5

Leftist Trees

Sartaj Sahni
University of Florida

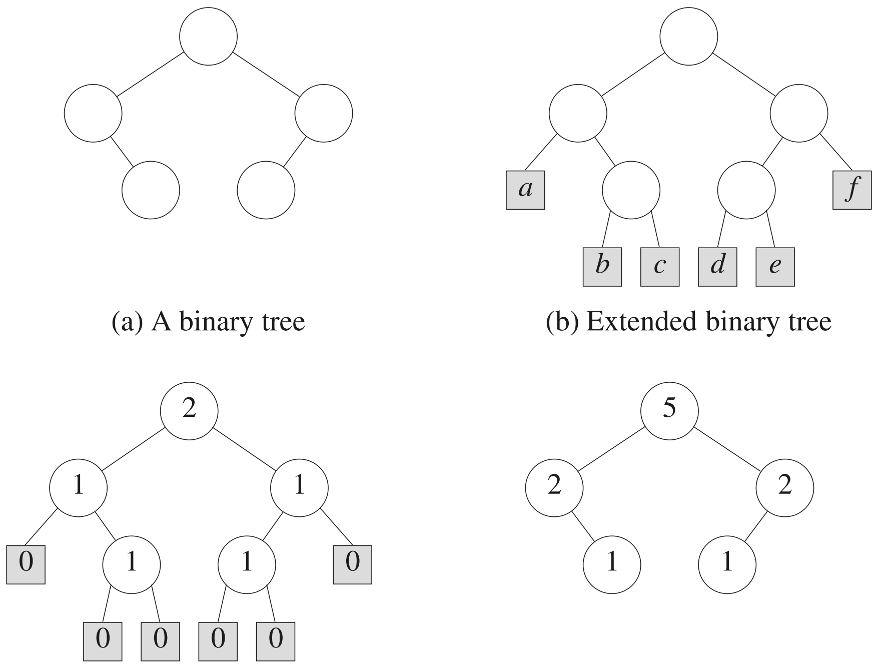
5.1 Introduction

A single-ended priority queue (or simply, a priority queue) is a collection of elements in which each element has a priority. There are two varieties of priority queues—max and min. The primary operations supported by a max (min) priority queue are (a) find the element with maximum (minimum) priority, (b) insert an element, and (c) delete the element whose priority is maximum (minimum). However, many authors consider additional operations such as (d) delete an arbitrary element (assuming we have a pointer to the element), (e) change the priority of an arbitrary element (again assuming we have a pointer to this element), (f) meld two max (min) priority queues (i.e., combine two max (min) priority queues into one), and (g) initialize a priority queue with a nonzero number of elements.

Several data structures: e.g., heaps (Chapter 3), leftist trees [2, 5], Fibonacci heaps [7] (Chapter 7), binomial heaps [1] (Chapter 7), skew heaps [11] (Chapter 6), and pairing heaps [6] (Chapter 7) have been proposed for the representation of a priority queue. The different data structures that have been proposed for the representation of a priority queue differ in terms of the performance guarantees they provide. Some guarantee good performance on a per operation basis while others do this only in the amortized sense. Max (min) heaps permit one to delete the max (min) element and insert an arbitrary element into an n element priority queue in $O(\log n)$ time per operation; a find max (min) takes $O(1)$ time. Additionally, a heap is an implicit data structure that has no storage overhead associated with it. All other priority queue structures are pointer-based and so require additional storage for the pointers.

Max (min) leftist trees also support the insert and delete max (min) operations in $O(\log n)$ time per operation and the find max (min) operation in $O(1)$ time. Additionally, they permit us to meld pairs of priority queues in logarithmic time.

The remaining structures do not guarantee good complexity on a per operation basis. They do, however, have good amortized complexity. Using Fibonacci heaps, binomial queues, or skew heaps, find max (min), inserts and melds take $O(1)$ time (actual and amortized) and a delete max (min) takes $O(\log n)$ amortized time. When a pairing heap is



(a) A binary tree (b) Extended binary tree

(c) *s* values (d) *w* values

FIGURE 5.1: s and w values.

used, the amortized complexity is $O(1)$ for find max (min) and insert (provided no decrease key operations are performed) and $O(logn)$ for delete max (min) operations [12]. Jones [8] gives an empirical evaluation of many priority queue data structures.

In this chapter, we focus on the leftist tree data structure. Two varieties of leftist trees– height-biased leftist trees [5] and weight-biased leftist trees [2] are described. Both varieties of leftist trees are binary trees that are suitable for the representation of a single-ended priority queue. When a max (min) leftist tree is used, the traditional single-ended priority queue operations– find max (min) element, delete/remove max (min) element, and insert an element–take, respectively, $O(1)$, $O(\log n)$ and $O(\log n)$ time each, where n is the number of elements in the priority queue. Additionally, an n-element max (min) leftist tree can be initialized in $O(n)$ time and two max (min) leftist trees that have a total of n elements may be melded into a single max (min) leftist tree in $O(\log n)$ time.

5.2 Height-Biased Leftist Trees

5.2.1 Definition

Consider a binary tree in which a special node called an **external node** replaces each empty subtree. The remaining nodes are called **internal nodes**. A binary tree with external nodes added is called an **extended binary tree**. Figure 5.1(a) shows a binary tree. Its corresponding extended binary tree is shown in Figure 5.1(b). The external nodes appear as shaded boxes. These nodes have been labeled a through f for convenience.

Let $s(x)$ be the length of a shortest path from node x to an external node in its subtree. From the definition of $s(x)$, it follows that if x is an external node, its s value is 0.

Furthermore, if x is an internal node, its s value is

$$\min\{s(L), s(R)\} + 1$$

where L and R are, respectively, the left and right children of x. The s values for the nodes of the extended binary tree of Figure 5.1(b) appear in Figure 5.1(c).

DEFINITION 5.1 [Crane [5]] A binary tree is a **height-biased leftist tree (HBLT)** iff at every internal node, the s value of the left child is greater than or equal to the s value of the right child.

The binary tree of Figure 5.1(a) is not an HBLT. To see this, consider the parent of the external node a. The s value of its left child is 0, while that of its right is 1. All other internal nodes satisfy the requirements of the HBLT definition. By swapping the left and right subtrees of the parent of a, the binary tree of Figure 5.1(a) becomes an HBLT.

THEOREM 5.1 *Let x be any internal node of an HBLT.*

(a) *The number of nodes in the subtree with root x is at least $2^{s(x)} - 1$.*
(b) *If the subtree with root x has m nodes, $s(x)$ is at most $\log_2(m + 1)$.*
(c) *The length, rightmost(x), of the right-most path from x to an external node (i.e., the path obtained by beginning at x and making a sequence of right-child moves) is $s(x)$.*

Proof From the definition of $s(x)$, it follows that there are no external nodes on the $s(x) - 1$ levels immediately below node x (as otherwise the s value of x would be less). The subtree with root x has exactly one node on the level at which x is, two on the next level, four on the next, $\cdots$, and $2^{s(x)-1}$ nodes $s(x) - 1$ levels below x. The subtree may have additional nodes at levels more than $s(x) - 1$ below x. Hence the number of nodes in the subtree x is at least $\sum_{i=0}^{s(x)-1} 2^i = 2^{s(x)} - 1$. Part (b) follows from (a). Part (c) follows from the definition of s and the fact that, in an HBLT, the s value of the left child of a node is always greater than or equal to that of the right child.

DEFINITION 5.2 A **max tree** (**min tree**) is a tree in which the value in each node is greater (less) than or equal to those in its children (if any).

Some max trees appear in Figure 5.2, and some min trees appear in Figure 5.3. Although these examples are all binary trees, it is not necessary for a max tree to be binary. Nodes of a max or min tree may have an arbitrary number of children.

DEFINITION 5.3 A **max HBLT** is an HBLT that is also a max tree. A **min HBLT** is an HBLT that is also a min tree.

The max trees of Figure 5.2 as well as the min trees of Figure 5.3 are also HBLTs; therefore, the trees of Figure 5.2 are max HBLTs, and those of Figure 5.3 are min HBLTs. A max priority queue may be represented as a max HBLT, and a min priority queue may be represented as a min HBLT.

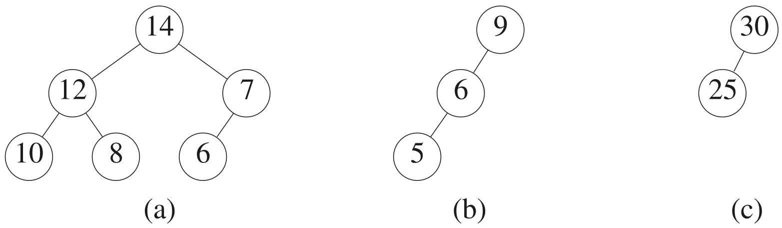


FIGURE 5.2: Some max trees.

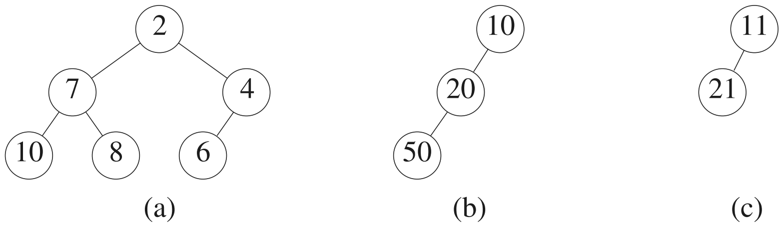


FIGURE 5.3: Some min trees.

5.2.2 Insertion into a Max HBLT

The insertion operation for max HBLTs may be performed by using the max HBLT meld operation, which combines two max HBLTs into a single max HBLT. Suppose we are to insert an element x into the max HBLT H. If we create a max HBLT with the single element x and then meld this max HBLT and H, the resulting max HBLT will include all elements in H as well as the element x. Hence an insertion may be performed by creating a new max HBLT with just the element that is to be inserted and then melding this max HBLT and the original.

5.2.3 Deletion of Max Element from a Max HBLT

The max element is in the root. If the root is deleted, two max HBLTs, the left and right subtrees of the root, remain. By melding together these two max HBLTs, we obtain a max HBLT that contains all elements in the original max HBLT other than the deleted max element. So the delete max operation may be performed by deleting the root and then melding its two subtrees.

5.2.4 Melding Two Max HBLTs

Since the length of the right-most path of an HBLT with n elements is $O(\log n)$, a meld algorithm that traverses only the right-most paths of the HBLTs being melded, spending $O(1)$ time at each node on these two paths, will have complexity logarithmic in the number of elements in the resulting HBLT. With this observation in mind, we develop a meld algorithm that begins at the roots of the two HBLTs and makes right-child moves only.

The meld strategy is best described using recursion. Let A and B be the two max HBLTs that are to be melded. If one is empty, then we may use the other as the result. So assume that neither is empty. To perform the meld, we compare the elements in the two roots. The root with the larger element becomes the root of the melded HBLT. Ties may be broken arbitrarily. Suppose that A has the larger root and that its left subtree is L. Let C be the max HBLT that results from melding the right subtree of A and the max HBLT B. The result of melding A and B is the max HBLT that has A as its root and L and C as its subtrees. If the s value of L is smaller than that of C, then C is the left subtree. Otherwise, L is.

Example 5.1

Consider the two max HBLTs of Figure 5.4(a). The s value of a node is shown outside the node, while the element value is shown inside. When drawing two max HBLTs that are to be melded, we will always draw the one with larger root value on the left. Ties are broken arbitrarily. Because of this convention, the root of the left HBLT always becomes the root of the final HBLT. Also, we will shade the nodes of the HBLT on the right.

Since the right subtree of 9 is empty, the result of melding this subtree of 9 and the tree with root 7 is just the tree with root 7. We make the tree with root 7 the right subtree of 9 temporarily to get the max tree of Figure 5.4(b). Since the s value of the left subtree of 9 is 0 while that of its right subtree is 1, the left and right subtrees are swapped to get the max HBLT of Figure 5.4(c).

Next consider melding the two max HBLTs of Figure 5.4(d). The root of the left subtree becomes the root of the result. When the right subtree of 10 is melded with the HBLT with root 7, the result is just this latter HBLT. If this HBLT is made the right subtree of 10, we get the max tree of Figure 5.4(e). Comparing the s values of the left and right children of 10, we see that a swap is not necessary.

Now consider melding the two max HBLTs of Figure 5.4(f). The root of the left subtree is the root of the result. We proceed to meld the right subtree of 18 and the max HBLT with root 10. The two max HBLTs being melded are the same as those melded in Figure 5.4(d). The resultant max HBLT (Figure 5.4(e)) becomes the right subtree of 18, and the max tree of Figure 5.4(g) results. Comparing the s values of the left and right subtrees of 18, we see that these subtrees must be swapped. Swapping results in the max HBLT of Figure 5.4(h).

As a final example, consider melding the two max HBLTs of Figure 5.4(i). The root of the left max HBLT becomes the root of the result. We proceed to meld the right subtree of 40 and the max HBLT with root 18. These max HBLTs were melded in Figure 5.4(f). The resultant max HBLT (Figure 5.4(g)) becomes the right subtree of 40. Since the left subtree of 40 has a smaller s value than the right has, the two subtrees are swapped to get the max HBLT of Figure 5.4(k). Notice that when melding the max HBLTs of Figure 5.4(i), we first move to the right child of 40, then to the right child of 18, and finally to the right child of 10. All moves follow the right-most paths of the initial max HBLTs.

5.2.5 Initialization

It takes $O(n \log n)$ time to initialize a max HBLT with n elements by inserting these elements into an initially empty max HBLT one at a time. To get a linear time initialization algorithm, we begin by creating n max HBLTs with each containing one of the n elements. These n max HBLTs are placed on a FIFO queue. Then max HBLTs are deleted from this queue in pairs, melded, and added to the end of the queue until only one max HBLT remains.

Example 5.2

We wish to create a max HBLT with the five elements 7, 1, 9, 11, and 2. Five single-element max HBLTs are created and placed in a FIFO queue. The first two, 7 and 1,

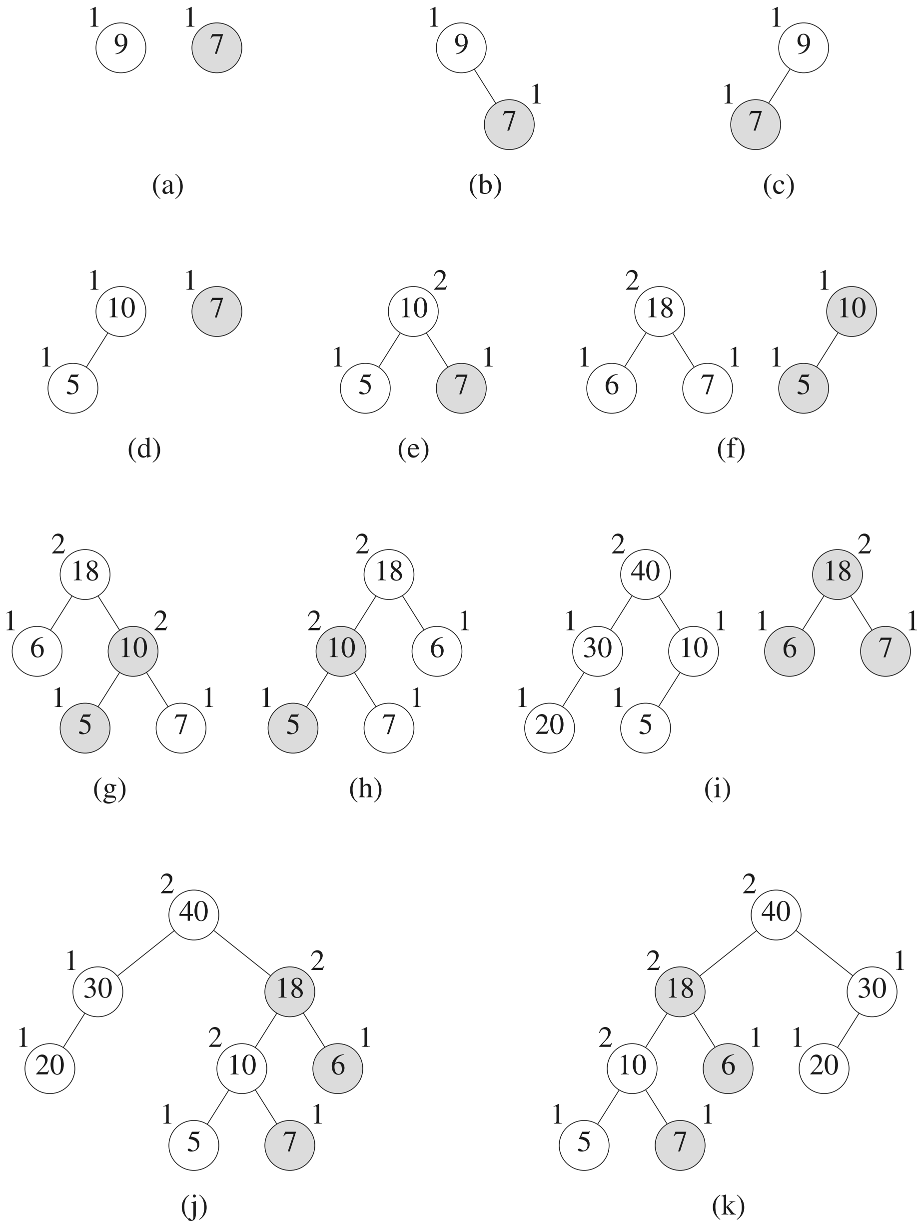


FIGURE 5.4: Melding max HBLTs.

are deleted from the queue and melded. The result (Figure 5.5(a)) is added to the queue. Next the max HBLTs 9 and 11 are deleted from the queue and melded. The result appears in Figure 5.5(b). This max HBLT is added to the queue. Now the max HBLT 2 and that of Figure 5.5(a) are deleted from the queue and melded. The resulting max HBLT (Figure 5.5(c)) is added to the queue. The next pair to be deleted from the queue consists of the max HBLTs of Figures Figure 5.5 (b) and (c). These HBLTs are melded to get the max HBLT of Figure 5.5(d). This max HBLT is added to the queue. The queue now has just one max HBLT, and we are done with the initialization.

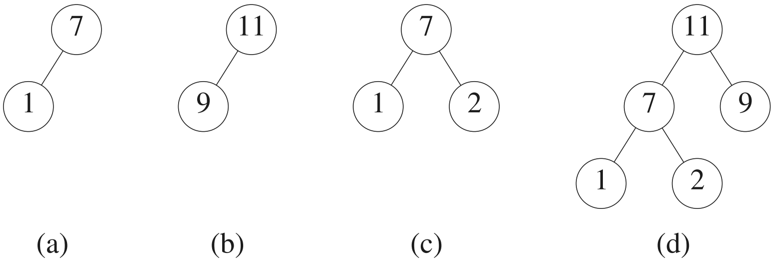


FIGURE 5.5: Initializing a max HBLT.

For the complexity analysis of of the initialization operation, assume, for simplicity, that n is a power of 2. The first $n/2$ melds involve max HBLTs with one element each, the next $n/4$ melds involve max HBLTs with two elements each; the next $n/8$ melds are with trees that have four elements each; and so on. The time needed to meld two leftist trees with 2^i elements each is $O(i+1)$, and so the total time for the initialization is

$$O(n/2 + 2*(n/4) + 3*(n/8) + \cdots) = O(n \sum \frac{i}{2^i}) = O(n)$$

5.2.6 Deletion of Arbitrary Element from a Max HBLT

Although deleting an element other than the max (min) element is not a standard operation for a max (min) priority queue, an efficient implementation of this operation is required when one wishes to use the generic methods of Cho and Sahni [3] and Chong and Sahni [4] to derive efficient mergeable double-ended priority queue data structures from efficient single-ended priority queue data structures. From a max or min leftist tree, we may remove the element in any specified node *theNode* in $O(\log n)$ time, making the leftist tree a suitable base structure from which an efficient mergeable double-ended priority queue data structure may be obtained [3, 4].

To remove the element in the node *theNode* of a height-biased leftist tree, we must do the following:

1. Detach the subtree rooted at *theNode* from the tree and replace it with the meld of the subtrees of *theNode*.
2. Update s values on the path from *theNode* to the root and swap subtrees on this path as necessary to maintain the leftist tree property.

To update s on the path from *theNode* to the root, we need parent pointers in each node. This upward updating pass stops as soon as we encounter a node whose s value does not change. The changed s values (with the exception of possibly $O(\log n)$ values from moves made at the beginning from right children) must form an ascending sequence (actually, each must be one more than the preceding one). Since the maximum s value is $O(\log n)$ and since all s values are positive integers, at most $O(\log n)$ nodes are encountered in the updating pass. At each of these nodes, we spend $O(1)$ time. Therefore, the overall complexity of removing the element in node *theNode* is $O(\log n)$.

5.3 Weight-Biased Leftist Trees

5.3.1 Definition

We arrive at another variety of leftist tree by considering the number of nodes in a subtree, rather than the length of a shortest root to external node path. Define the weight $w(x)$ of node x to be the number of internal nodes in the subtree with root x. Notice that if x is an external node, its weight is 0. If x is an internal node, its weight is 1 more than the sum of the weights of its children. The weights of the nodes of the binary tree of Figure 5.1(a) appear in Figure 5.1(d)

DEFINITION 5.4 [Cho and Sahni [2]] A binary tree is a **weight-biased leftist tree (WBLT)** iff at every internal node the w value of the left child is greater than or equal to the w value of the right child. A max (min) WBLT is a max (min) tree that is also a WBLT.

Note that the binary tree of Figure 5.1(a) is not a WBLT. However, all three of the binary trees of Figure 5.2 are WBLTs.

THEOREM 5.2 *Let x be any internal node of a weight-biased leftist tree. The length, $rightmost(x)$, of the right-most path from x to an external node satisfies*

$$rightmost(x) \leq \log_2(w(x) + 1).$$

Proof The proof is by induction on $w(x)$. When $w(x) = 1$, $rightmost(x) = 1$ and $\log_2(w(x) + 1) = \log_2 2 = 1$. For the induction hypothesis, assume that $rightmost(x) \leq \log_2(w(x)+1)$ whenever $w(x) < n$. Let $RightChild(x)$ denote the right child of x (note that this right child may be an external node). When $w(x) = n$, $w(RightChild(x)) \leq (n-1)/2$ and $rightmost(x) = 1 + rightmost(RightChild(x)) \leq 1 + \log_2((n-1)/2+1) = 1 + \log_2(n+1) - 1 = \log_2(n+1)$.

5.3.2 Max WBLT Operations

Insert, delete max, and initialization are analogous to the corresponding max HBLT operation. However, the meld operation can be done in a single top-to-bottom pass (recall that the meld operation of an HBLT performs a top-to-bottom pass as the recursion unfolds and then a bottom-to-top pass in which subtrees are possibly swapped and s-values updated). A single-pass meld is possible for WBLTs because we can determine the w values on the way down and so, on the way down, we can update w-values and swap subtrees as necessary. For HBLTs, a node's new s value cannot be determined on the way down the tree.

Since the meld operation of a WBLT may be implemented using a single top-to-bottom pass, inserts and deletes also use a single top-to-bottom pass. Because of this, inserts and deletes are faster, by a constant factor, in a WBLT than in an HBLT [2]. However, from a WBLT, we cannot delete the element in an arbitrarily located node, $theNode$, in $O(\log n)$ time. This is because $theNode$ may have $O(n)$ ancestors whose w value is to be updated. So, WBLTs are not suitable for mergeable double-ended priority queue applications [3, 8].

C++ and Java codes for HBLTs and WBLTs may be obtained from [9] and [10], respectively.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

References

[1] M. Brown, Implementation and analysis of binomial queue algorithms, *SIAM Jr. on Computing*, 7, 3, 1978, 298-319.

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

[3] S. Cho and S. Sahni, Mergeable double-ended priority queues. *International Journal on Foundations of Computer Science*, 10, 1, 1999, 1-18.

[4] K. Chong and S. Sahni, Correspondence based data structures for double ended priority queues. *ACM Jr. on Experimental Algorithmics*, Volume 5, 2000, Article 2, 22 pages.

[5] C. Crane, Linear Lists and Priority Queues as Balanced Binary Trees, Tech. Rep. CS-72-259, Dept. of Comp. Sci., Stanford University, 1972.

[6] M. Fredman, R. Sedgewick, D. Sleator, and R.Tarjan, The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1, 1986, 111-129.

[7] M. Fredman and R. Tarjan, Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, *JACM*, 34, 3, 1987, 596-615.

[8] D. Jones, An empirical comparison of priority-queue and event-set implementations, *Communications of the ACM*, 29, 4, 1986, 300-311.

[9] S. Sahni, *Data Structures, Algorithms, and Applications in C++*, McGraw-Hill, NY, 1998, 824 pages.

[10] S. Sahni, *Data Structures, Algorithms, and Applications in Java*, McGraw-Hill, NY, 2000, 846 pages.

[11] D. Sleator and R. Tarjan, Self-adjusting heaps, *SIAM Jr. on Computing*, 15, 1, 1986, 52-69.

[12] J. Stasko and J. Vitter, Pairing heaps: Experiments and analysis, *Communications of the ACM*, 30, 3, 1987, 234-249.