Purely Functional Data Structures

Chris Okasaki

Book DOI: http://dx.doi.org/10.1017/CBO9780511530104

Online ISBN: 9780511530104

Hardback ISBN: 9780521631242

Paperback ISBN: 9780521663502

Chapter

3 - Some Familiar Data Structures in a Functional Setting pp. 17-30

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.004

Cambridge University Press

3

Some Familiar Data Structures in a Functional Setting

Although many imperative data structures are difficult or impossible to adapt to a functional setting, some can be adapted quite easily. In this chapter, we review three data structures that are commonly taught in an imperative setting. The first, leftist heaps, is quite simple in either setting, but the other two, binomial queues and red-black trees, have a reputation for being rather complicated because imperative implementations of these data structures often degenerate into nightmares of pointer manipulations. In contrast, functional implementations of these data structures abstract away from troublesome pointer manipulations and directly reflect the high-level ideas. A bonus of implementing these data structures functionally is that we get persistence for free.

3.1 Leftist Heaps

Sets and finite maps typically support efficient access to arbitrary elements. But sometimes we need efficient access only to the *minimum* element. A data structure supporting this kind of access is called a *priority queue* or a *heap*. To avoid confusion with FIFO queues, we use the latter name. Figure 3.1 presents a simple signature for heaps.

Remark In comparing the signature for heaps with the signature for sets (Figure 2.7), we see that in the former the ordering relation on elements is included in the signature while in the latter it is not. This discrepancy is because the ordering relation is crucial to the semantics of heaps but not to the semantics of sets. On the other hand, one could justifiably argue that an *equality* relation is crucial to the semantics of sets and should be included in the signature. ◇

Heaps are often implemented as *heap-ordered* trees, in which the element at each node is no larger than the elements at its children. Under this ordering, the minimum element in a tree is always at the root.

```
signature HEAP =
sig
  structure Elem : ORDERED

  type Heap

  val empty    : Heap
  val isEmpty  : Heap → bool

  val insert   : Elem.T × Heap → Heap
  val merge    : Heap × Heap → Heap

  val findMin  : Heap → Elem.T   (* raises EMPTY if heap is empty *)
  val deleteMin : Heap → Heap    (* raises EMPTY if heap is empty *)
end
```

Figure 3.1. Signature for heaps (priority queues).

Leftist heaps [Cra72, Knu73a] are heap-ordered binary trees that satisfy the *leftist property*: the rank of any left child is at least as large as the rank of its right sibling. The rank of a node is defined to be the length of its *right spine* (i.e., the rightmost path from the node in question to an empty node). A simple consequence of the leftist property is that the right spine of any node is always the shortest path to an empty node.

Exercise 3.1 Prove that the right spine of a leftist heap of size n contains at most $\lfloor \log(n+1) \rfloor$ elements. (All logarithms in this book are base 2 unless otherwise indicated.) ◇

Given some structure Elem of ordered elements, we represent leftist heaps as binary trees decorated with rank information.

```
datatype Heap = E | T of int × Elem.T × Heap × Heap
```

Note that the elements along the right spine of a leftist heap (in fact, along any path through a heap-ordered tree) are stored in sorted order. The key insight behind leftist heaps is that two heaps can be merged by merging their right spines as you would merge two sorted lists, and then swapping the children of nodes along this path as necessary to restore the leftist property. This can be implemented as follows:

```
fun merge (h, E) = h
  | merge (E, h) = h
  | merge (h1 as T (_, x, a1, b1), h2 as T (_, y, a2, b2)) =
      if Elem.leq (x, y) then makeT (x, a1, merge (b1, h2))
      else makeT (y, a2, merge (h1, b2))
```

where makeT is a helper function that calculates the rank of a T node and swaps its children if necessary.

```
fun rank E = 0
  | rank (T (r, _, _, _)) = r
fun makeT (x, a, b) = if rank a ≥ rank b then T (rank b + 1, x, a, b)
                      else T (rank a + 1, x, b, a)
```

Because the length of each right spine is at most logarithmic, merge runs in $O(\log n)$ time.

Now that we have an efficient merge function, the remaining functions are trivial: insert creates a new singleton tree and merges it with the existing heap, findMin returns the root element, and deleteMin discards the root element and merges its children.

```
fun insert (x, h) = merge (T (1, x, E, E), h)
fun findMin (T (_, x, a, b)) = x
fun deleteMin (T (_, x, a, b)) = merge (a, b)
```

Since merge takes $O(\log n)$ time, so do insert and deleteMin. findMin clearly runs in $O(1)$ time. The complete implementation of leftist heaps is given in Figure 3.2 as a functor that takes the structure of ordered elements as a parameter.

Remark To avoid cluttering our examples with minor details, we usually ignore error cases when presenting code fragments. For example, the above code fragments do not describe the behavior of findMin or deleteMin on empty heaps. We always include the error cases when presenting complete implementations, as in Figure 3.2.

Exercise 3.2 Define insert directly rather than via a call to merge.

Exercise 3.3 Implement a function fromList of type Elem.T list → Heap that produces a leftist heap from an unordered list of elements by first converting each element into a singleton heap and then merging the heaps until only one heap remains. Instead of merging the heaps in one right-to-left or left-to-right pass using foldr or foldl, merge the heaps in $\lceil \log n \rceil$ passes, where each pass merges adjacent pairs of heaps. Show that fromList takes only $O(n)$ time.

Exercise 3.4 (Cho and Sahni [CS96]) Weight-biased leftist heaps are an alternative to leftist heaps that replace the leftist property with the *weight-biased leftist property*: the size of any left child is at least as large as the size of its right sibling.

```
functor LeftistHeap (Element : ORDERED) : HEAP =
struct
  structure Elem = Element

  datatype Heap = E | T of int × Elem.T × Heap × Heap

  fun rank E = 0
    | rank (T (r, _, _, _)) = r
  fun makeT (x, a, b) = if rank a ≥ rank b then T (rank b + 1, x, a, b)
                        else T (rank a + 1, x, b, a)

  val empty = E
  fun isEmpty E = true | isEmpty _ = false

  fun merge (h, E) = h
    | merge (E, h) = h
    | merge (h1 as T (_, x, a1, b1), h2 as T (_, y, a2, b2)) =
        if Elem.leq (x, y) then makeT (x, a1, merge (b1, h2))
        else makeT (y, a2, merge (h1, b2))
  fun insert (x, h) = merge (T (1, x, E, E), h)

  fun findMin E = raise EMPTY
    | findMin (T (_, x, a, b)) = x
  fun deleteMin E = raise EMPTY
    | deleteMin (T (_, x, a, b)) = merge (a, b)
end
```

Figure 3.2. Leftist heaps.

(a) Prove that the right spine of a weight-biased leftist heap contains at most $\lfloor \log(n+1) \rfloor$ elements.

(b) Modify the implementation in Figure 3.2 to obtain weight-biased leftist heaps.

(c) Currently, merge operates in two passes: a top-down pass consisting of calls to merge, and a bottom-up pass consisting of calls to the helper function makeT. Modify merge for weight-biased leftist heaps to operate in a single, top-down pass.

(d) What advantages would the top-down version of merge have in a lazy environment? In a concurrent environment?

3.2 Binomial Heaps

Another common implementation of heaps is binomial queues [Vui78, Bro78], which we call *binomial heaps* to avoid confusion with FIFO queues. Binomial heaps are more complicated than leftist heaps, and at first appear to offer no compensatory advantages. However, in later chapters, we will see ways in

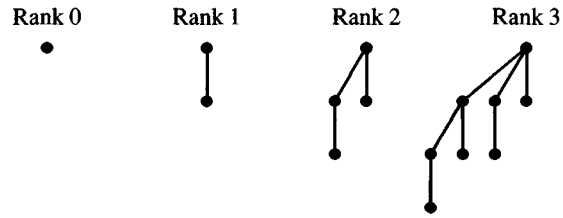

Figure 3.3. Binomial trees of ranks 0–3.

which insert and merge can be made to run in $O(1)$ time for various flavors of binomial heaps.

Binomial heaps are composed of more primitive objects known as binomial trees. Binomial trees are inductively defined as follows:

- A binomial tree of rank 0 is a singleton node.
- A binomial tree of rank $r + 1$ is formed by *linking* two binomial trees of rank r, making one tree the leftmost child of the other.

From this definition, it is easy to see that a binomial tree of rank r contains exactly 2^r nodes. There is a second, equivalent definition of binomial trees that is sometimes more convenient: a binomial tree of rank r is a node with r children $t_1 \ldots t_r$, where each t_i is a binomial tree of rank $r - i$. Figure 3.3 illustrates binomial trees of ranks 0 through 3.

We represent a node in a binomial tree as an element and a list of children. For convenience, we also annotate each node with its rank.

```
datatype Tree = Node of int × Elem.T × Tree list
```

Each list of children is maintained in decreasing order of rank, and elements are stored in heap order. We maintain heap order by always linking trees with larger roots under trees with smaller roots.

```
fun link (t1 as Node (r, x1, c1), t2 as Node (_, x2, c2)) =
      if Elem.leq (x1, x2) then Node (r+1, x1, t2 :: c1)
      else Node (r+1, x2, t1 :: c2)
```

We always link trees of equal rank.

Now, a binomial heap is a collection of heap-ordered binomial trees in which no two trees have the same rank. This collection is represented as a list of trees in increasing order of rank.

```
type Heap = Tree list
```

Because each binomial tree contains 2^r elements and no two trees have the same rank, the trees in a binomial heap of size n correspond exactly to the ones in the binary representation of n. For example, the binary representation of 21 is `10101` so a binomial heap of size 21 would contain one tree of rank 0, one tree of rank 2, and one tree of rank 4 (of sizes 1, 4, and 16, respectively). Note that, just as the binary representation of n contains at most $\lfloor \log(n+1) \rfloor$ ones, a binomial heap of size n contains at most $\lfloor \log(n+1) \rfloor$ trees.

We are now ready to describe the functions on binomial heaps. We begin with insert and merge, which are defined in loose analogy to incrementing or adding binary numbers. (We will tighten this analogy in Chapter 9.) To insert a new element into a heap, we first create a new singleton tree (i.e., a binomial tree of rank 0). We then step through the existing trees in increasing order of rank until we find a missing rank, linking trees of equal rank as we go. Each link corresponds to a carry in binary arithmetic.

```
fun rank (Node (r, x, c)) = r
fun insTree (t, [ ]) = [t]
  | insTree (t, ts as t' :: ts') =
      if rank t < rank t' then t :: ts else insTree (link (t, t'), ts')
fun insert (x, ts) = insTree (Node (0, x, [ ]), ts)
```

The worst case is insertion into a heap of size $n = 2^k - 1$, requiring a total of k links and $O(k) = O(\log n)$ time.

To merge two heaps, we step through both lists of trees in increasing order of rank, linking trees of equal rank as we go. Again, each link corresponds to a carry in binary arithmetic.

```
fun merge (ts1, [ ]) = ts1
  | merge ([ ], ts2) = ts2
  | merge (ts1 as t1 :: ts1', ts2 as t2 :: ts2') =
      if rank t1 < rank t2 then t1 :: merge (ts1', ts2)
      else if rank t2 < rank t1 then t2 :: merge (ts1, ts2')
      else insTree (link (t1, t2), merge (ts1', ts2'))
```

Both findMin and deleteMin call an auxiliary function removeMinTree that finds the tree with the minimum root and removes it from the list, returning both the tree and the remaining list.

```
fun removeMinTree [t] = (t, [ ])
  | removeMinTree (t :: ts) =
      let val (t', ts') = removeMinTree ts
      in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
```

Now, findMin simply returns the root of the extracted tree.

```
fun findMin ts = let val (t, _) = removeMinTree ts in root t end
```

The deleteMin function is a little trickier. After discarding the root of the extracted tree, we must somehow return the children of the discarded node to the remaining list of trees. Note that each list of children is *almost* a valid binomial heap. Each is a collection of heap-ordered binomial trees of unique rank, but in decreasing rather than increasing order of rank. Thus, we convert the list of children into a valid binomial heap by reversing it and then merge this list with the remaining trees.

```
fun deleteMin ts = let val (Node (_, x, ts1), ts2) = removeMinTree ts
                   in merge (rev ts1, ts2) end
```

The complete implementation of binomial heaps is shown in Figure 3.4. All four major operations require $O(\log n)$ time in the worst case.

Exercise 3.5 Define findMin directly rather than via a call to removeMinTree.

Exercise 3.6 Most of the rank annotations in this representation of binomial heaps are redundant because we know that the children of a node of rank r have ranks $r-1, \ldots, 0$. Thus, we can remove the rank annotations from each node and instead pair each tree at the top-level with its rank, i.e.,

```
datatype Tree = Node of Elem × Tree list
type Heap = (int × Tree) list
```

Reimplement binomial heaps with this new representation.

Exercise 3.7 One clear advantage of leftist heaps over binomial heaps is that findMin takes only $O(1)$ time, rather than $O(\log n)$ time. The following functor skeleton improves the running time of findMin to $O(1)$ by storing the minimum element separately from the rest of the heap.

```
functor ExplicitMin (H : HEAP) : HEAP =
struct
    structure Elem = H.Elem
    datatype Heap = E | NE of Elem.T × H.Heap
    ...
end
```

Note that this functor is not specific to binomial heaps, but rather takes any implementation of heaps as a parameter. Complete this functor so that findMin takes $O(1)$ time, and insert, merge, and deleteMin take $O(\log n)$ time (assuming that all four take $O(\log n)$ time or better for the underlying implementation H).

```
functor BinomialHeap (Element : ORDERED) : HEAP =
struct
  structure Elem = Element

  datatype Tree = Node of int × Elem.T × Tree list
  type Heap = Tree list

  val empty = []
  fun isEmpty ts = null ts

  fun rank (Node (r, x, c)) = r
  fun root (Node (r, x, c)) = x
  fun link (t1 as Node (r, x1, c1), t2 as Node (_, x2, c2)) =
        if Elem.leq (x1, x2) then Node (r+1, x1, t2 :: c1)
        else Node (r+1, x2, t1 :: c2)
  fun insTree (t, []) = [t]
    | insTree (t, ts as t' :: ts') =
        if rank t < rank t' then t :: ts else insTree (link (t, t'), ts')

  fun insert (x, ts) = insTree (Node (0, x, []), ts)
  fun merge (ts1, []) = ts1
    | merge ([], ts2) = ts2
    | merge (ts1 as t1 :: ts1', ts2 as t2 :: ts2') =
        if rank t1 < rank t2 then t1 :: merge (ts1', ts2)
        else if rank t2 < rank t1 then t2 :: merge (ts1, ts2')
        else insTree (link (t1, t2), merge (ts1', ts2'))

  fun removeMinTree [] = raise EMPTY
    | removeMinTree [t] = (t, [])
    | removeMinTree (t :: ts) =
        let val (t', ts') = removeMinTree ts
        in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end

  fun findMin ts = let val (t, _) = removeMinTree ts in root t end
  fun deleteMin ts =
        let val (Node (_, x, ts1), ts2) = removeMinTree ts
        in merge (rev ts1, ts2) end
end
```

Figure 3.4. Binomial heaps.

3.3 Red-Black Trees

In Section 2.2, we introduced binary search trees. Although these trees work very well on random or unordered data, they perform very poorly on ordered data, for which any individual operation might take up to $O(n)$ time. The solution to this problem is to keep each tree approximately balanced. Then no individual operation takes more than $O(\log n)$ time. Red-black trees [GS78] are one of the most popular families of balanced binary search trees.

A red-black tree is a binary search tree in which every node is colored either

red or black. We augment the type of binary search trees from Section 2.2 with a color field.

```
datatype Color = R | B
datatype Tree = E | T of Color × Tree × Elem × Tree
```

All empty nodes are considered to be black, so the empty constructor E does not need a color field.

We insist that every red-black tree satisfy the following two balance invariants:

Invariant 1. No red node has a red child.

Invariant 2. Every path from the root to an empty node contains the same number of black nodes.

Taken together, these two invariants guarantee that the longest possible path in a red-black tree, one with alternating black and red nodes, is no more than twice as long as the shortest possible path, one with black nodes only.

Exercise 3.8 Prove that the maximum depth of a node in a red-black tree of size n is at most $2\lfloor \log(n+1) \rfloor$. ◇

The member function on red-black trees ignores the color fields. Except for a wildcard in the T case, it is identical to the member function on unbalanced search trees.

```
fun member (x, E) = false
  | member (x, T (_, a, y, b)) =
      if x < y then member (x, a)
      else if x > y then member (x, b)
      else true
```

The insert function is more interesting because it must maintain the two balance invariants.

```
fun insert (x, s) =
    let fun ins E = T (R, E, x, E)
          | ins (s as T (color, a, y, b)) =
              if x < y then balance (color, ins a, y, b)
              else if x > y then balance (color, a, y, ins b)
              else s
        val T (_, a, y, b) = ins s   (* guaranteed to be non-empty *)
    in T (B, a, y, b) end
```

This function extends the insert function for unbalanced search trees in three significant ways. First, when we create a new node in the ins E case, we initially color it red. Second, we force the final root to be black, regardless of the color returned by ins. Finally, we replace the calls to the T constructor in the $x < y$ and $x > y$ cases with calls to the balance function. The balance

function acts just like the T constructor except that it massages its arguments as necessary to enforce the balance invariants.

Coloring the new node red maintains Invariant 2, but violates Invariant 1 whenever the parent of the new node is red. We allow a single red-red violation at a time, and percolate this violation up the search path toward the root during rebalancing. The balance function detects and repairs each red-red violation when it processes the black parent of the red node with a red child. This black-red-red path can occur in any of four configurations, depending on whether each red node is a left or right child. However, the solution is the same in every case: rewrite the black-red-red path as a red node with two black children, as illustrated in Figure 3.5. This transformation can be coded as follows:

```
fun balance (B,T (R,T (R,a,x,b),y,c),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,T (R,a,x,T (R,b,y,c)),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,a,x,T (R,T (R,b,y,c),z,d)) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,a,x,T (R,b,y,T (R,c,z,d))) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance body = T body
```

It is routine to verify that the red-black balance invariants both hold for the resulting (sub)tree.

Remark Notice that the right-hand sides of the first four clauses are identical. Some implementations of Standard ML, notably Standard ML of New Jersey, support a feature known as *or-patterns* that allows multiple clauses with identical right-hand sides to be collapsed into a single clause [FB97]. Using or-patterns, the balance function might be rewritten

```
fun balance ( (B,T (R,T (R,a,x,b),y,c),z,d)
            | (B,T (R,a,x,T (R,b,y,c)),z,d)
            | (B,a,x,T (R,T (R,b,y,c),z,d))
            | (B,a,x,T (R,b,y,T (R,c,z,d))) ) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance body = T body
```

◇

After balancing a given subtree, the red root of that subtree might now be the child of another red node. Thus, we continue balancing all the way to the top of the tree. At the very top of the tree, we might end up with a red node with a red child, but with no black parent. We handle this case by always recoloring the root to be black.

This implementation of red-black trees is summarized in Figure 3.6.

Hint to Practitioners: Even without optimization, this implementation of balanced binary search trees is one of the fastest around. With appropriate optimizations, such as Exercises 2.2 and 3.10, it really flies!

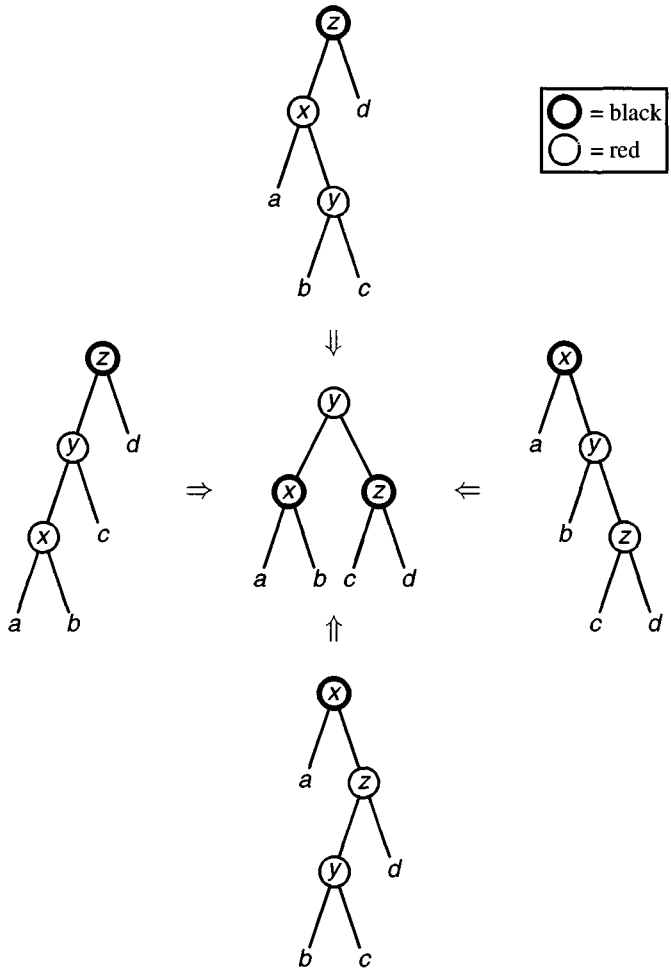

Figure 3.5. Eliminating red nodes with red parents.

Remark One of the reasons this implementation is so much simpler than typical presentations of red-black trees (e.g., Chapter 14 of [CLR90]) is that it uses subtly different rebalancing transformations. Imperative implementations typically split the four dangerous cases considered here into eight cases, according to the color of the sibling of the red node with a red child. Knowing the color of the red parent's sibling allows the transformations to use fewer assignments in some cases and to terminate rebalancing early in others. However, in a func-

```
functor RedBlackSet (Element : ORDERED) : SET =
struct
  type Elem = Element.T

  datatype Color = R | B
  datatype Tree = E | T of Color × Tree × Elem × Tree
  type Set = Tree

  val empty = E

  fun member (x, E) = false
    | member (x, T (_, a, y, b)) =
        if Element.lt (x, y) then member (x, a)
        else if Element.lt (y, x) then member (x, b)
        else true

  fun balance (B,T (R,T (R,a,x,b),y,c),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
    | balance (B,T (R,a,x,T (R,b,y,c)),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
    | balance (B,a,x,T (R,T (R,b,y,c),z,d)) = T (R,T (B,a,x,b),y,T (B,c,z,d))
    | balance (B,a,x,T (R,b,y,T (R,c,z,d))) = T (R,T (B,a,x,b),y,T (B,c,z,d))
    | balance body = T body

  fun insert (x, s) =
        letfun ins E = T (R, E, x, E)
             | ins (s as T (color, a, y, b)) =
                 if Element.lt (x, y) then balance (color, ins a, y, b)
                 else if Element.lt (y, x) then balance (color, a, y, ins b)
                 else s
           val T (_, a, y, b) = ins s   (* guaranteed to be non-empty *)
        in T (B, a, y, b) end
end
```

Figure 3.6. Red black trees.

tional setting, where we are copying the nodes in question anyway, we cannot reduce the number of assignments in this fashion, nor can we terminate copying early, so there is no point is using the more complicated transformations.

Exercise 3.9 Write a function fromOrdList of type Elem list $\rightarrow$ Tree that converts a sorted list with no duplicates into a red-black tree. Your function should run in $O(n)$ time.

Exercise 3.10 The balance function currently performs several unnecessary tests. For example, when the ins function recurses on the left child, there is no need for balance to test for red-red violations involving the right child.

(a) Split balance into two functions, lbalance and rbalance, that test for vio-

lations involving the left child and right child, respectively. Replace the calls to balance in ins with calls to either lbalance or rbalance.

(b) Extending the same logic one step further, one of the remaining tests on the grandchildren is also unnecessary. Rewrite ins so that it never tests the color of nodes not on the search path.

3.4 Chapter Notes

Núñez, Palao, and Peña [NPP95] and King [Kin94] describe similar implementations in Haskell of leftist heaps and binomial heaps, respectively. Red-black trees have not previously appeared in the functional programming literature, but several other kinds of balanced binary search trees have, including AVL trees [Mye82, Mye84, BW88, NPP95], 2-3 trees [Rea92], and weight-balanced trees [Ada93].

Knuth [Knu73a] originally introduced leftist heaps as a simplification of a data structure by Crane [Cra72]. Vuillemin [Vui78] invented binomial heaps; Brown [Bro78] examined many of the properties of this elegant data structure. Guibas and Sedgewick [GS78] proposed red-black trees as a general framework for describing many other kinds of balanced trees.