2-23-2005

Alternatives to Two Classic Data Structures

Chris Okasaki
United States Military Academy, chris.okasaki@westpoint.edu

Recommended Citation

Chris Okasaki. 2005. Alternatives to two classic data structures. In Proceedings of the 36th SIGCSE technical symposium on Computer science education (SIGCSE '05). Association for Computing Machinery, New York, NY, USA, 162–165. DOI:https://doi.org/10.1145/1047344.1047407

Alternatives to Two Classic Data Structures

Chris Okasaki*
United States Military Academy
West Point, NY
Christopher.Okasaki@usma.edu

ABSTRACT

Red-black trees and leftist heaps are classic data structures that are commonly taught in Data Structures (CS2) and/or Algorithms (CS7) courses. This paper describes alternatives to these two data structures that may offer pedagogical advantages for typical students.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees

General Terms

Algorithms

Keywords

Red-black trees, maxiphobic heaps, leftist heaps

1. INTRODUCTION

The field of computer science changes so rapidly that few of the topics we teach deserve the appellation "classic". Two common data structures that fall into that category are *red-black trees* [5] and *leftist heaps* [6], both developed in the 1970's. Classic data structures provide a welcome sense of history to the computer science classroom, but, because such data structures were rarely devised with an eye toward pedagogy, we should continue to look for alternatives that may be pedagogically superior. This paper describes two such alternatives.

The first alternative is not a new data structure per se, but rather an alternative approach to inserting an element into a red-black tree. Our algorithm is simpler to understand and dramatically simpler to code than the usual algorithm found in textbooks [3, 9]. The second alternative is a new data structure similar to leftist heaps. This new data structure, called *maxiphobic heaps*, is simpler to design than leftist heaps and hopefully offers greater insight into the process of designing a non-trivial data structure. Both alternatives are suitable for use in either Data Structures (CS2) or Algorithms (CS7).

Our insertion algorithm for red-black trees has previously been descibed in [7]. Maxiphobic heaps have previously been described in [8]. However, both data structures were devised and described in the context of functional programming languages. Unfortunately, few Data Structures or Algorithms courses are taught in functional programming languages. This paper adapts these data structures to an imperative pseudocode that is compatible with all the major imperative and object-oriented languages commonly used in the classroom today, and explicitly considers their potential pedagogical benefits.

2. RED-BLACK TREES

Red-black trees and AVL trees [1] are probably the two most widely taught forms of balanced binary search trees. The algorithms and presentation of both are similar, involving left single rotations, left double rotations, right single rotations, and right double rotations. Although easy to understand at a superficial level, both kinds of trees are extremely difficult for a beginner to implement. We describe an alternative approach to insertion in a red-black tree that replaces the four kinds of rotations with a single balancing transformation, dramatically reducing the amount of code needed to implement the insertion function.

Besides the usual search-tree ordering, red-black trees obey several invariants related to the color of the nodes. Every node is colored either red or black in such a way that

1. every red node has a black parent, and
2. every path from the root to a node with one or two empty children contains the same number of black nodes.

The trick is to maintain these color invariants when modifying the tree.

To insert an element into a red-black tree, we create a new node containing the element and attach it to the bottom of the tree in the appropriate location to maintain the search-tree ordering, exactly as if we were inserting into an unbalanced binary search tree.

*This work was supported, in part, by the National Science Foundation under grant CCR-0098288. The views expressed in this paper are those of the author and do not reflect the official policy or position of the United States Military Academy, the Department of the Army, the Department of Defense, or the U.S. Government.

SIGCSE'05, February 23–27, 2005, St. Louis, Missouri, USA.
ACM 1-58113-997-7/11/0002.

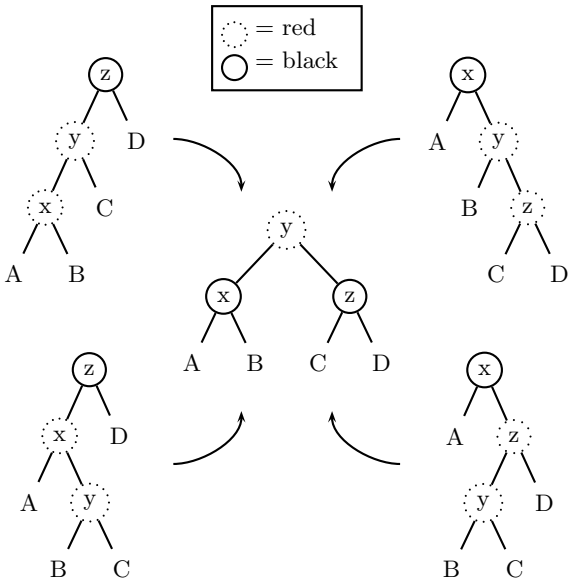

Figure 1: The four cases for balancing a red-black tree.

Next we color the new node. If we color it red, we risk violating the first color invariant. If we color it black, we risk violating the second color invariant. However, note that the first invariant is a local property and the second invariant is a global property. In the hope that a local property will be easier to fix than a global property, we color the new node red.

If the parent of the new node is also red, then we have violated the first color invariant and need to rearrange and recolor the tree to restore the invariant. This is where rotations are used in ordinary red-black trees. Instead of rotations we use the following balancing transformation:

> Take the red child, the red parent, and the (black) grandparent and locally balance these three nodes by making the smallest and largest nodes children of the middle node. Then color the middle node red and the other two nodes black. The middle node is linked back into the tree in place of the former black grandparent.

This balancing transformation is illustrated in Figure 1. Students find this picture satisfying because all four cases *look* like they are making the tree more balanced, whereas the usual pictures of left and right single rotations do not look like they are making any progress.

After the balancing transformation, we might still have a violation of the first color invariant, because the middle node, which is now red, might again have a red parent. However, note that the red-red violation is now closer to the root. By repeating the balancing transformation, we will eventually eliminate the red-red violation or reach the root. (It is worth exploring with students the consequences of trying the opposite color scheme in the balancing transformation, making the middle node black and the other two nodes red. What happens in that scheme is that the algorithm sometimes falls into an infinite loop, repeatedly rotating the same few nodes back and forth.)

```
-- a Tree node contains four fields:
--   Key, Color, Left, and Right

function Insert(K : Key, T : Tree) returns Tree is
   T := Ins(K, T)
   T.Color := BLACK  -- always recolor root black
   return T

function Ins(K : Key, T : Tree) returns Tree is
   if T = null then
      T := allocate a new Tree node
      T.Key := K
      T.Color := RED
      T.Left := null
      T.Right := null
   elseif K < T.Key then
      T.Left := ins(Key, T.Left)
   elseif K > T.Key then
      T.Right := ins(Key, T.Right)
   else return T  -- K is already in T

   -- check for red child and red grandchild
   if IsRed(T.Left) and IsRed(T.Left.Left) then
      T := Balance(T.Left.Left, T.Left, T,
                   T.Left.Left.Right, T.Left.Right)
   elseif IsRed(T.Left) and IsRed(T.Left.Right) then
      T := Balance(T.Left,T.Left.Right,T,
                   T.Left.Right.Left,T.Left.Right.Right)
   elseif IsRed(T.Right) and IsRed(T.Right.Left) then
      T := Balance(T, T.Right.Left, T.Right,
                   T.Right.Left.Left, T.Right.Left.Right)
   elseif IsRed(T.Right) and IsRed(T.Right.Right) then
      T := Balance(T, T.Right, T.Right.Right,
                   T.Right.Left, T.Right.Right.Left)
   return T

function Balance(X : Tree, Y : Tree, Z : Tree,
                 B : Tree, C : Tree) returns Tree is
   X.Right := B
   Y.Left  := X
   Y.Right := Z
   Z.Left  := C
   X.Color := BLACK
   Y.Color := RED
   Z.Color := BLACK
   return Y

function IsRed(T : Tree) returns Boolean is
   return T != null and T.Color = RED
```

Figure 2: Pseudocode for insertion into a red-black tree.

Notice that the first color invariant implies that the root must be black because it has no parent. If the root becomes red as a result of the balancing transformation, we simply color it black. In practice, it is easier to always color the root black than to check whether the root has become red. Pseudocode for the complete insertion algorithm is shown in Figure 2.

If you compare this code to a typical textbook presentation of red-black trees, such as [3], you will immediately be struck by how short this code is compared to an ordinary implementation. Most of this savings is from replacing the four separate kinds of rotations with a single transformation. In addition, ordinary implementations of red-black trees check whether the sibling of the red parent is also red. In such cases, these implementations recolor several of the nodes without rearranging them. Our implementation com-

pletely ignores the color of the sibling node, further reducing the amount of code required.

Although the resulting code is quite compact compared to other implementations, the density of the balancing code in the second half of the `Ins` function can be rather daunting at first glance. However, comparing the code side-by-side with the picture in Figure 1 makes it easy to sort out the `.Left`'s and `.Right`'s. Two more comments on this code are in order. First, the reason that we only check for red nodes in `Ins`, instead of also verifying that `T` is black, is that we are guaranteed that `T` is black if its child and grandchild are both red, because there will be at most one red node with a red parent at a time. Second, the reason we do not need to pass the A and D subtrees from Figure 1 into the `balance` function is that they are guaranteed to already be in the right places (to the left of X and the right of Z, respectively).

We omit the analysis showing that insertion into a red-black tree takes $O(\log N)$ time, because it is identical to the analysis for ordinary red-black trees, found in many textbooks [3, 9].

3. MAXIPHOBIC HEAPS

A common failing in the way we teach data structures and algorithms is presenting finished products rather than guiding students through the algorithmic design steps necessary to reach that product. For example, presentations of red-black trees (including our own!) commonly conjure the red-black invariants out of thin air. As a second example, consider leftist heaps [6]. Leftist heaps are simple to understand, simple to code, and simple to analyze, but students typically regard the leftist height invariant as "magic". They see how to proceed once they are given the height invariant, but do not see how they could have come up with the height invariant on their own. *Maxiphobic heaps* are an alternative to leftist heaps that retain the good qualities of leftist heap, but without the "magic".

Maxiphobic heaps, like leftist heaps, are a form of priority queue. Elements can be inserted into a heap, and the minimum element in a heap can be inspected or removed. In addition, two heaps can be merged into a single heap. Like leftist heaps, a maxiphobic heap is represented as a binary tree in *heap order*, meaning that the value at each node is never bigger than the values at any of its descendants. In addition, each node is annotated with the size of the subtree rooted at that node. Unlike leftist heaps, maxiphobic heaps place no restrictions on the shape of the binary tree—a maxiphobic heap can be arbitrarily unbalanced in any direction.

Except for merge, the operations on maxiphobic heaps are trivial. The minimum element in a heap-ordered tree is always the root, so to find the minimum element in a maxiphobic heap, we simply return the value at the root. To delete the minimum element, we simply delete the root and merge its two subtrees. To insert an element, we create a singleton tree containing the new element and merge it with the existing tree. Given the idea of heap-ordered trees and the existence of a merge function, students can readily come up with these algorithms on their own, although they may need a little coaxing to think of implementing insertion using merge.

Only the merge operation remains. To merge two maxiphobic heaps, we first compare their roots. The smaller root becomes the root of the combined tree. Now we reach the crucial step. Having decided on the new root, we next need to determine the two subtrees of the new root. However, we currently have three candidate subtrees vying for those two spots: the tree that lost the comparison of the roots and the two existing subtrees of the winning root. We must somehow reduce these three subtrees into two by (recursively) merging two of them together. But which two should we merge?

```
-- a Heap node contains four fields:
--   Value, Size, Left, and Right

function FindMin(H : Heap) returns Value is
   if H = null then error
   else return H.Value

function DeleteMin(H : Heap) returns Heap is
   if H = null then error
   else return Merge(H.Left, H.Right)

function Insert(V : Value, H : Heap) returns Heap is
   NewH := allocate a new Heap node
   NewH.Value := V
   NewH.Size  := 1
   NewH.Left  := null
   NewH.Right := null
   return Merge(H, NewH)

function Merge(H1 : Heap, H2 : Heap) returns Heap is
   if H1 = null then return H2
   if H2 = null then return H1

   -- force H1 to have smaller root
   if H2.Value < H1.Value then Swap(H1, H2)

   -- calculate size of merged tree
   H1.Size := Size(H1) + Size(H2)

   -- get the three subtrees
   A := H1.Left
   B := H1.Right
   C := H2

   -- force A to be biggest of the three subtrees
   if Size(B) > Size(A) then Swap(A,B)
   if Size(C) > Size(A) then Swap(A,C)

   -- rebuild tree
   H1.Left := A
   H1.Right := Merge(B, C)
   return H1

function Size(H : Heap) returns Integer is
   if H = null then return 0
   else return H.Size
```

Figure 3: Pseudocode for maxiphobic heaps.

Working through a few examples with trees of different sizes and shapes leads students to the following insight: we should always merge the two smallest of the three trees, leaving the largest of the three untouched (hence the name *maxiphobic*, meaning "biggest avoiding", although this name should probably not be introduced until after the students have had a chance to wrestle with the merge algorithm). A pseudocode implementation of maxiphobic heaps appears in Figure 3.

The key point here is that students can come up with this all-important design decision with only the gentlest of prodding from the instructor. They are left with a sense of em-

powerment, a feeling that they too are capable of *designing* data structures, rather than merely implementing somebody else's design. In comparison, the height invariant of leftist heaps is usually handed down by the instructor or textbook author without any insight into how or why Knuth came up with that invariant.[1] Students are passive spectators in the design process, rather than active participants.

After designing the merge algorithm with students, an easy analysis verifies that `merge` runs in $O(\log N)$ time, and therefore so do `insert` and `deleteMin`. For `merge`, N is the combined number of values in the two heaps being merged. Because the biggest of the three subtrees is avoided at each step, and that tree contains at least $(N-1)/3$ values, it is easy to see that

$$T(N) \leq T(2N/3) + O(1)$$

The solution to this recurrence relation is $O(\log_{3/2} N) = O(\log N)$.

Finally, note that "size" in maxiphobic heaps can be interpreted as either number of nodes or height of the tree. Either interpretation leads to a successful solution, so go with whichever one students come up with. The pseudocode in Figure 3 assumes a number-of-nodes interpretation, but changing to a height interpretation requires changing a single line of code (the size calculation of the merged tree in `merge`). Taking a number-of-nodes interpretation makes maxiphobic heaps similar to *weighted* leftist heaps [2], whereas a height interpretation makes them more similar to ordinary leftist heaps.

4. DISCUSSION

The data structures described in this paper offer several pedagogical advantages over their classical brethren. Our algorithm for insertion into red-black trees is significantly simpler than the usual insertion algorithms, so much so that implementing the insertion algorithm from scratch becomes a feasible task for a typical student. Our second data structure, maxiphobic heaps, offers greater insight into the design process than leftist heaps, without sacrificing simplicity in other areas. We believe that these alternatives could be seamlessly added to many instances of Data Structures (CS2) or Algorithms (CS7) courses, particularly those aimed at students of average or below average ability.

On the other hand, there may be some pedagogical disadvantages for courses aimed at highly talented students. Such a course might want to present deletion from red-black trees in addition to insertion. Our balancing transformation does not extend to deletion, so traditional rotation-based deletion algorithms would need to be used. If both insertion and deletion are to be presented, then it may be preferable to use traditional rotations for both.

For maxiphobic heaps, the elimination of the leftist height invariant may diminish the opportunity to impress upon top students the vital role of such invariants in the design of advanced data structures. If the design and use of such complex invariants is considered a significant objective, then leftist heaps may indeed be preferable, provided substantial attention is paid to the motivation and development of the height invariant.

[1]Ironically, the original data structure by Crane [4] upon which Knuth based leftist heaps, was more similar to maxiphobic heaps, except that it avoided the bigger of the two subtrees of the winning root, even if the losing tree was bigger still.

5. SOURCE CODE

Source code for red-black trees and maxiphobic heaps in a variety of languages is available on the World Wide Web at the author's web site:
`http://www.eecs.usma.edu/personnel/okasaki/sigcse05/`

6. ACKNOWLEDGEMENTS

Thanks to Jean Blair and John Hill for their feedback on an earlier draft of this paper.

7. REFERENCES

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, Sept. 1962. English translation of Russian orginal appearing in *Doklady Akademia Nauk SSSR*, 146:263-266.

[2] S. Cho and S. Sahni. Weight-biased leftist trees and modified skip lists. *ACM Journal of Experimental Algorithmics*, 1998. Article 2.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2001.

[4] C. A. Crane. *Linear lists and priority queues as balanced binary trees*. PhD thesis, Computer Science Department, Stanford University, Feb. 1972. Available as STAN-CS-72-259.

[5] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *IEEE Symposium on Foundations of Computer Science*, pages 8–21, Oct. 1978.

[6] D. E. Knuth. *Searching and Sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[7] C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[8] C. Okasaki. *Fun with binary heap trees*, pages 1–16. Palgrave MacMillan, 2003.

[9] M. A. Weiss. *Data Structures & Algorithm Analysis in Java*. Addison-Wesley, 1998.