. After the permutation, we decrease the weight of some non-root elements, and only increase the weight of the root. Since the potential of the root always equals to $\mu(S) = \log(\sum_{i=1}^{n} 1/i^2)$, and the potential of other nodes can only decrease, the net potential can only drop or remain unchanged. So the amortized cost of permuting is either negative or zero. After adding the $O(n \log n)$ net potential drop, we get the total access time $O(m+n \log n+\sum_{i=1}^{m} \log(t_i+1))$. ■

2.2.1 Static optimality of Splay

The working-set property implies that if an element i appears $f(i)$ times in the access sequence, its average access time is bounded by $O(\log(m/f(i)))$ (we can prove this by the working-set property and the concavity of log function). So we have the following entropy bound:

Theorem 5 (entropy bound/static optimality theorem) *Suppose that each element i is accessed $f(i) \geq 1$ times in the access sequence X. Then we have*

$$R_{Splay}(X) = O\left(m + \sum_{i=1}^{n} f(i) \log\left(\frac{m}{f(i)}\right)\right).$$

The proof has been shown in class, and thus we just omit it. We call it entropy bound because the average accessing time of each element is $O(-\sum_{i=1}^{n}(f(i)/m)\log(f(i)/m))$, which is Shannon's entropy. Intuitively, Shannon's entropy is the best result that can be achieved by a static BST, and thus we call this property static optimality.

However, arguing that Shannon entropy represents the performance of the best static BST seems not very easy. From the view of encoding, we can try to construct such an optimal static BST that can meet this bound, just as what a Huffman tree does. But with the constraint on the elements' order in BST model, such a construction is not very easy to find. Instead of showing the optimality through entropy, we can prove the static optimality of Splay directly:

Theorem 6 *For any access sequence S that access each element at least once, the running time of Splay is $O(OPT_S(X))$, where $OPT_S(X)$ is the running time of the optimal static BST for X.*

Proof Suppose that the depth of i in the optimal static BST is d_i. We assign the weight of i as $w_i = 3^{-d_i}$. Since the number of vertices with depth d in a BST is at most 2^{d-1}, the total weight of the tree is always less than 1. So the amortized cost of accessing i is $3(\mu(S) - \mu(i)) + 1 = O(d_i)$. Then we consider the drop in potential. The total potential is always greater than $\sum_{i=1}^{n} \log(3^{-d_i}) = -\log(3)\sum_{i=1}^{n} d_i$. So the drop in potential is bounded by $O(\sum_{i=1}^{n} d_i)$. If we assume that each element is accessed at least once, then the running time of splay is $O(\sum_{i=1}^{m} d_{q_i}) = O(OPT_S(X))$. ■

2.3 Unified bound

Unified bound is a combination of dynamic finger bound and working-set bound. It implies both the dynamic finger bound and the working-set bound. The definition is as follows:

Definition 7 (unified property) *Suppose that $t_{i,j}$ is the number of distinct elements between access q_i and q_j. Then a BST has unified property if for each search q_i, the running time is bounded by*

$$O(\log(\min_{i'<i}\{|q_i - q_{i'}| + t_{i',i} + 2\})).$$

Iacono (2001) proves that there exists a BST with unified property. So unified bound is actually an upper bound for the optimal BST.

Whether Splay has the unified property is still open. We even have not found a BST with unified property, and the best BST known is Skip-Splay(Derryberry and Sleator (2009)) that requires $O(m \log\log n + UB(X))$ running time.

3. A geometric view of BSTs

Given an access sequence X, how can we get $OPT(X)$? This problem seems hard to study if we directly consider all possible rotation-based BSTs. Demaine et al. (2009) presents a geometric view that makes this problem easier to study by showing a connection between BSTs and points on the plane satisfying a simple property.

3.1 Equivalent representation of BST algorithms

For any access sequence $X = q_1, q_2, \ldots, q_m$, we can represent it as $S_X = \{(q_i, i) \mid 1 \leq i \leq m\}$, which is a set of points with x-axis being the space and y-axis being the time. For any BST algorithm A executing on X, let $t_A(i)$ be all the nodes touched when we execute **search**(q_i), and we can know that the cost of this query is $\Theta(|t_A(i)|)$. Then define the set of points $S_X^A = \{(x, i) \mid x \in t_A(i)\}$. Note that we must have $S_X \subseteq S_X^A$ due to the correctness of algorithm A. Then the running time of A satisfies $R_A(X) = \Theta(|S_X^A|)$.

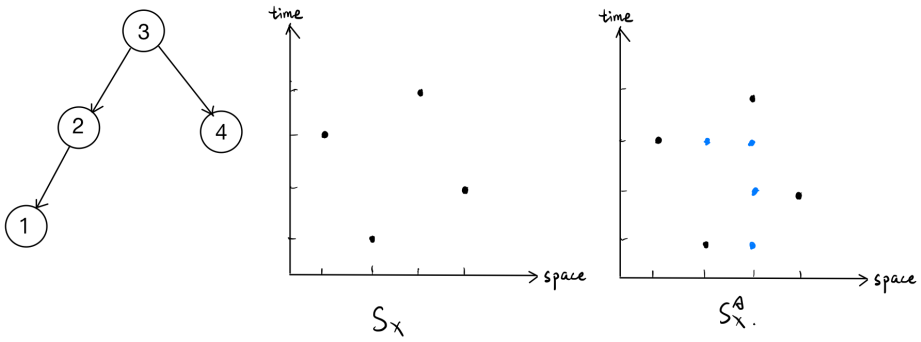

Figure 1: An example of S_X and S_X^A with $X = 2, 4, 1, 3$ and A be the corresponding algorithm of the static BST shown on the left.

Now we try to find a BST algorithm on X by directly finding a superset of S_X. We will see that $S_X \subseteq S$ corresponds to a BST algorithm iff S is **arborally satisfied**. The definition is as follows:

Definition 8 (arborally satisfied) *For any point set S, we say that S is arborally satisfied (AS for simplicity) if for all pair of points $p, q \in S$ that do not share the same x or y coordinate, there is a third point $r \in S$ inside or on the boundary of the rectangle spanned by p, q (i.e., the rectangle with pq as the diagonal and with edges parallel to the axis).*

With this definition, we have the following theorem:

Theorem 9 *$S = S_X^A$ for some BST algorithm A iff S is an arborally satisfied superset (ASS for simplicity) of S_X.*

This theorem directly implies that $OPT(X) = \Theta(minASS(X))$. Now we have changed the problem of finding the optimal BST over X into the problem of finding $minASS(X)$, which seems easier to study.

3.2 Greedy algorithm

The geometric view of BSTs tells us that if we can find an $O(1)$ approximation online arborally satisfied algorithm, then we have a BST algorithm with dynamic optimality. Note that we have a simple greedy algorithm for ASS: consider all points in S_X row by row, then add any points into the current row that would be necessary to make the current subset of S_X arborally satisfied. This algorithm is conjectured to be dynamically optimal in Demaine et al. (2009).

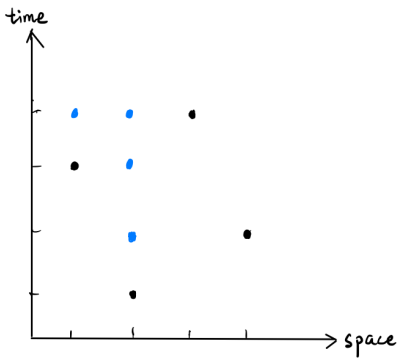

Figure 2: Result of the greedy algorithm on $X = 2, 4, 1, 3$.

4. Lower bounds for $OPT(X)$

There are also non-trivial lower bounds on $OPT(X)$ for a given access sequence X. In this section, we present some of these lower bounds and their applications.

4.1 Independent rectangle bounds

The independent rectangle bound by Demaine et al. (2009) is the best known bound on $OPT(X)$. The basic idea is to count the maximum number of independent rectangles in S_X.

Definition 10 (independent rectangles) *A pair of rectangles spanned by ab and cd are called independent if the rectangles*

1. *are not arborally satisfied;*
2. *no corner of either rectangle is strictly inside of the other rectangle.*

We say that a set of rectangles are independent if any two rectangles in this set pairwisely independent.

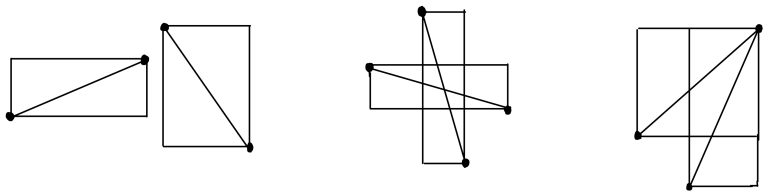

Figure 3: Examples of independent rectangles

Let $IRB(S)$ be the size of the maximum set of independent rectangles on S. We have $OPT(X) = \Omega(IRB(S_X))$ due to the following theorem:

Theorem 11 *Given a set of points S, we have $minASS(S) \geq |S| + \frac{1}{2}IRB(S)$.*

Proof According to Lemma 12 that we will show in 4.1.1, we have

$$\begin{aligned} minASS(S) &\geq \max\{minASS^+(S), minASS^-(S)\} \\ &\geq |S| + \max\{IRB^+(S), IRB^-(S)\} \\ &\geq |S| + \frac{1}{2}(IRB^+(S) + IRB^-(S)) \\ &\geq |S| + \frac{1}{2}IRB(S). \end{aligned}$$

■

In the following subsection, we will show that $IRB(S)$ can be estimated within a constant factor by a simple greedy algorithm.

4.1.1 Signed greedy algorithm

Note that the rectangles spanned by two points ab can be separated into two types according to whether the diagonal ab looks like "/" or "\". More precisely, a rectangle ab (suppose that $a.x < b.x$) belongs to one of the following two types:

1. +type: if $a.y < b.y$;
2. −type: if $a.y > b.y$.

Define $IRB^+(S)$ ($IRB^-(S)$) as the size of the maximum set of independet +type (−type) rectangles on the point set S. Similarly we can define $minASS^+(S)$ and $minASS^-(S)$ as the minimum arborally satisfied superset considering only one type of rectangles. Since +type and −type are wholly symmetric, we only consider +type in the following discussion.

We will see that $IRB^+(S)$ and $minASS^+(S)$ can be calculated easily by the signed greedy algorithm, which is just a variation of the greedy algorithm in section 3.2 that considers only +type rectangles:

First, we have the following lemma:

Lemma 12 $minASS^+(S) \geq |S| + IRB^+(S)$.

Proof The geometric proof is not very related to our topic, so we just omit it. Details of the proof is shown in Demaine et al. (2009). ■

Then, we can show that newly added points in the signed greedy algorithm correspond to a set of independent +type rectangles (see Demaine et al. (2009)). Then let $Greedy^+(S)$ be the result of the signed greedy algorithm, and we can get

$$Greedy^+(S) \geq minASS^+(S) \geq |S| + IRB^+(S) \geq Greedy^+(S).$$

This leads to the result that $Greedy^+(S) = minASS^+(S)$. Note that calculating $minASS(S)$ is shown to be NP-complete by Demaine et al. (2009), but calculating $minASS^+(S)$ and $minASS^-(S)$ is surprisingly easy. With the signed greedy algorithm, we can efficiently estimate $IRB(S) = \Theta(IRB^+(S) + IRB^-(S))$.

4.2 Wilber's first lower bound [Wilber (1989)]

Wilber's first bound (or alternation bound) gives a lower bound for $OPT(X)$ by counting the number of alternations on a static BST. For a given access sequence X and an arbitrary static BST T, let $ALT(u)$ ($u \in T$) be the number of alternations between accesses of u's left and right subtree on the subsequence of X without y and elements outside y. Then we have

$$OPT(X) = \Omega(\sum_{u \in T} ALT(u)).$$

We can prove this bound from the geometric view, by showing that all these alternations correspond to a set of independent rectangles.

4.2.1 Example: bit-reversal sequence

Suppose that $n = 2^k$ and the elements are $0, 1, \ldots, n-1$. Now consider the bit-reversal sequence, i.e., reverse the digits (in binary) of each number in the sequence $0, 1, 2, \ldots, 2^k$. For example, the sequence is

$$0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15$$

for $k = 4$. We will show that any BST requires $\Omega(n \log n)$ time on this kind of access sequence.

Consider the perfect binary sarch tree. For any node, the bit-reversal sequence visits its two subtrees alternatively. So each node reaches the maximum number of possible alternations. According to Wilber's first lower bound, any BST algorithm requires $\Omega(n \log n)$ time, and thus we get a tight bound $\Theta(n \log n)$ for the bit-reversal sequence.

4.2.2 Application: Tango tree

Tango tree (Demaine et al. (2007)) is a online BST algorithm that runs in a $O(\log \log n)$ factor within Wilber's first lower bound, and thus it is $O(\log \log n)$ competitive to the offline optimal BST. Tango tree is the first known BST that has a nontrivial $O(\log \log n)$ approximation factor, and is the BST closest to dynamic optimality now.

Tango tree maintains a structure that is similar to link-cut trees (LCT). The algorithm starts with a balanced BST with depth $\lfloor \log n \rfloor$, maintains the preferred child of each node, and decomposes the BST into several paths according to the preferred child. Similarly as a LCT, we maintain each path by a self-adjusting BST (e.g., Splay). The key idea of Tango tree is that every time Wilber's first lower bound increases by 1 (i.e., an alternation happens), we change the preferred child of the corresponding node with a cost $O(\log \log n)$ (since the size of each path is $O(\log n)$, BST operations require only $O(\log \log n)$). Therefore, we get an $O(\log \log n)$ approximation. This structure can be maintained in a BST model by recording extra information at the positions we change from one path to another.

Note that Tango tree also implies that Wilber's first lower bound is close to $OPT(X)$ within a factor of $O(\log \log n)$.

Acknowledgement

Great thanks to the MIT 6.851 Lecture 5& 6 for introducing the big picture of dynamic optimality. Also thanks Weijuan Dong's IOI-2018 national training team paper for the introduction of finger search.

References

Richard Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

Erik D Demaine, Dion Harmon, John Iacono, and Mihai P a ˇ traşcu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.

Erik D Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Patraşcu. The geometry of binary search trees. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 496–505. SIAM, 2009.

Jonathan C Derryberry and Daniel D Sleator. Skip-splay: Toward achieving the unified bound in the bst model. In *Workshop on Algorithms and Data Structures*, pages 194–205. Springer, 2009.

John Iacono. Alternatives to splay trees with o (log n) worst-case access times. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 516–522, 2001.

John Iacono. In pursuit of the dynamic optimality conjecture. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 236–250. Springer, 2013.

Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM journal on Computing*, 18(1):56–67, 1989.