

HAIRY SEARCH TREES

C.H.A. Koster and Th.P. van der Weide

Department of Information Systems, University of Nijmegen
 Toernooiveld, NL-6525 ED Nijmegen, The Netherlands
 {kees,tvdw}@cs.kun.nl

Abstract

Random search trees have the property that their depth depends on the order in which they are built. They have to be *balanced* in order to obtain a more efficient storage-and-retrieval datastructure. Balancing a search tree is time consuming. This explains the popularity of datastructures which approximate a balanced tree but have lower amortised balancing costs, such as AVL trees, Fibonacci trees and 2-3 trees. The algorithms for maintaining these datastructures efficiently are very complex and hard to derive. This complexity follows from the fact that even partial balancing upon insertion needs extensive rearrangement of the tree, not confined to the path from the root of the tree to the place of insertion. This observation led us to consider insertion algorithms that perform *local balancing* around the newly inserted node, without backtracking on the search path.

In this note, we investigate the implementation and properties of hairy search trees.

Keywords: search trees, heuristic balancing, local balancing, hairy trees.

1 Hairy Trees

Hairy trees are a class of binary trees with the property:

$$\text{is hairy } (t) \equiv \forall_{\text{node } v \in \tau} [v \text{ has single son } s \Rightarrow s \text{ is leaf}]$$

The intuition behind this condition is that it prevents trees from having list-like substructures longer than two nodes (“bare twigs”). Some examples of hairy trees are presented in figure 1.

The class of hairy trees can be described by the following recursive definition:

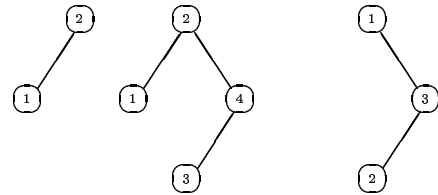


Figure 1: Two hairy trees and a non-hairy one

1. is hairy (ε)
2. If t is a singleton tree and x some key value, then is hairy(t), is hairy(Tree:(x, t, ε)) and is hairy(Tree:(x, ε, t)).
3. If $t_1 \neq \varepsilon$, $t_2 \neq \varepsilon$ and x some key value, then is hairy (Tree: (x, t_1, t_2)).

This inductive definition gives us the opportunity to use structural induction in reasoning about hairy trees.

definition 1 *The function single counts the number of single-son nodes in a tree:*

$$\begin{aligned} \text{single } (\varepsilon) &= 0 \\ \text{single } (\text{Tree: } (x, t_1, t_2)) &= \begin{cases} 0 & \text{if } t_1 = \varepsilon \wedge t_2 = \varepsilon \\ 1 & \text{if } t_1 = \varepsilon \vee t_2 = \varepsilon \\ \text{single } (t_1) + \text{single } (t_2) & \text{otherwise} \end{cases} \end{aligned}$$

The following property is easily proved by structural induction,;

lemma 1

$$\text{is hairy } (t) \Rightarrow 0 \leq \text{single } (t) \leq \text{leaves } (t)$$

where $\text{single } (t)$ is the number of single-son nodes and $\text{leaves } (t)$ the number of leaves on t and $\text{nkeys } (t)$ the number of keys in t . Furthermore let $\text{ext } (t)$ the number of external nodes of tree t .

lemma 2

$$\text{nkeys } (t) + 1 = \text{ext } (t) = \text{single } (t) + 2 \times \text{leaves } (t)$$

Our goal in introducing hairy trees is to reduce the ratio between the number of single-son nodes and the number of external nodes in a search tree. This ratio will be denoted as $\Delta(t)$.

lemma 3

$$\text{is hairy } (t) \Rightarrow 0 \leq \Delta(t) \leq \frac{1}{3}$$

Both bounds are sharp. This lemma is easily proved using the two previous lemma's.

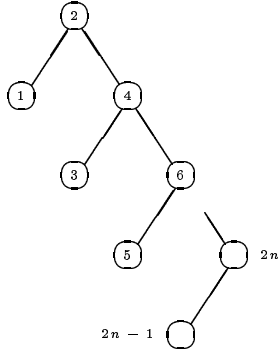


Figure 2: The worst hairy tree

In general, hairy trees are not balanced. In the worst case, a hairy tree of n elements has depth $\lceil \frac{n+1}{2} \rceil$ (see figure 2).

2 Insertion in hairy search trees

We define a new operation `inserth`¹ for inserting a key into a search tree, which maintains the search tree as a hairy tree by restructuring it whenever a node is to be inserted at the end of a twig. Its structure follows the case-distinction in the definition of `is hairy`.

¹The programming language used is Elan[KOS87], an educational algorithmic language.

```

PROC inserth (TREE VAR t, EL CONST e):
  { is search tree (t), is hairy (t) }
  IF is empty (t)
  THEN t := tree (e)
  ELIF e < t.key
  THEN
    IF is empty (t.left)
    THEN replace left
    ELIF is empty (t.right)
    THEN
      IF e < t.left.key
      THEN insert left left
      ELIF t.left.key < e
      THEN insert left right
      FI
    ELSE inserth (t.left, e)
    FI
  ELIF t.key < e
  THEN
    IF is empty (t.right)
    THEN replace right
    ELIF is empty (t.left)
    THEN
      IF e < t.right.key
      THEN insert right left
      ELIF t.right.key < e
      THEN insert right right
      FI
    ELSE inserth (t.right, e)
    FI
  FI
  {is search tree (t), is hairy (t),
  is in (e, t)}
ENDPROC inserth;

```

with the refinements:

```

replace left:
  t.left := tree (e).

insert left left:
  t.left.right := t;
  t := t.left;
  t.right.left := empty;
  t.left := tree (e).

insert left right:
  TREE CONST x :: t.left;
  t.left := empty;
  t := tree (e, x, t).

replace right:
  t.right := tree (e).

insert right right:
  t.right.left := t;
  t := t.right;
  t.left.right := empty;
  t.right := tree (e).

```

```

insert right left:
TREE CONST z :: t.right;
t.right := empty;
t := tree (e, t, z).

```

The implementation can be further optimized by unfolding, specialization and elimination of the recursion. Its correctness is easy to prove, since it closely follows the inductive structure of the definition of is hairy .

3 Efficiency of Hairy Trees

We analyse the efficiency of hairy trees in terms of the cost of a random successful search (S_n) in a tree with n keys, and the cost of a random unsuccessful search (U_n). Let I_n be the average internal path length of all hairy trees with n keys, so $S_n = \frac{1}{n}I_n$. Then we have:

$$I_{n+1} = I_n + (U_n + 1) - 2\Delta_n \quad (1)$$

as obviously the internal path length is augmented with $U_n + 1$ by the insertion of a new key, and occasionally diminished by a restructuring. A restructuring is performed if and only if we start from (up to symmetry) the situation of figure 3, which is transformed by insertion of a node at its end into the one of the cases in figure 4. After restructuring we have figure 5.

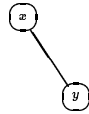


Figure 3: Addition via single-son node

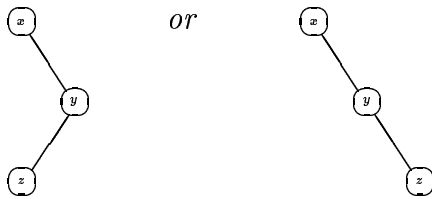


Figure 4: After addition

In both cases the internal path length decreases by 1 as a result of restructuring. The probability of this situation to occur in tree t is:

$$\frac{2 \times \text{single}(t)}{\text{ext}(t)} = 2\Delta(t)$$

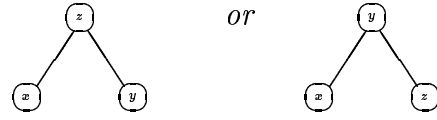


Figure 5: After restructuring

From equation (1) we derive:

$$I_n = \sum_{k=0}^{n-1} (U_k + 1 - 2\Delta_k) \quad (2)$$

The following relation is well known:

$$S_n = (1 + \frac{1}{n}U_n) - 1$$

and can be rewritten as

$$I_n = (n + 1)U_n - n \quad (3)$$

Combining (2) and (3) yields

$$(n + 1)U_n = \sum_{k=0}^{n-1} (U_n + 2 - 2\Delta_k)$$

This is transformed into a recurrence relation by computing $(n + 1)U_n - nU_{n-1} = U_{n-1} + 2 - 2\Delta_{n-1}$, leading to:

$$U_n - U_{n-1} = \frac{2}{n + 1}(1 - \Delta_{n-1})$$

and thus:

$$U_n = 2 \sum_{k=0}^{n-1} \frac{1}{k + 2}(1 - \Delta_k)$$

Next we consider Δ_n . Let σ_n be the average number of single-son nodes in a hairy tree. When a new node is inserted via a search path through a single-son node, the number of single-son nodes will be decremented by 1. In the other case, this number will be incremented by 1. This leads to the following recurrence relation:

$$\begin{aligned} \sigma_{n+1} &= \sigma_n - \frac{3\sigma_n}{n + 1} + \frac{(n + 1) - 3\sigma_n}{n + 1} \\ &= \frac{n - 5}{n + 1}\sigma_n + 1 \end{aligned}$$

From this recurrence relation we derive $\sigma_6 = 1$, and therefore $\sigma_n = (n + 1)/7$ for $n > 6$. As $\Delta_n = \sigma_n/(n + 1)$, we conclude:

$$\Delta_n = \frac{1}{7} \quad \text{for } n > 6$$

| | random search tree | hairy tree | AVL tree | balanced tree |
|---------------------|---------------------|---------------------|---------------------|---------------|
| expected searchtime | $1.386..^2 \log(n)$ | $1.188..^2 \log(n)$ | $1.012..^2 \log(n)$ | $^2 \log(n)$ |
| worst case depth | n | $\frac{n+1}{2}$ | $1.440..^2 \log(n)$ | $^2 \log(n)$ |

Table 1: Comparing methods

lemma 4

$$U_n = \frac{6}{7} U_n^\sim \approx 1.1883...^2 \log n$$

where $U_n^\sim = 2 \sum_{k=0}^{n-1} \frac{1}{k+2} \approx 1.3863...^2 \log n - 0.8456...$ is the average cost of an unsuccessful search in a random binary tree. The result of this analysis is summarized in table 1 (see [KNU73, ?]).

4 Conclusions

In this note, we have investigated the implementation and properties of hairy search trees, a locally balanced form of search trees.

The efficiency of hairy search trees is about halfway between that of random search trees and AVL trees. Although their worst-case behaviour is only a factor of two better than that of random search trees, the expected search and insertion complexity of hairy search trees is nearer to that of AVL trees. By contrast, their implementation is much simpler than that of AVL trees, and hardly more complicated than that of conventional random search trees.

Therefore we propose the use of hairy search trees instead of random search trees, as a poor man's approximation to AVL trees in applications where insertion and search efficiency are of some importance.

References

- [KNU73] D.E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms. Addison-Wesley, 1973.
- [KOS87] C.H.A. Koster, Top-Down Programming wit Elan, Ellis Horwood, 1987.
- [GON83] G.H. Gonnet, Handbook of Algorithms and Data Structures, International Computer Science Services, 1983.