Lecture 9 — March 12, 2003

Prof. Erik Demaine *Scribe: A. Andoni, C. Cadar*

1 Overview

In the last lecture we discussed the unified property and the unified structure. We started to prove that the unified structure has the unified property, by describing how the search and the add operations work.

In this lecture we will re-iterate the main properties of the unified structure and we will finish analyzing the add operation in this data structure. Next, we will cover key-independent optimality, Wilbur's second lower bound, and Munro's offline binary search tree.

2 Unified Structure

We begin by re-iterating the main characteristics of the unified structure.

2.1 Description

The unified structure is a collection of k trees and k queues, where $k \sim \lg\lg n$ and n is the number of elements in the data structure. The trees are doubly exponentially increasing in size, except for the last tree which contains all n elements. Each queue stores the "childless" nodes of the corresponding tree. A graphical representation of the unified structure can be found in Figure 1.

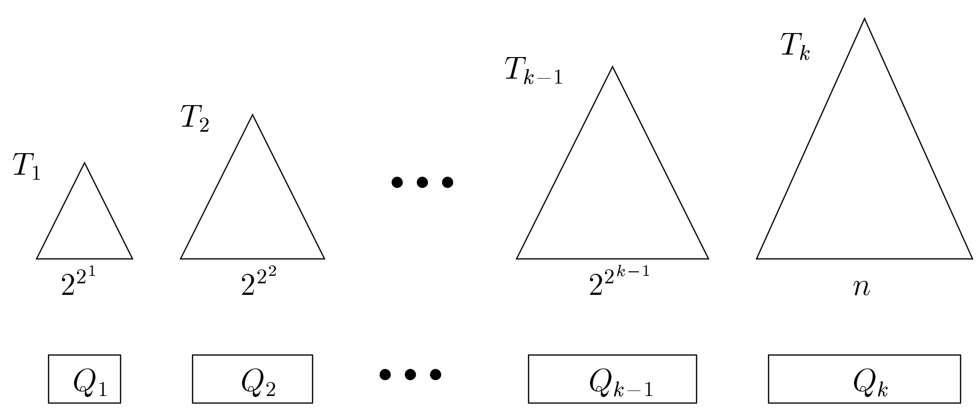

Figure 1: The Unified Structure.

2.2 Search Operation

The search operation, already discussed during the last lecture, is the following:

Search(x)

- search in $T_1, T_2, \ldots$ until we find some node y in T_l within $2 \cdot 2^{2^l}$ of x;
- add x to T_l with the same parent as y;
- add x to T_{l-1} with the same parent as x in T_l;
- $\ldots$
- add x to T_l with the same parent as x in T_2.

We saw how to do the first operation in $O(2^l)$ time.

2.3 Add Operation

The add operation requires adding an element x to the tree T_i with the same parent as some other element y. We distinguish here two main cases: when x is within $2^{2^{i+1}}$ of the parent of y, and when x is at a bigger distance from the parent of y.

The first case, when x is within $2^{2^{i+1}}$ of the parent of y, can be easily treated in the following way:

Add(x) to T_i with the same parent as y

- insert x into T_i and enqueue it into Q_i;
- link x to $parent(y)$ and maybe remove $parent(y)$ from Q_{i+1};
- dequeue z from Q_i and delete it from T_i;
- unlink z from $parent(z)$, maybe enqueue $parent(z)$ into Q_{i+1}.

The second case, when x is not within $2^{2^{i+1}}$ of the parent of y, requires more work. In this case, x is within 2^{2^i} of y, which in turn is within $2^{2^{i+1}}$ of $parent(y)$. This situation is shown graphically in Figure 2.

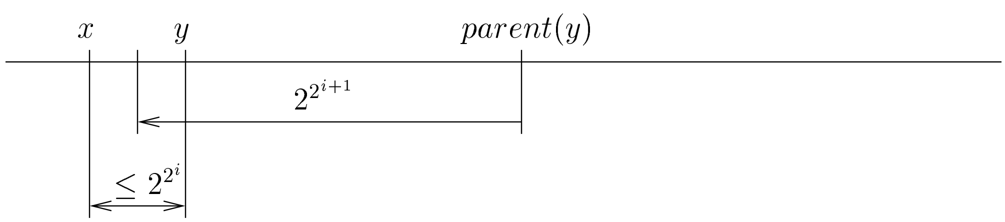

Figure 2: x is not within $2^{2^{i+1}}$ of the parent of y.

Next, we remark that every node in the unified structure in tree T_{i+1} stores its children in the four quarters of size $2^{2^{i+1}}/2$, as shown in Figure 3 for the parent of y.

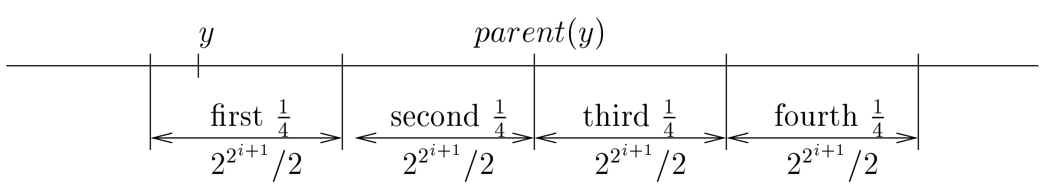

Figure 3: The four quarters of the children of the node $parent(y)$.

With these observation, we can describe how the add operation works when x is not within $2^{2^{i+1}}$ of the parent of y:

<u>`Add(x) to` T_i `with the same parent as` y</u>

- `add` $z = parent(y) - 2^{2^{i+1}}$ `to` T_{i+1} `with the same parent as` $parent(y)$`;`
- `move the first quarter of` $parent(y)$`'s children to third quarter of` z`'s children;`
- `remove` z `from` Q_{i+1} `and maybe enqueue` $parent(y)$ `into` Q_{i+1}`;`
- `add` x `to` T_i `with parent` z`.`

2.4 Analysis of Add Operation

We would like to prove that the add operation works in $O(2^l)$ amortized time. In order to prove this, we define the following potential function:

$$\Phi = c \cdot E,$$

where c is a constant and E is the number of extreme children, i.e., the number of children in the first or last quarter of the nodes in the structure.

Again, we divide our analysis in two cases. The first case, in which x is within $2^{2^{i+1}}$ of the parent of y, is very simple to analyze. In this case, the add operation takes $O(2^l)$ time (the time to add a node into a tree of size 2^{2^l}). As for the potential increase, $O(1)$ potential is added per level, and so there is a total $O(l)$ potential increase. Thus, the amortized time complexity of the search operation is $O(2^l)$.

The analysis of the second case, when x is not within $2^{2^{i+1}}$ of the parent of y, is a little bit more complicated. First of all, let's assume that $x < parent(y)$. The other case, when $x > parent(y)$, can be treated symmetrically.

First, let's analyze the change of potential, by following the evolution of z's children. When z is first added, z (from a smaller tree) is the only child. Next, each new child is added within $2 \cdot 2^{2^i}$ of the previous child. So, in order to get a new child at distance greater than $2^{2^{i+1}}$ of z, we must have added at least $2^{2^{i+1}} \; / \; (2 \cdot 2^{2^i}) = 2^{2^i}/2$ children before this child, and thus at least $2^{2^i}/4$ children in the first quarter of z.

Because all the children of z in the first quarter are going into the third quarter of some other node, we can conclude that the potential loss is at least $c \cdot 2^{2^i}/4$. On the other hand, the cost for the recursive call to operation add is $O(2^{i+1})$, which is dominated for c sufficiently large. The amortized cost is thus at most zero.

Detail: The n elements of the unified structure are kept in sorted order in an array, and each node in a tree keeps the index in this array. Alternatively, the array can be replaced by a finger search tree, which is a level-linked 2-3-tree, as shown in Figure 4.

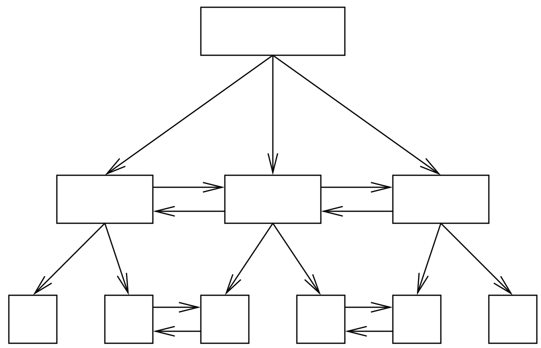

Figure 4: Level-linked 2-3-tree.

3 Key-Independent Optimality

The concept of key-independent optimality was developed in [1].

The main assumption here is that the keys are "meaningless". We can understand this concept as taking a uniformly random permutation π of the n keys. With this concept in mind, we define the key-independent dynamic optimal as being the expected value of dynamic optimum on a randomly permutated sequence, E[dynamic OPT(π(sequence))].

Theorem: Let $s = x_1, x_2, \ldots, x_m$ be the request sequence. Then, key-independent OPT(s) is $\Theta(\sum_{i=1}^{m} \lg w_i)$, where w_i is the number of distinct elements accessed since the last access to x_i, also called the working-set number.

Example: Splay trees, the working-set structure, and the unified structure all have this property. Splay trees are truly key-independent optimal, because they operate in the search-tree model.

We won't give the proof of this theorem in lecture, but we will describe Wilbur's (second) lower bound, which is used in the proof.

4 Wilbur's (second) lower bound

Wilbur's second lower bound was formulated in [3].

Let $x_1, x_2, \ldots, x_m$ be the request sequence and consider the amortized cost of x_j. For this, look at where x_j fits among the subsequence $x_i, x_{i+1}, \ldots, x_{j-1}$, for all i from $j-1$ down to 1. For each i, let's say x_j fits between a_i and b_i. Then, as i decreases, a_i will increase and b_i will decrease. Wilbur's (second) lower bound says that the amortized cost of x_j is at least the number of alternations between a_i increasing and b_j decreasing.

Example: $n = 9$, $m = 9$
Let the sequence be 3 7 1 9 5 2 8 4 6.
Then, for $j = 9$, we have the sequence of alternations shown in Figure 5.

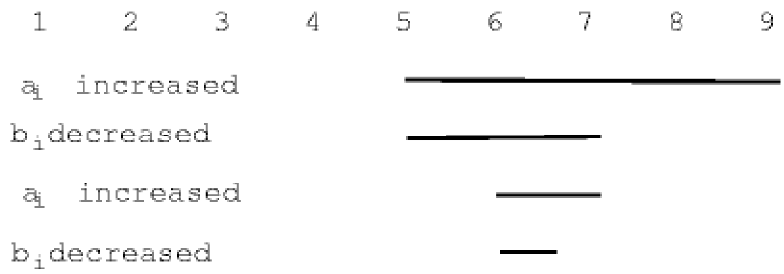

Figure 5

This lower bound holds for any data structure in the search-tree model.

OPEN: Is dynamic OPT $= O($ Wilbur's lower bound)?

OPEN: How do the dynamic OPT, Wilbur's lower bound, and splay trees relate?

5 Munro's Offline Binary Search Trees

Munro's offline binary search trees were presented in [2].

In Munro's offline binary search trees, we perform the following additional operations. Upon search for an element, we look at the path from the root to the element as shown in Figure 6. Then, we re-arrange this structure as a binary search tree, without entering the side subtrees, to minimize the depth of the next-requested node.

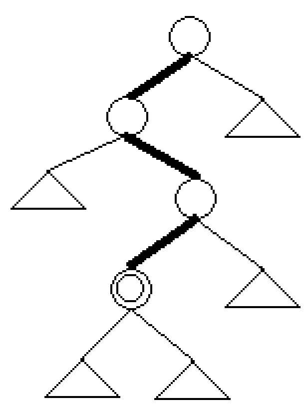

Figure 6

We consider two cases. If the next-requested node is on the path, then we put it at root. Otherwise, if the next-requested node is in a subtree, we have one of the two subcases shown in Figure 7. In this situation, if the next-requested node is in the minimum or the maximum subtree, we make it the child of the root.

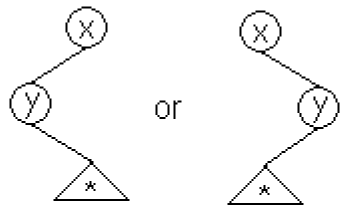

Figure 7

Next, subject to this constraint (minimizing the depth of the next-requested node), we try to minimize the depth of the second-next-requested node. In particular, if this element is less than or equal to x, then we put x at root. If however this element is greater than or equal to y, then we put y at the root. Otherwise, if the element is between x and y, we do nothing.

Next, subject to these first two constraints, we try to minimize the depth of the third-next-requested node, then the depth of the fourth-next-requested node, and so on.

OPEN: Is Munro's method $= O(1) \times$ dynamic OPT?

OPEN: Is Munro's method $\geq O(1) +$ dynamic OPT?

OPEN: Are splay trees $= O$(Munro's method)?

References

[1] John Iacono, "Key independent optimality", *Proceedings of the 13th Annual International Symposium on Algorithms and Computation*, LNCS 2518, November 2002, pages 211–218.

[2] J. Ian Munro, "On the Competitiveness of Linear Search", *Proceedings of the 8th European Symposium on Algorithms*, LNCS 1879, September 2000, pages 338–345.

[3] Robert E. Wilbur, "Lower Bounds for Accessing Binary Search Trees with Rotations", *SIAM Journal on Computing*, 18(1):56–67, February 1989.