J. Inform. Process. Cybernet. **EIK 30** (1994) 1, 29-43
(formerly: Elektron. Inform.verarb. Kybernet.)

Segmented Winner Trees[1]

By Andrew Lim and Sartaj Sahni

Abstract: A new data structure, *segmented winner tree*, is introduced. This is useful when one needs to represent data that are partitioned into segments. The segment operations that are efficiently supported are: initialize a unit length segment, find the element with least value in any given segment, update an element in any segment, merge two adjacent segments, and split a segment.
Key Words and Phrases: Data structures, winner trees, segmented data

1. Introduction

Suppose that we have n elements $1, 2, \ldots, n$ that have been partitioned into some number of segments such that the elements in each segment are contiguous. Figure 1 shows a possible partitioning of the eleven elements $1, 2, \ldots, 11$ into three segments. A value, $v[i]$, is associated with each element i.

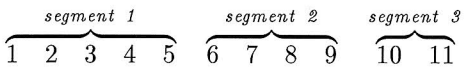


Figure 1: Eleven elements partitioned into three segments

The operations we wish to perform on these segments are:

1. *Initialize*(l_1): Initialize the one element segment $l_1 : r_1$ where $r_1 = l_1$.
2. *FindMin*(l_1, r_1, x): Find the smallest value in the segment comprised of elements $l_1 : r_1$. This value is returned in variable x.
3. *Update*(l_1, r_1, i, y): Change the value of element i to y. Element i is a member of the segment that is comprised of elements $l_1 : r_1$. Note that $l_1 \leq i \leq r_1$.
4. *Merge*(l_1, r_1, r_2): Merge the adjacent segments $l_1 : r_1$ and $r_1 + 1 : r_2$ into a single segment $l_1 : r_2$.
5. *Add*(l_1, r_1, y): This is a special case of merge in which the right segment $r_1 + 1 : r_2$ consists of a single element with value y. I.e., $r_1 + 1 = r_2$ and $v[r_2] = y$.

[1]This research was supported, in part, by the National Science Foundation under grant MIP91-03379.

6. *Split*(l_1, r_1, q)*:* The segment $l_1 : r_1$ is split into two adjacent segments $l_1 : q$ and $q + 1 : r_1$.

Two applications for a data structure that efficiently supports these operations are:

1. *Geographic Service Environment:* In this, the n elements represent n clients who are located adjacent to one another on the same side of a street. The clients are served by servers who, for efficiency reasons, serve a contiguous set of clients. So, each segment represents a set of clients served by the same server. In the ideal situation, we have one server per client. In this case all segments are of size one. If a server is disabled, then the set of clients served is assigned to one of the servers associated with an adjacent server (segment merging). Alternatively, the client set could be partitioned (segment split) with the left (right) partition being assigned to the left (right) server (segment merging). If a new server becomes available, then a client set for this server is created by partitioning off clients from at most two adjacent client sets (segment splitting). A server selects a client from its client set, for service, based on the clients value. We assume priority is given to the client with the least value. This uses the *FindMin* operation. The value (or priority) of a client may change in time. This requires one to update the clients value.
2. *Cell Joining:* When stretching two adjacent cells of a VLSI design so as to minimize the total wire length subject to a minimum area constraint[1], the pins on one side of a cell represent the n elements of our data structure. The pins are maintained as segments of adjacent pins and the distance between pins is the pin value. In the course of the algorithm, adjacent pin segments may coalesce; it is necessary to obtain the pin with the least value in any segment; and it is necessary to update the pin values. To implement this algorithm efficiently, we need an efficient data structure for the segment operations (except split) defined above.

The data structure we propose, *segmented winner trees*, represents each segment as a collection of winner trees [2] with each winner tree representing a number of elements that is a power of 2. All winner trees for all segments are compactly represented within an array of size $2n - 1$. Because of this, explicit pointers are not used and one can move up and down a tree by performing simple arithmetic operations on the current location within a tree [2].

Two of the already known data structures that could be used to represent our segments are *Leftist trees* and *Fibonacci heaps* [2, 3]. Table 1 compares the complexity of these data structures and that of segmented winner trees. s is the size of the segment involved (in the case of a merge, it is the size of the resulting segment). The complexities are amortized complexities for the case of Fibonacci heaps and worst case per operation complexities for leftist trees and segmented winner trees.

We note that if the updates are restricted to those that only reduce a value, then the amortized update cost for Fibonacci heaps is $O(1)$.

Segmented winner trees are defined in Section 2. Algorithms for the segmented winner tree operations are provided in Section 3. Experimental data comparing the performance of leftist trees, Fibonacci heaps, and segmented winner trees is provided in Section 4.

Operation	Leftist Trees	Fibonacci Heaps	Segmented Winner Trees
Initialize	$O(1)$	$O(1)$	$O(1)$
FindMin	$O(1)$	$O(1)$	$O(\log s)$
Update	$O(s)$	$O(s)$	$O(\log s)$
Merge	$O(\log s)$	$O(1)$	$O(\log s)$
Add	$O(\log s)$	$O(1)$	$O(\log s)$
Split	$O(s)$	$O(s)$	$O(1)$

Table 1: Data structure complexities

2. Segmented Winner Trees

A *winner tree* is a tree where each node represents the smaller of its children [2]. A *segmented winner tree* is a complete binary tree [2]. I.e., each node has degree 0 or 2; the leaves are on at most two adjacent levels; if the leaves are on two adjacent levels, then all leaves on the next to last level are at the right end of that level. Figure 2 shows a complete binary tree with 11 leaves and 21 nodes. Node numbers are inside the nodes. The leaves have been numbered left to right. The leaf numbers are given outside the leaves. Note that for every m, $m \geq 0$, there is a unique m node complete binary tree. Further, for every n, $n \geq 0$, there is a unique n leaf complete binary tree. The number, m, of nodes in this complete binary tree is $2n-1$ when $n \geq 1$ and 0 when $n = 0$.

A complete binary tree with m nodes is efficiently stored in an array $t[1..m]$ with node i of the tree represented in position i of the array. The parent of node i is in position i div 2, its left child (if it exists) is in position $2i$, and its right child (if it exists) is in position $2i+1$ [2].

To represent a collection of segments with n elements, we use a complete binary tree with n leaves. This has $m = 2n-1$ nodes. Element i is represented by leaf i, where the leaves are numbered 1 through n left to right (see Figure 2). The position of element i in the array $t[1..m]$ is easily obtained using the function *Position* of Figure 3. In this, k is a power of 2 such that $\frac{k}{2} < n \leq k$. So, when $8 < n \leq 16$, $k = 16$. I.e. $k = 2^{\lceil \log_2 n \rceil}$, $n > 0$. k gives the position of element 1. Hence, we expect to find element i in position $j = k + i - 1$ unless $j > n$. In this case, only $m - k + 1$ leaves are at the lowest level and $i > m - k + 1$. The first leaf on the preceding level is at position $\lfloor \frac{m}{2} \rfloor + 1$. So, the i'th leaf is in position $\lfloor \frac{m}{2} \rfloor + 1 + i - (m - k + 1) - 1 = \lfloor \frac{2n-1}{2} \rfloor - (2n-1) + j = j - n$.

The non leaf nodes of a segmented winner tree are used to maintain winner trees. When we have only one segment, we get a winner tree as defined in [2]. However, when there is more than one segment, several winner trees are embedded in $t[\ldots]$. Consider the eleven element example of Figure 1. Segment 1 is represented by two winner trees comprised of the nodes labeled a and b in Figure 4 . The nodes labeled a form one tree and the node labeled b forms the other. The nodes labeled c, d, and e, respectively, form the three winner trees that represent segment 2 while the nodes labeled f form the single winner tree that represents segment 3. Note that each winner tree embedded in a segmented winner tree is a full binary tree.

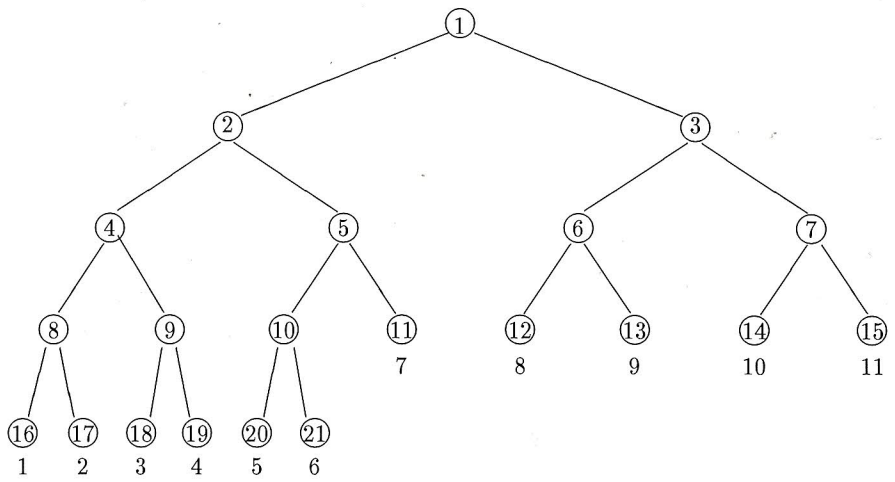


Figure 2: An 11 leaf 21 node complete binary tree

```
function Position(i);
    /* find the position of element i in t[1..m]
       k/2 < n ≤ k */
begin
    j := k + i − 1;
    if j > m then j := j − n;
    Position := j;
end;
```

Figure 3: Computing the position of element i

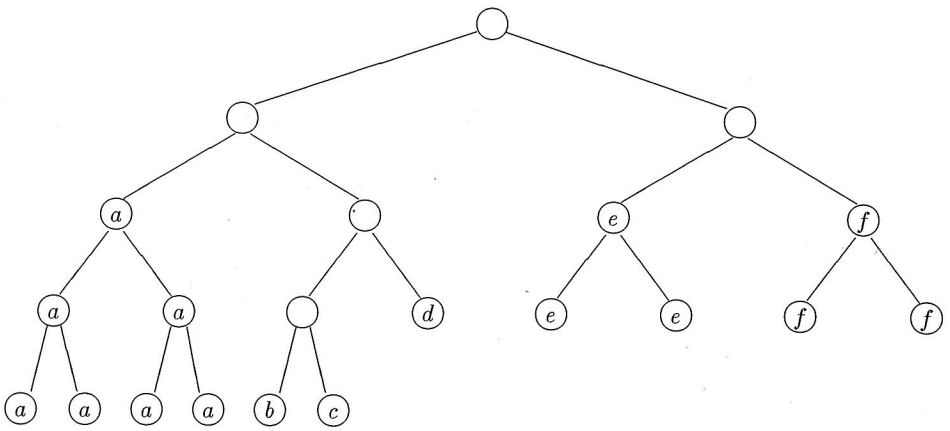
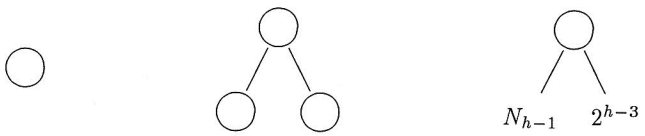


Figure 4: Segmented winner tree for example of Figure 1



(a) $h = 1$ (b) $h = 2$ (c) $h > 2$

Figure 5: Minimum winner trees

The winner trees that represent any segment are obtained by the following rules:

R1. All leaf nodes are members of winner trees.

R2. A non leaf node is a member of a winner tree iff all the leaves in the subtree of which it is root represent elements from the same segment.

Lemma 2.1 *The minimum number of leaves in a winner tree of height h that represents a portion (or all) of a segment is $2^{h-2}+1$ for $h \geq 2$ and 1 for $h = 1$.*

Proof. When $h = 1$, the winner tree is unique and consists of only a single leaf (i.e., no other nodes are in the tree). For $h = 2$, the winner tree is also unique and is shown in Figure 5(b). It has 2 leaves. Let N_h be the minimum number of nodes in any winner tree of height h, $h > 2$. Let T be such a tree. Since the leaves must be on two adjacent levels of T and since T has a minimum number of leaves, all leaves in the right subtree of T are on level $h-1$. Their number is 2^{h-3} (Figure 5(c)). Furthermore, the left subtree of T is a minimum winner tree of height $h-1$. So, it has N_{h-1} leaves.

Hence, we obtain the recurrence:

$$\begin{aligned} N_h &= N_{h-1} + 2^{h-3},\ h \geq 3 \\ N_2 &= 2 \end{aligned}$$

Its solution is $N_h = 2^{h-2} + 1,\ h \geq 2$. □

Lemma 2.2 *The following are true for every segment of size s, $s \geq 1$.*

(a) No winner tree in its representation has height $\geq \log_2(s-1) + 2$.
(b) The number of winner trees in its representation is at most

$$max\{2\lfloor \log_2 \frac{2(s+1)}{3} \rfloor,\ 2\lfloor \log_2 \frac{s+1}{2} \rfloor + 1\}.$$

Proof. (a) Follows from Lemma 2.1.
(b) Suppose that a collection of q winner trees represents some segment of size s. Assume that the leaves of these winner trees are all at the same level. If $q = 2k+1$ is odd, then since each winner tree has a number of leaves that is a power of 2, s is at least

$$2^0 + 2^1 + \ldots + 2^{k-1} + 2^k + 2^{k-1} + \ldots + 2^0$$

In this sum, each term represents the number of leaves in one of the winner trees that represents the segment. Note that no segment's representation can have more than two winner trees of the same size. From the lower bound on s, we obtain

$$s \geq 3 \times 2^k - 2$$

So, $k \leq \lfloor \log_2 \frac{s+2}{3} \rfloor$ (the floor results from the observation that k is an integer). Hence, $q \leq 2\lfloor \log_2 \frac{s+2}{3} \rfloor + 1$.

If $q = 2k$ is even, then we get

$$\begin{aligned} s &\geq 2^0 + 2^1 + \ldots + 2^{k-1} + 2^{k-1} + \ldots + 2^1 + 2^0 \\ &= 2^{k+1} - 2 \end{aligned}$$

So, $k \leq \lfloor \log_2 \frac{s+2}{2} \rfloor$ and $q \leq 2\lfloor \log_2 \frac{s+2}{2} \rfloor$. Combining the two cases, we get

$$q \leq max\{\ 2\lfloor \log_2 \frac{s+2}{2} \rfloor,\ 2\lfloor \log_2 \frac{s+2}{3} \rfloor + 1\ \}$$

One may verify that this is a tight bound.

Next suppose that the leaves of the winner trees are on two levels with s_1 leaves at the lowest level A and s_2 on the preceding level B. $s = s_1 + s_2$. Extend the s_2 leaves by adding their children (Figure 6). This results in a new segment of size $s' = s_1 + 2s_2$ with all leaves on the same level. We shall refer to the new $2s_2$ leaves on level A as *extended leaves* and to the segment of size s' as the *extended segment*. The number of winner trees, q, in this extended segment of size s' is the same as that in the original segment of size s. However, the rightmost tree for the extended segment must have an even number of leaves on level A. When $q = 2k$, we get

$$\begin{aligned} s' &\geq 2^0 + 2^1 + \ldots + 2^{k-1} + 2^k + 2^{k-1} + \ldots + 2^1 \\ &= 3 \times 2^k - 3 \end{aligned}$$

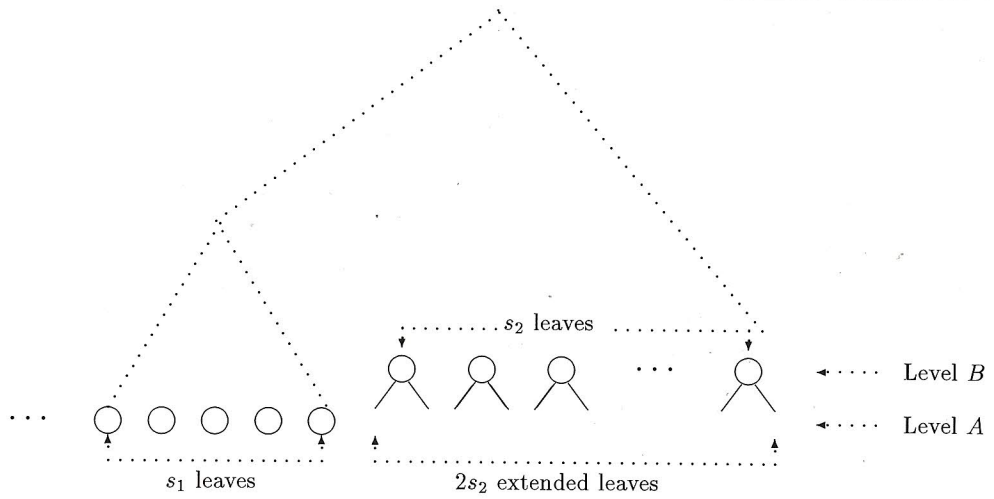


Figure 6: Leaf extension

So, $k \leq \lfloor \log_2 \frac{s'+3}{3} \rfloor$ and $q \leq 2\lfloor \log_2 \frac{s'+3}{3} \rfloor \leq 2\lfloor \log_2 \frac{s_1+2s_2+3}{3} \rfloor$. Since $s_1 + s_2 = s$, for any fixed s, $s_1 \geq 1$, $s_1 + 2s_2$ is maximized when $s_1 = 1$ and $s_2 = s - 1$. So, $q \leq 2\lfloor \log_2 \frac{1+2(s-1)+3}{3} \rfloor = 2\lfloor \log_2 \frac{2(s+1)}{3} \rfloor$. When $q = 2k + 1$, we get

$$\begin{aligned} s' &\geq 2^0 + 2^1 + \ldots + 2^k + 2^k + \ldots + 2^1 \\ &= 2^{k+2} - 3 \end{aligned}$$

Hence, $k \leq \lfloor \log_2 \frac{s'+3}{4} \rfloor$ and $q \leq 2\lfloor \log_2 \frac{s'+3}{4} \rfloor + 1 \leq 2\lfloor \log_2 \frac{s+1}{2} \rfloor + 1$. Combining the bounds for the two cases, we get

$$q \leq max \{ 2\lfloor \log_2 \frac{2(s+1)}{3} \rfloor, 2\lfloor \log_2 \frac{s+1}{2} \rfloor + 1 \}.$$

Since this bound is at least as large as that for the case when all leaves are at the same level, the lemma is proved. One may verify that this is a tight bound in that for every s, $s \geq 1$ one can construct a segment of size s that has the stated number of winner trees. □

Lemma 2.3 *Consider a segment $l_1 : r_1$ all of whose elements are represented by leaves at the same level. The height , h, of the rightmost winner tree in its representation is $min\{\lfloor \log_2(r \oplus (r+1)) \rfloor, \lfloor \log_2(r - l + 1) \rfloor\} + 1$ ($\oplus$ denotes the exclusive or of the binary representations of its two operands) where r = Position(r_1) and l = Position(l_1), and the position of the root of this winner tree is $\lfloor \frac{r}{2^{h-1}} \rfloor$.*

Proof. Let h be the height of the rightmost winner tree for the segment. Since all leaves are at the same level, the number of elements in this winner tree is 2^{h-1}. The

total number of elements in the segment is $r_1 - l_1 + 1 = r - l + 1$. So,

$$\begin{aligned} & 2^{h-1} \leq r - l + 1 \\ \text{or} \quad & h \leq \lfloor \log_2(r - l + 1) \rfloor + 1 \end{aligned}$$

If r is even, then the rightmost winner tree has height 1. If r is odd, then elements r_1 and $r_1 - 1$ (if it is in the same segment) can combine into the same tree. If the parent, q, of r is even then no further combining is possible. If q is odd, then $r_1, r_1 - 1, r_1 - 2$, and $r_1 - 3$ can combine provided $r_1 - 3 \geq l_1$. In general, the tallest winner tree the rightmost element r_1 can be in is determined by its nearest ancestor that is even (see Figure 4). I.e., it is determined by the position of the rightmost zero in the binary representation of r. So, when r is even the rightmost zero is in bit position 0 and the height is 1; when r is of the form $XX \ldots XX01$, the rightmost zero is in position 1 and the height is at most 2; and so on. The position of the rightmost zero in the binary representation of r is given by $\lfloor \log_2(r \oplus (r+1)) \rfloor$. So, $h \leq \lfloor \log_2(r \oplus (r+1)) \rfloor + 1$. The only time the height is smaller than this bound is when the segment has fewer than 2^{h-1} elements. At this time, the height is $\lfloor \log_2(r - l + 1) \rfloor + 1$. Combining, we get:

$$h = min\{\lfloor \log_2(r \oplus (r+1)) \rfloor, \lfloor \log_2(r - l + 1) \rfloor\} + 1.$$

The rightmost leaf of the winner tree is at r. Its parent is at $\lfloor \frac{r}{2} \rfloor$, its grandparent at $\lfloor \frac{r}{4} \rfloor$, etc. Since the height of the winner tree is h, its root must therefore be at $\lfloor \frac{r}{2^{h-1}} \rfloor$. □

3. The Algorithms

3.1. Initialize(l_1)

To initialize the one element segment $l_1 : l_1$, we need merely set $t[Position(l_1)] = v[l_1]$. Note that a one element segment is represented by a single winner tree that has just the node that represents the element. Initialization clearly takes $O(1)$ time.

3.2. FindMin(l_1, r_1, x)

To find the smallest element in the segment $l_1 : r_1$ we need to examine the roots of the winner trees that correspond to this segment and determine the least value in these roots. We examine these winner trees right to left. Lemma 2.3 is used to determine the root of the rightmost winner tree. However, since Lemma 2.3 applies only to the case when all segment elements are represented by leaves at the same level, we need to simulate the case when this is not true by an equivalent but hypothetical segment for which it is true. This is done by considering the corresponding extended segment as in Figure 6. The *FindMin* algorithm is given in Figure 7. Since the maximum number of winner trees for a segment of size s is $O(\log s)$ (Lemma 2.2), the complexity of the algorithm *FindMin* is $O(\log s)$. We assume that the *XOR* ($\oplus$) is done in constant time.

3.3. Update(l_1, r_1, i_1, y)

To update element i_1 of segment $l_1 : r_1$ we need to replay the tournaments played by element i in its winner tree. I.e., we need to follow the path from element i to the

```
procedure FindMin(l_1, r_1, x);
    /* The minimum value in the segment
       l_1 : r_1 is returned in x */
begin
    l :=Position(l_1);
    r :=Position(r_1);
    x := t[r];

    /* convert two level case to extended segment */
    if (l > r) then r := 2 * r + 1;

    /* examine winner tree roots */
    while r ≥ l do
    begin
        k := min{⌊log_2(r ⊕ (r + 1))⌋, ⌊log_2(r − l + 1)⌋}   /* h − 1 */
        root := r div 2^k;
        x := min{x, t[root]};
        r := r − 2^k;    /* right of remaining segment */
    end;
end;
```

Figure 7: Find minimum value in segment $l_1 : r_1$

root of its winner tree. The procedure for this is given in Figure 8.

In this procedure l, r, and i are initialized to the positions of l_1, r_1, and i_1, respectively and then $t[i]$ is updated to its new value y. To restructure the winner tree that contains element i_1, variable i will move up the tree resetting values as needed. In order to handle the cases when the leaves are on two levels the same as when they are on one, we again use the concept of an extended segment. In case the leaves are on two levels, r is changed to $2 \times r + 1$ (line 6) to correspond to the right end of the extended segment. The variable a gives the number of elements (or extended elements) in the winner tree with root i. When l and i are on different levels, i has two extended elements below it. Otherwise subtree i has just one element. $left$ and $right$, respectively, give the positions of the leftmost and rightmost elements (or extended elements) in the subtree with root i. These variables are initialized to conform to this definition in lines $7 - 17$. In lines 19 and 20 $left$ and $right$ are updated to correspond to the positions of the leftmost and rightmost elements (or extended elements) in the subtree whose root is the parent of i.

Now we are ready to move up the tree containing i. The parent of i is a member of a winner tree iff all its elements (extended elements) are from the same segment. I.e., iff $left \geq l$ and $right \leq r$. So long as this is true, we move up to i's parent. This is done in the **while** loop of lines $21 - 31$. If i is odd then its sibling is $i - 1$ and the value in its parent should be $min\{t[i], t[i-1]\}$. When i is even, its sibling is $i + 1$ and its parent should have value $min\{t[i], t[i+1]\}$. As we move up the tree, $left$ and $right$ are updated to maintain the invariant: The (extended) elements in the subtree whose root is the parent of i occupy positions $left, \ldots, right$ (lines $27 - 30$). It is possible to terminate the upward movement of i earlier than in procedure update. In fact, if $t[p]$ is unchanged on any iteration (lines 25 and 26), then it will remain unchanged in future iterations. We have not done this so as to keep the code simple. Since the height of the winner tree containing element i_1 is $O(\log s)$ where $s = r_1 - l_1 + 1$ (Lemma 2.2), the complexity of procedure update is $O(\log s)$.

3.4. Merge(l_1, r_1, r_2)

This is easily accomplished by the invocation of $Update(l_1, r_2, r_1, t[Position(r_1)])$. A slightly more efficient procedure results by beginning the update **while** loop (lines $21 - 31$) at the root of the rightmost winner tree of the first segment rather than at a leaf node. This is done in procedure *Merge* of Figure 9. In this procedure, l and r, respectively, denote the positions of the first and last elements of the first segment. b and c are the corresponding variables for the second segment. The variables a, $left$, and $right$ have the same significance as in procedure *Update*. In lines $8 - 13$, i is set to the root of the rightmost winner tree of the first segment and the variables a, $left$, and $right$ appropriately initialized. The following **while** loop is identical to that of procedure *Update*. The complexity of this procedure is readily seen to be $O(\log s)$ where s is the size of the new segment that results from the merge.

```
procedure Update(l_1, r_1, i_1, y);
   /* Update element i_1 of l_1 : r_1 to y */
1  begin
2     l :=Position(l_1);
3     r :=Position(r_1);
4     i :=Position(i_1);
5     t[i] := y;
6     if (l > r) then r := 2 * r + 1; /* leaves on two levels */
7     if (l > i)
8     then
9     begin
10       /* l and i on two levels */
11       a := 2; left := 2 * i; right := left + 1;
12    end
13    else
14    begin
15       /* l and i on same level */
16       a := 1; left := i; right := i;
17    end;
18    if i is odd
19       then left := left - a
20       else right := right + a;
21    while (left ≥ l) and (right ≤ r) do
22    begin
23       p := i div 2; a := 2 * a;
24       if i is odd
25          then t[p] := min{t[i], t[i - 1]}
26          else t[p] := min{t[i], t[i + 1]};
27       i := p;
28       if i is odd
29          then left := left - a
30          else right := right + a;
31    end;
32 end;
```

Figure 8: Updating of the element i_1 in segment $l_1, \ldots, r_1$

```
procedure Merge(l1, r1, r2);
   /* Merge the segments l1 : r1 and r1 + 1 : r2 */
1  begin
2     l :=Position(l1);
3     r :=Position(r1);
4     b :=Position(r1 + 1);
5     c :=Position(r2);
6     if (l > c) then c := 2 * c + 1; /* two levels */
7     /* find the rightmost tree in first segment */
8     k := min{⌊log2(r ⊕ (r + 1))⌋, ⌊log2(r − l + 1)⌋}; /* h − 1 */
9     a := 2^k;
10    i := r div a; left := i * a; right := (i + 1) * a − 1;
11    if i is odd
12       then left := left − a
13       else right := right + a;
14    while (left ≥ l) and (right ≤ c) do
15    begin
16       p := i div 2; a := 2 * a;
17       if i is odd
18          then t[p] := min{t[i], t[i − 1]}
19          else t[p] := min{t[i], t[i + 1]};
20       i := p;
21       if i is odd
22          then left := left − a
23          else right := right + a;
24    end;
25 end;
```

Figure 9: Merge two segments

```
procedure Add(l1, r1, x);
    /* add x to the end of the segment l1 : r1 */
begin
    r1 := r1 + 1;
    l :=Position(l1); r :=Position(r1);
    t[r] := x; i := r;
    if (l > r)
    then
    begin
        a := 2; left := 2 * r;
    end
    else
    begin
        a := 1; left := r;
    end;
    left := left − a;
    while (left ≥ l) and i is odd do
    begin
        p := i div 2; a := 2 * a;
        t[p] := min{t[i], t[i − 1]};
        i := p; left := left − a;
    end;
end;
```

Figure 10: Procedure to add an element to a segment

3.5. Add(l_1, r_1, x)

This is equivalent to the code :

$r_1 := r_1 + 1;$
$Update(l_1, r_1, r_1, x);$

As in the case of the merge operation one can obtain a slightly more efficient implementation by customizing the code. The result is procedure *Add* of Figure 10. Its complexity is $O(\log s)$ where $s = r_1 - l_1 + 1$.

3.6. Split(l_1, r_1, q)

To split a segment no work is to be done. This procedure is a null procedure and its complexity is $O(1)$.

n	*Leftist Tree*		*Fibonacci Heap*			*Seg. Winner Tree*		
	merge	deletemin	merge	deletemin	delete	merge	delete	deletemin
10000	38	204	16	4735	3741	64	71	160
25000	57	234	16	5245	4280	64	75	163
50000	76	245	16	5560	4645	67	77	171
100000	93	280	16	5919	4921	67	78	168

Table 2: Run time comparison

4. Experimental Results

The asymptotic complexity of segmented winners tree was compared to that of the competing data structures: leftist trees and Fibonacci heaps in Table 1. Because of the need to perform updates and splits in the targeted applications, segmented winner trees are expected to generally outperform leftist trees and Fibonacci heaps.

To demonstrate the practicality of segmented winner trees, we benchmarked this data structure against the other two for operations on which these other two structures are known to be very practical. Specifically, we considered the merge and deletemin operations for leftist trees and the merge, deletemin, and delete operations for Fibonacci heaps. For segmented winner trees, the deletemins are accomplished by first finding the minimum element in the segment and then updating this to to have a large value. A delete operation is simply an update to a large value.

The code for the test operations was written in *C* and run on a SUN SPARCstation SLC. Run times were obtained for n = 10,000, 25,000, 50,000 and 100,000. For each value of n we generated n random values. These formed the n initial segments of size one. Random pairs of adjacent segments were merged until only one segment remained. The time for the last 1000 of these merges was measured and averaged. This experiment was repeated twenty times and the average of the averages obtained. This is reported in Table 2 in the columns labeled "merge". The times are in microseconds. The Fibonacci heap has the best merge times.

For deletemins, we started with the structures that resulted from the n merges and performed 1000 deletemins. The average time for these was computed and the experiment repeated twenty times. The average of the averages is reported in Table 2 in the columns labeled "deletemin". The segmented winner tree outperformed the other two data structures.

The experiments for the "delete" operation were performed in an analogous manner and the average of the averages reported in Table 2. Once again, the segmented winner tree exhibited superior performance.

5. Conclusion

We have developed an efficient data structure to perform find min, update, merge, add, and split operations in segmented data (i.e., data partitioned into segments of adjacent elements). This structure is very practical both in terms of run times and space utilization. It was pointed out that leftist trees and Fibonacci heaps do not support all the required operations efficiently. In addition, the space requirements of

leftist trees and Fibonacci heaps are considerably greater.

Acknowlegement

The authors would like to thank the referee for providing insightful comments.

References

[1] *Lim, A., S. Cheng, S. Sahni:* Optimal Joining of Compacted Cells. IEEE transactions on Computers **42** 3, pp. 597-607.

[2] *Horowitz, E., S. Sahni:* Fundamentals of Data Structures in Pascal. Computer Science Press, 3rd ed., 1990.

[3] *Fredman, M., R. Tarjan:* Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithm. Journal of ACM **34** (1987) 3, pp. 596-615.

(Received: October 13, 1993; revised version: April 6, 1994)

Authors' addresses:

Andrew Lim
Information Technological Institute
National Computer Board
71 Science Park Drive
Singapore 0511
Republic of Singapore
E-mail: `alim@iti.gov.sg`

Sartaj Sahni
Department of Computer and Information Sciences
University of Florida
Gainesville
Florida 32611
U.S.A.