# New Designs for Priority Queue Machine

M. R. Meybodi

Computer Science Department
Ohio University

**Abstract**

A priority queue is a device which stores a set of elements and their associated priorities and provides a set of operations on these elements, called priority queue operations. The standard operations on priority queue are *XMAX* and *INSERT*. *XMAX* operation retrieves and deletes the element with the highest priority and *INSERT* operation inserts an element and its associated priority into the priority queue. A number of multiprocessor designs for maintaining priority queue have been proposed in the literatures. These designs can be classified into two main groups: 1) designs with many processors each having an small amount of memory, and 2) designs with small number of processors each having a large amount of memory. In this paper two new designs, both belonging to the second group, for performing a group of priority queue operations on a set of element are presented. Processors in either design are connected together to form a linear chain. The first machine, called binary heap machine, is based on heap data structure. The algorithms for XMAX and INSERT operations are different from their sequential counterparts in that they both perform the restructuring process from the top. The second machine is called banyan heap machine. The algorithms for this machine are the generalization of heap algorithms to a more general acyclic graph called banyan. This design requires fewer processors than the heap machine in order to meet the same capacity requirement and unlike some of the existing designs, processors do not have geometrically varying memory sizes, resulting in a completely homogeneous system. The key advantage of banyan heap machine is that it is possible to retrieve elements at different percentile levels.

## 1 Introduction

Special classes of computational tasks have led to the development and realization of special purpose systems that most efficiently perform the given tasks. One class of these special purpose architectures is characterized by strong, inherent parallelism functioning in a systolic manner. A systolic system is a network of processors which rhythmically computes and passes data through the system [11]. In systolic systems, processors are typically interconnected in a such a way which results in a simple, regular communication and control structure. Data flows between processors in a piplined fashion and communicate with the outside world only at the boundary processors. Systolic architecture is mainly used to design algorithms and data structures for direct layout in integrated circuits. Systolic architectures for variety of computational tasks such as: signal and image processing, matrix arithmetic, graph algorithms, relational database operations have been proposed [4,8,9,10]. This architecture was originally proposed for VLSI implementation of some matrix operations [4]. In this paper we will consider two systolic designs for a priority queue data structure.

Priority queue is a very important data structure which has found applications in varieties of situations [18,19,21]. This data structure is a set of elements each of which has an associated number, its priority. For each element $x, p(x)$, the priority of $x$ is a number from some linearly ordered set. Standard operations on a priority queue are *INSERT*, which inserts an element and its associated priority into the priority queue, and *XMAX*, which deletes the element with the highest priority from the queue. Let P denote the set of all element- priority pairs. Define

$$P(s) = \{(x, p(x)) | p(x) = s \text{ and } (x, p(x)) \in P\};$$

The effect of priority queue operations are as follows:

*INSERT*$(x, p(x))$ :

$P \leftarrow P \bigcup \{(x, p(x))\}$ .
Response is null.

*XMAX:*

$P \leftarrow P - P(p_{max})$ where $P(p_{max})$ is the pair with the highest priority.
Response is $P(p_{max})$.

A priority queue machine receives a stream of operations *(INSERT,XMAX)*, execute them in a piplined manner, and, in the case of *XMAX* operation, reports the element with the highest priority via the *I/O* port. The *response time* for an operation is the time elapsed between the initiation and completion of an operation, and the *pipeline interval* of an operation is the minimum time needed before the initiation of the next operation. The *period* of the machine is the maximum of all operation pipeline intervals.

Hardware realization of data structures have been investigated by several researchers. Leiserson [2] proposed a machine to implement priority queue operations and Bentley and Kung [1] have given an implementation of dictionary operations on a tree in which the data elements are stored in the leaves of the tree. Using X-trees, Ottmann et al. [3] designed a more efficient implementation of dictionary operations at the expense of additional wires. Atallah and Kosaraju [7] and Somani and Agarwal [5,6] have shown that dictionary operations can be implemented on a tree which does not use any links other than the binary tree links. Schmeck and Schroders [12], and Dehne and Santoro [22,24] have given an implementation of dictionary operations on mesh- connected array. Recently, J. H. Chang, O. H. Ibarra, M. J. Chung and K. Rao [25] have proposed systolic tree architectures for data structures such as stacks, queues, dequeues, priority queues, and dictionary machines. Cray and Thompson [15], Fisher [13], and Tanaka, Nozaka, and Masuyama [16] have shown that a dictionary machine can be constructed using search trees implemented on a linear array of processors. For related litratures refer to [14,17,26-30].

In this paper, We propose two new designs for performing priority queue operations. The first machine, called binary heap machine, is based on heap data structure. The algorithms for priority queue operations are different from their sequential counterparts in that they both perform the restructuring process from the top. The second machine is called banyan heap machine. The algorithms for this machine are the generalization of heap algorithms to a more general acyclic graph called banyan. This design requires fewer processors than the existing designs [1-7,12,22,24] in order to meet the same capacity requirement and unlike some of the existing designs [15,16], processors do not have geometrically varying memory sizes resulting in a completely homogeneous system. The key advantage of banyan heap machine is in its ability to retrieve elements at different percentile levels. The control mechanism is identical for all the processors excepts those at the first and last levels. The input and output are done through the root in the binary heap machine and through the leftmost processor of the first level in the banyan heap machine which makes them suitable for VLSI implementation.

The rest of this report is organized as follows. Section 2 presents binary heap machine. Section 3 first defines banyan graphs and banyan heaps, and then discuss a priority queue machine based on banyan heap. Section 4 derives analytical formulas for percentile level of a retrieved element. The last section is the conclusion.

## 2 Proposed Design

### 2.1 Binary Heap Machine

This machine is a linear array of log N processors, one for each level of the heap. See figure 1. Processor $p_i$ , $(1 \leq i \leq \log N)$, stores the $i^th$ level of the heap. Each node has 6 fields: DATA, PRIORITY, LCHILD, RCHILD,LEMPTYNODES, and REMPTYNODES. For a node, the DATA field holds an element and the PRIORITY field holds the priority associated with that element, LCHILD and RCHILD

CH2728-4/90/0000/0123$01.00 © 1990 IEEE

holds respectively pointers to the left child and right child of that node, and LEMPTYNODES and REMPTYNODES hold the number of null nodes (nodes with no information) in the left and right subtrees of that nodes. Initially, the DATA field of all the nodes are set to null and the PRIORITY field of all the nodes are set to $-1$. The LEMPTYNODES and REMPTYNODES fields of all the nodes at level i are initialized to $2^{\log N - i} - 1$. These two fields are updated as data elements are inserted into or deleted from the machine. Information about the number of empty nodes is used by the *INSERT* operation to decide which path in the heap should be followed during the insertion process. Lack of such information may lead to an overflow situation in the last processor. This happens if the *INSERT* operation moves along a path in which all the nodes are non-empty.

The instruction set for each processor includes compare, move, add, subtract, branch, compare, to read and write into the local memory, and send and receive to communicate with the neighboring processors. Receive instruction is of blocking type, that is, it is not complete until a message is received from the specified processor. Now we describe operations *XMAX* and *INSERT* for this machine.

*XMAX* operation: Operation *XMAX* generates an operation called *ADJUST* which propagates through the chain of processors and converts the binary tree (after the root is removed) into a heap. With respect to operation *XMAX*, all the processors except processor $p_1$ perform the same set of actions. We describe operation *XMAX* as follows:

**processor** $p_1$: When processor $p_1$ receives operation *XMAX*, it reports the element with the highest priority to the outside world and then sends operation ADJUST to processor $p_2$.

**processor** $p_i$, $(1 < i < \log N)$ : Let p be the address of the node in the local memory of processor $p_{i-1}$ whose value had been moved up or reported by processor $p_{i-1}$ in the previous step. On receiving operation *adjust(p)* by $p_i$, it finds the child of node p, q, that contains the element with higher priority, sends DATA(q) to $p_{i-1}$ to replace DATA(p), and next sends *adjust*(q) to processor $p_{i+1}$. Processor $p_{i-1}$, $(i > 1)$ , after filling up the node p with DATA(q), increases the LEMPTYNODES or REMPTYNODES field of node p by one, depending on whether q has been the address of left or right child of node p. If both children of node p are empty then processor $p_i$ will not generate any more *adjust* operations; it only sends a message to processor $p_{i-1}$ asking to empty node p. No processor will receive an adjust operation until the most recent adjust operation issued by that processor is completely executed by its neighboring processor.

*INSERT* operation: When processor $p_i$ , $(1 \le i \le \log N)$, receives operation *INSERT*(p,item) it performs the following actions. It first compares the priority of DATA(p) with the priority of *item*, if the priority of *item* is greater than that of DATA(p) it replaces DATA(p) by *item* and then issues *INSERT*(q,DATA(p)) to processor $p_{i+1}$. If the priority of *item* is less than that of DATA(p), then processor $p_i$ only sends *INSERT*(q,item) to $p_{i+1}$ . Q is the address of the right child of node p if LEMPTYNODES(p)< REMPTYNODES(p) and is the address of the left child of node p if LEMPTYNODES(p) > REMPTYNODES(p). Processor $p_i$ also decreases the LEMPTYNODES(p) or REMPTYNODES(p) by one, depending on whether the *item* will be inserted into the right (q=RCHILD(p)) or left (q=LCHILD(p)) subtree of node p. *INSERT* operation will not be propagated further down if node p is null node which in that case the only action performed by processor $p_i$ is storing the element in DATA(p).

Like all the priority queue machines reported in the literature, this machine also receives a stream of *INSERT* and *XMAX* operations and executes them in a piplined manner. The response time for *XMAX*, and pipeline period for both the XMAX and *INSERT* operations is $O(1)$, independent of the length of the array processors . However, it takes $O(\log N)$ time for each of the *XMAX* or *INSERT* operation to be executed. This is due the fact that it may take $O(\log N)$ time for an element to find its correct place within the heap.

## 3 Banyan Heap Machine

Like the binary heap machine, the processors in the banyan heap machine are connected together to form a linear array, but unlike the binary heap machine they don't have geometrically varying size memory. This leads to a completely homogeneous system. The algorithms for XMAX and INSERT operations are the extension of those for the first machine to more general acyclic graphs called banyan.

In the following section we first describe banyan graphs and then define different classes of banyan heap.

### 3.1 Banyan graphs and banyan heaps

A banyan graph is a Hasse diagram [23] of a partial ordering in which there is only one path from any base to any apex. A base is defined as any vertex with no arcs incident out of it and an apex is defined as any vertex with no arcs incident into it. A vertex that is neither an apex nor a base vertex is called an intermediate vertex.

An L-level banyan is a banyan in which the path from base to apex(or apex to base) is of length L. Therefore, in an L-level banyan, there are $L + 1$ level of nodes and L-level of edges. By convention, apexes are considered to be at level 0 and bases at level L. In a banyan graph, the outdegree and the indegree of a node are called spread and fanout of that node. If there is an edge between two nodes, $x$ at level $i$ and y at level $i + 1$, then we say $x$ is the parent of $y$, and $y$ is the child of $x$.

**Definition 1** *A banyan is called a uniform banyan if all the nodes within the same level have identical spread and fanout.*

In a uniform banyan , the fanout and spread values may be characterized by L component vectors, $F = (f_0, f_1, ..., f_{L-1})$ and $S = (s_1, s_2, ..., s_L)$, the fanout vector and spread vector, respectively, where $s_i$ and $f_i$ denotes the spread and fanout of a node at level $i$.

**Definition 2** *If $s_{i+1} = f_i$, $(0 \le i \le L-1)$, that is $F = S$, then the banyan is called rectangular. If $s_{i+1} \ne s_i$ for some $i$, then the banyan is non-rectangular.*

**Definition 3** *A banyan is said to be regular if $s_i = s$, $(1 \le i \le L)$, and $f_i = f$, $(0 \le i \le L-1)$, for some constant $s$ and $f$. Otherwise it is said to be irregular.*

**Definition 4** *A banyan is an SW- banyan if it has the following two additional properties: a) Two nodes at an intermediate level $i$, have either no or all common parents at level $i-1$. b) two nodes at intermediate level $i$ have either no or all common children at level $i+1$.*

**Definition 5** *An SW-banyan is said to be rectangular if it is regular and $s_i = d$, $(1 \le i \le L)$, and $f_i = d$, $(1 \le i \le L-1)$, for some constant $d$.*

Now we define Banyan heap.

**Definition 6** *An L-level banyan heap is an L-level banyan such that the priority of the element at each node is equal or greater than the priorities of the elements at each of its children.*

Figure 2 shows an example of 4-level rectangular banyan heap. In this report we study the implementation of MxM rectangular SW-banyan heap with $d = 2$ in an array of $\log M + 1$ processors. M is the number of apexes. In such banyans the number of levels is $\log M + 1$, each of which assigned to one processor with the apexes assigned to processor $p_1$. See figure 3. Each node in the banyan heap has six fields: DATA, PRIORITY, LCHILD, RCHILD, LEMPTYNODES, and REMPTYNODES. In addition to the above six fields, each apex has another field called NEXT. This field is used to link apexes together. Initially, the DATA field of all the nodes are set to null and the priority field of all the nodes are set to $-1$. To initialize these fields, first partition the heap into N disjoint binary trees and then use the same rule as for the first design to initialize the LEMPTYNODES and REMPTYNODES fields of the nodes in each partition. The partitioning process starts with the leftmost apex and continues in increasing order of the apex numbers. The leftmost apex is numbered 1. Partition $i$ is the set of all nodes which are reachable from apex $i$ by moving down the heap and are not part of partition $i-1$. The root of the binary tree in partition i is apex $i$. The depth of a partition is the depth of the corresponding binary tree. The set of partitions and the initial settings of LEMPTYNODES and REMPTYNODES fields for an 8x8 SW- banyan is given in figure 4.

**Definition 7** *An L-level partitioned banyan heap is an L-level banyan such that each partition of the banyan (as defined above) is a binary heap.*

**Definition 8** *A partitioned L-level banyan heap is said to be full up to apex d if all the nodes in partitions $j$, $(j < d)$, are non-null and the nodes in the remaining partitions are null.*

**Definition 9** *A node in a banyan heap is said to be reachable by partition from apex $i$ if its parent is reachable by partition from apex $i$. Node $l$ at level j+1 is reachable by partition from node $k$ at level $j$ if node $l$ either has non-zero REMPTYNODES and it is the right child of node l,or has non-zero LEMPTYNODES and it is the left child of node l. An apex is reachable by partition from itself.*

**Remark 1** The null nodes which are reachable by partition from a given apex will be filled up by insertions initiated at that apex unless the reachablity of the nodes will change by a later deletion operation initiated at some other apexes. Reachablity does not imply reachablity by partition.

Each processor in the array is equipped with send and receive instructions. They are used for communication between neighboring processors. Send(<processor>, '<instruction>') sends instruction <instruction> to processor <processor> for execution. The execution of receive(<processor>,(information)) causes the information specified by the second argument be obtained from processor <processor> and forwarded to the requesting processor (the processor executing the receive instruction). A receive instruction executed by processor $p_i$ is not complete until a message is received from processor $p_{i+1}$.

In the next section we describe the implementation of priority queue operations for partitioned banyan heap. Implementation of priority queue operations for banyan heap is reported in [31].

### 3.2 Algorithms for Partitioned Banyan Heap

Insertion into a partitioned banyan heap is performed by operation *INSERT*. This operation, executed by processor $p_1$, first finds the leftmost partition which has at least one empty node. This can be done by checking the REMPTYNODES and LEMPTYNODES of the apexes. It then pushes the element requested to be inserted down the banyan heap using operation *insert- adjust*. The operation *insert-adjust* pushes down the element (along the paths from the root of the partition to the bases) until it finds its correct position.

Upon receiving *INSERT*(p,(item,priority)) by processor $p_1$, it executes the following codes. P is the address of the leftmost apex and (item, priority) is the pair requested to be inserted.

```
found ← false
While (not found ) do
  if DATA(p) ≠ null then
    if priority > PRIORITY(p)
      begin
        if LEMPTYNODES(p)≠ 0 or REMPTYNODES(p)≠ 0 then
          begin
            if LEMPTYNODES(p) > REMPTYNODES(p) then
              begin
                p'← RCHILD(p);
                REMPTYNODES(p) ← REMPTYNODES(p) -1
              end
            else
              begin
                p'← LCHILD(p);
                LEMPTYNODES(p)← LEMPTYNODES(p) -1
              end
            send(P2 , 'insert-adjust(p', DATA(p)));
            DATA(p) ←item; PRIORITY(p)← priority;
            found ← true
          end
        else
          p ← NEXT(p)
      end
    else
      begin ( priority < PRIORITY(p) )
        if LEMPTYNODES(p) ≠ 0 or REMPTYNODES(p)≠ 0 then
          begin
            if LEMPTYNODES(p) > REMPTYNODES(p) then
              begin
                p'← LCHILD(p);
                LEMPTYNODES(p) ← LEMPTYNODES(p) -1
              end
            else
              begin
                p' ← RCHILD(p);
                REMPTYNODES(p)← REMPTYNODES(p) -1
              end
            send(p2, 'insert-adjust(p', (item,priority))');
            found ← true
          end
        else
          p ← NEXT(p)
    else
      begin
        DATA(p) ← item;
        PRIORITY(p) ← priority
      end;
```

Processor $P_i$, $(2 \leq i \leq L)$, upon receiving *insert-adjust*(p,(item,priority)) executes the following codes.

```
if DATA(p) ≠ null then
  if priority > PRIORITY(p) then
    begin
      if LEMPTYNODES(p) ≠ 0 or REMPTYNODES(p) ≠ 0 then
        begin
          if LEMPTYNODES(p) > REMPTYNODES(p) then
            begin
              p'← LCHILD(p) ;
              LEMPTYNODES(p) ←LEMPTYNODES(p)-1
            end
          else
            begin
              p' ← RCHILD(p);
              REMPTYNODES(p) ← LEMPTYNODES(p) -1
            end
        end
        send(pi+1,'insert-adjust(p',(DATA(p),PRIORITY(p));
        DATA(p) ←item; PRIORITY(p) ← priority
    end
  else
    begin
      if LEMPTYNODES(p) ≠ 0 or REMPTYNODES(p) ≠ 0 then
        begin
          if LEMPTYNODES(p) > REMPTYNODES(p) then
            begin
              p' ← RCHILD(p);
              REMPTYNODES(p) ← REMPTYNODES(p) -1
            end
          else
            begin
              p'← LCHILD(p);
              LEMPTYNODES(p) ← LEMPTYNODES(p) -1
            end
          send(pi+1, 'insert-adjust(p',(item,priority))');
          end
      else
        begin
          DATA(p) ← item; PRIORITY(p) ← priority
        end
else
  begin
    DATA(p) ← item;
    PRIORITY(p) ← priority
  end;
```

*XMAX* operation first locates the apex which contains the element with the highest priority, reports that element to the outside world, and then fills up that apex with the element in one of its children. *xmax-adjust* is responsible for restructuring the banyan as it moves down the heap. When *XMAX*(p) is received by processor $p_1$, it executes the following codes. *P* is the address of the leftmost apex. This address is known to the outside world (front end computer).

```
p' ← p
while NEXT(p) ≠ nil and DATA(NEXT(p)) ≠ null do
  begin
    if PRIORITY(p) > PRIORITY(NEXT(p)) then p' ← p
    p ← NEXT(p)
  end
  send('outside world', DATA(p'));
  receive (p2, ((PRIORITY(RCHILD(p')),DATA(RCHILD(p'))),
                    (PRIORITY(LCHILD(p')),DATA(LCHILD(p')))) )
  if DATA(RCHILD(p')) ≠ null or DATA(LCHILD(p'))≠ null then
    if DATA(RCHILD(p')) > DATA(LCHILD(p')) then
      begin
        DATA(p') ← DATA(RCHILD(p'));
        PRIORITY(p') ← PRIORITY(RCHILD(p'));
        REMPTYNODES(p') ← REMPTYNODES(p') + 1;
        send(p2, 'xmax-adjust(RCHILD(p')')
      end
    else
      begin
        DATA(p') ← DATA(LCHILD(p'));
        PRIORITY(p') ← PRIORITY(LCHILD(p'));
        LEMPTYNODES(p') ← LEMPTYNODES(p') + 1;
        send (p2, 'xmax-adjust(LCHILD(p')')
      end
  else
```

```
begin
   DATA(p') ← null;
   PRIORITY(p') ← −1
end
```

Processor $p_i$, $(2 \leq i \leq L)$, upon receiving *xmax- adjust*(p)) executes the following codes.

```
receive(p_{i+1}, ((PRIORITY(RCHILD(p)),DATA(RCHILD(p))),
                 (PRIORITY(LCHILD(p)),DATA(LCHILD(p)))) );
if DATA(RCHILD(p)) ≠ null or DATA(LCHILD(p)) ≠ null then
   if DATA(RCHILD(p)) > DATA(LCHILD(p)) then
      begin
         DATA(p) ← DATA(RCHILD(p));
         PRIORITY(p) ← PRIORITY(RCHILD(p));
         REMPTYNODES(p) ← REMPTYNODES(p) + 1;
         send(p_{i+1}, 'xmax-adjust(RCHILD(p)')
      end
   else
      begin
         DATA(p) ← DATA(LCHILD(p));
         PRIORITY(p) ← PRIORITY(LCHILD(p));
         LEMPTYNODES(p) ← PRIORITY(p) + 1;
         send(p_{i+1}, 'xmax-adjust(LCHILD(p)')
      end
else
   begin
      DATA(p) ← null;
      PRIORITY(p) ← −1
   end
```

**Remark 2** The elements stored in the apex nodes are not ranked in any particular order. This speeds up the insertion process, but will lead to O(M) time for deletion. It is possible to insert the elements in such a way that the element with the highest priority is always available at the leftmost apex, in this case, locating the correct apex to initiate the insertion takes O(M) time. This method seems to be more efficient because a portion of the time spent to find the correct position can be overlapped with the time spent to locate an apex with zero REMPTYNODES or zero LEMPTYNODES. In the algorithms presented above we have used the first approach. The latter approach will be reported in another report.

From the properties of SW-banyan graphs and the above algorithms we can state the following results.

**Lemma 1** *a) The insert-adjust operation never encounters a node which is non-null and has zero LEMPTYNODES and zero REMPTYNODES. b) The insert-adjust operation always finds a null node to insert its element.*

**Proof** a) If operation xmax-adjust encountered a non- null node with LEMPTYNODES and REMPTYNODES fields equal to zero then it must have been initiated from an apex with both REMPTYNODES and LEMPTYNODES equal to zero. This is impossible according to the algorithm for INSERT. b) Proof is immediate from the proof of part a and the definitions of LEMPTYNODES and REMPTYNODES.

**Remark 3** Deletion of an element from partition i may cause one of the elements in other partitions whose nodes are reachable from apex $i$ to become null. This happens if a delete operation causes the xmax-adjust, on its way down the heap, to move up the content of one of the leaf nodes of partition i to fill up its parent which has been emptied by xmax- adjust operation at the previous step. The emptiness of this node now will be reflected in the REMPTYNODES or LEMPTYNODES of apex i. This node is now reachable by partition from apex i and will be filled by an insertion initiated at apex i. The maximum number of nodes that may become reachable by partition from apex i as a result of a deletion is equal to (L+1) - L' where L' is the depth of partition i.

**Lemma 2** *Zero REMPTYNODES and zero REMPTYNODES for an apex does not imply that all the nodes in the corresponding partition are non- null.*

**Proof** From remark 3.

**Lemma 3** *Apex i, $(1 \leq i \leq N)$, always contains the element which has the highest priority among the elements stored in the nodes of partition i.*

**Proof** From algorithms for *XMAX*, *xmax-adjust*, *INSERT*, and *insert-adjust.*

**Lemma 4** *The element with the highest priority is always reported by operation* XMAX.

**Proof:** From lemma 4 and the first part of algorithm for *XMAX*.

**Definition 10** *A partition induced by LEMPTYNODES and REMPTYNODES fields of apex i is the set of all nodes which are reachable by partition from apex i.*

## 4 Retrieval at Percentile Levels

One of the most important advantages of banyan heap machine over the binary heap machine and Tanaka's machine is that it is possible to retrieve elements at different percentile levels. In this section we derive formulas for the percentile level of the element reported by operation *XMAX* for different cases.

**Definition 11** *An element removed from a banyan heap is at percentile c if at at least c percent of the elements stored in the heap have priority less than or equal to the priority of the deleted element.*

We define $REMPTYNODES_i$ and $LEMPTYNODES_i$ to denote respectively the value of REMPTYNODES field and LEMPTYNODES field of apex i. The proof of the following 4 lemmas are immediate from the definitions of REMPTYNODES and LEMPTYNODES.

**Lemma 5** *The total number of null nodes which are reachable by partition from apex i is $REMPTYNODES_i + LEMPTYNODES_i$.*

**Lemma 6** *If an MxM partitioned rectangular SW-banyan banyan heap is full up to apex d then*

$$\sum_{j=1}^{d}(REMPTYNODES_j + LEMPTYNODES_j) = 0.$$

**Lemma 7** *In an MxM rectangular SW-banyan, the total number of null nodes reachable by partition from apexes 1 through d, written NULLNODES(M,d), is given by:*

$$NULLNODES(M,d) = \sum_{i=1}^{d}(REMPTYNODES_i + LEMPTYNODES_i) + K.$$

*where K is the number of null apexes i, (i≤ d).*

**Lemma 8** *The total number of non-null nodes in a MxM partitioned rectangular SW-banyan, written NONNULLNODES(M,M), is* $M(\log M+1) - NULLNODES(M,$

**Lemma 9** *The total number of partitions of depth k in a full partitioned banyan heap up to apex d, written $NP_k$, is given by:*

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor$$

*where* $NP_1 = \lfloor \frac{d}{2} \rfloor$.

**Proof.** We prove this lemma by induction on d.

Basis: For $d = 1$, $NP_j = 0$ for all $1 \leq j \leq \log N + 1$.

Induction: Assume that

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

Consider d+1. Either d is even or it is odd. If d is even then apex d+1 is the root of a partition with depth 1. Therefore,

$$NP_k = \lfloor \frac{d + 1 - \sum_{j=2}^{k-1} NP_j - (NP_1 + 1)}{2} \rfloor,$$

that is,

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

For the case that d is odd we consider two subcases: Apex $d+1$ is either the root of a partition of depth k or it is the root of a partition with depth $h$, $(h < k)$. If $d+1$ is the root of a partition of depth d then we will have

$$NP_k + 1 = \lfloor \frac{d + 1 - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

Since d is even then we have

$$NP_k + 1 = 1 + \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor,$$

or

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

If apex $d+1$ is the root of a partition with depth $h$ where $1 \leq h < \log N + 1$ and $h \neq k$ then we have

$$NP_k = \lfloor \frac{d + 1 - \sum_{1 \le j \le k-1, j \ne k} NP_j - (NP_k + 1)}{2} \rfloor,$$

or

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

**Lemma 10** *The total number of non-null nodes in a full MxM partitioned rectangular SW-banyan up to apex d, written size(M,d), is given by:*

$$size(M, d) = \sum_{k=0}^{M \log M} 2^k + \sum_{j=1}^{d} NP_j 2^j$$

**Proof** From lemma 9.

**Lemma 11** *If an MxM rectangular SW-banyan partitioned heap is full up to apex d then the element stored at apex 1 is at percentile level*

$$\frac{(2M - 1) * 100}{size(M, d)}.$$

**Proof** $size(M, d)$ is the total number of nodes in an MxM rectangular SW-banyan heap which is full up to apex d and $2M - 1$ is the total number of nodes in partition 1.

**Lemma 12** *In an MxM partitioned rectangular SW-banyan heap which is full up to apex d, if operation XMAX investigates i,(i<d), non-null apexes then the percentile of the reported element is*

$$\frac{size(M, i) * 100}{size(M, d)}.$$

**Proof** From lemma 10.

**Lemma 13** *If operation XMAX examines apexes 1 through d in an MxM rectangular banyan heap then the percentile of the reported element is smaller than or equal to*

$$\frac{size(M, d) * 100}{(size(M, M) - NULLNODES(M, M))}.$$

**Proof** If a banyan heap is full up to apex d then by lemma 12 the percentile of the element reported by operation XMAX is

$$\frac{size(M, d) * 100}{size(M, d)}.$$

By lemma 7, this can be written as

$$\frac{size(M, d) * 100}{(size(M, M) - NULLNODES(M, M))}.$$

But the banyan is not full up to apex d and therefore some of the nodes which are reachable from apexes 1 to d are null. Let F be the total number of such nodes. Therefore the percentile of the element reported by XMAX operation will be

$$\frac{(size(M, d) - F) * 100}{(size(M, M) - NULLNODES(M, M))}.$$

This proves the lemma.

**Remark 4** A partition banyan heap can be converted into a banyan heap by an operation called *adjust*. $M \log M - 2$ *adjust* operations are broadcast by *XMAX* operation when it inserts an element into an empty partition. These *adjust* operations cause some the elements in the nodes of those partitions which are reachable from apex $i$ to move up and fill up the nodes of partition $i$. As a result, all the nodes whose contents (empty or non-empty) have been moved by *adjust* operation become reachable by partition from apex $i$. It should be noted that some of the *adjust* operation initiated at processor $p_1$ by XMAX operation may not have any effect on the structure of the heap. The advantage of banyan heap over partitioned banyan heap is that it allows a more uniform distribution of data elements among the partition in the heap and leads to a more uniform increase in the percentile level of the reported element as the number of examined apexes is increased. Algorithms for *XMAX* , *xmax-adjust* and *insert-adjust* are the same as partitioned banyan heap. The operation *INSERT* and the new operation *adjust* are described in details in [31].

## 5 Conclusion

Two new designs for priority queue machine are proposed in this paper. The first machine, called binary heap machine, is a linear array of $logN$ processors, one for each level of the heap. Like all the existing machines for priority queue, this machine also receive a stream of *INSERT* and *XMAX* operations and executes them in a piplined manner. The response time for *XMAX*, and pipeline period for both the *XMAX* and *INSERT* operations is $O(1)$, independent of the length of the array of processors. However, it takes $O(logN)$ time for each of the *XMAX* or *INSERT* operation to be executed. The second design is called banyan heap machine. Processors in the banyan heap machine are connected together to form a linear array, but unlike the binary heap machine they do not have geometrically varying size memory. The algorithms for *XMAX* and *INSERT* operations are the extension of those for the first machine to more general acyclic graphs called banyan. *INSERT* and *XMAX* operations both require $O(M)$ time to be completed where M is the number of apex processors in the banyan heap machine. By using banyan heap instead of heap, it is possible to retrieve elements at different percentile levels.

## 6 References


1. J. L. Bentley and H. T. Kung, "A tree Machine for Searching Problems," Proceeding of the International Conference on Parallel Processing, 1979

2. C. E. Leiserson, "Systolic Priority Queues," Dept. of Computer Science, Carnegie Melon University, Pittsburgh, PA, Report CMU-CS-115,1979.

3. T. A. Ottmann, A. L. Rosenberg, and L.J. Stockmeyer, "A Dictionary Machine for VLSI," IEEE Transaction on Computers, Vol. c-31, No. 9, Sept. 1982, pp. 892-897.

4. H. T. Kung and C. E. Leiserson, " Systolic Arrays (for VLSI), " Proceedings of Symposium on Sparse Matrix Computations and their Applications, Nov. 1978, pp. 256-282.

5. A. K. Somani and V. K. Agarwal, "An Unsorted VLSI Dictionary Machine," Proceedings of 1983 Canadian VLSI Conference, University of Waterloo, Waterloo.

6. A. K. Somani and V. K. Agarwal, "An Efficient VLSI Dictionary Machine, " Proceedings of 11th Annual International Symposium on Computer Architecture, 1985, pp. 142-150.

7. M. J. Atallah and S. R. Kosaraju,"A Generalized Dictionary Machine for VLSI," IEEE transactions on Computers, Vol. C-34, No. 2, Feb. 1985, pp. 151-155.

8. L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," Proceedings of Conference in Very Large Scale Integration: Architecture, Design, Fabrication, California Institute of Technology, Jan. 1979, pp. 509-525.

9. H. T. Kung, "Special Purpose Devices for Signal and Image Processing : An Opportunity in VLSI," Proceedings. SPIE, Vol. 241: Real- Time Signal Processing III, Society of Photo-Optical Instrumentation Engineers, July 1980, pp. 76-84.

10. H. T. Kung and P. L. Lehman, "Systolic Arrays for Relational Operations," Proceedings. ACM-SIGMOD 1980 International Conference on Data, May 1980, pp. 105-116.

11. H. T. Kung, "Why Systolic Architectures?" IEEE Computer Magazine 15(1), Jan. 1982, pp. 37-46.

12. H. Schmeck and H. Schroder, "Dictionary Machines for Different Models of VLSI," IEEE transaction on computers, Vol. C- 34, No. 5, May 1985, pp. 472-475.

13. A. L. Fisher, "Dictionary Machines with Small Number of Processors," Proceedings of International Symposium on Computer Architectures, 1984, pp. 151-156.

14. J. Biswas and J. C., "Simultaneous Update of Priority Structures," Proceedings of International Conference on Parallel Processing, August 1987, pp. 124-131.

15. M. J. Carey and C. D. Thompson, "An efficient Implementation of Search trees on $\lceil \log N + 1 \rceil$ processors," IEEE Transactions on Computers, Vol. C-33, No. 11, Nov. 1984, pp. 1038-1041.

16. C. D. Thompson, "The VLSI Complexity of Sorting," IEEE Transactions on Computers, Vol. C-32, No. 12, Dec. 1983, pp. 373-386.

17. A. R. Omondi and J. D. Brock, "Implementing a Dictionary on Hypercube Machine," Proceedings of International Conference on Parallel Processing, August 1987, pp. 707-709.

18. T. A. Standish, Data Structures Techniques, Addison Wesley, 1980.

19. E. Horowitz and A. Sahni, Fundamentals of Data structures, Computer Science Press, 1983.

20. D. Knuth, The Art of Computer Programming, Vol. 3, 1973.

21. N. J. Nilsson, Problem Solving Methods in Artificial Intelligence, McGraw Hill.

22. F. Dehne and N. Santoro, "Optimal VLSI Dictionary Machines on Meshes," Proceedings of International Conference on Parallel Processing, August 1987, pp. 832-840.

23. L. R. Goke and G. L. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," Proceedings of the First Annual Symposium on Computer Architecture, 1973, pp. 21-28.

24. F. Dehne and N. Santoro, "Optimal VLSI Dictionary Machines on Meshes," Report SCS-TR-106, Carlton University, Jan. 1987.

25. Jik H. Chang, Oscar H. Ibarra, Moon Jung Chung, and Kotesh K. Rao, "Systolic Tree Implementation of Data Structures," IEEE Transactions on Computers, Vol. 37, No 6, June 1988, pp. 727-735.

26. Kam Hoi Cheng, "Efficient Design of Priority Queue," Proceedings of International Conference on Parallel Processing, August 1988, pp. 363-366.

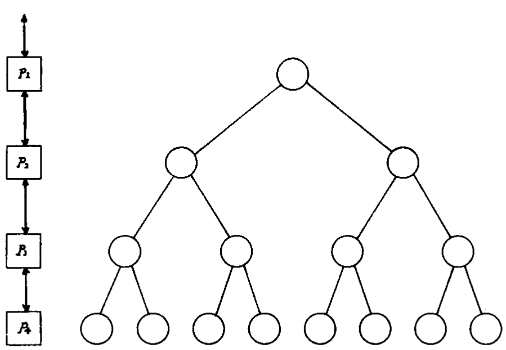
27. M. R. Meybodi, "Tree Structured Dictionary Machines for VLSI," Report CS-1-M87, Ohio University, Jan. 1987.

28. V. Nageshwara Rao and Vipin Kumar, "Concurrent Access of Priority Queues," IEEE Transactions on Computers, Vol. 37, No 12, Dec. 1988, pp. 1657-1665.

29. Douglas W. Jones, "Concurrent Operations on Priority Queues," ACM, Vol. 32 No. 1, Jan. 1989, pp. 132-137.

30. M. R. Meybodi, "Implementing Priority Queue on Hypercube Machine," Report CS-2-M89, Ohio University, Jan. 1989.

31. M. R. Meybodi, "New Designs for Priority Queue Machine," Report CS-1-M89, Ohio University, July 1989.




Binary Heap Machine

FIGURE 1

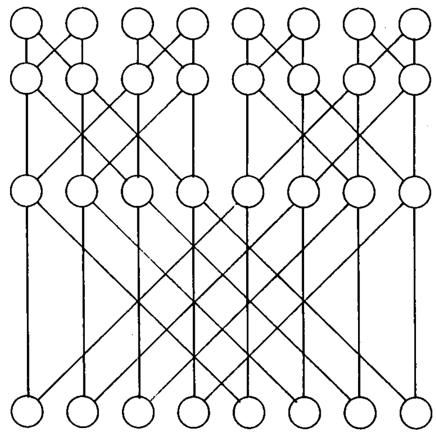
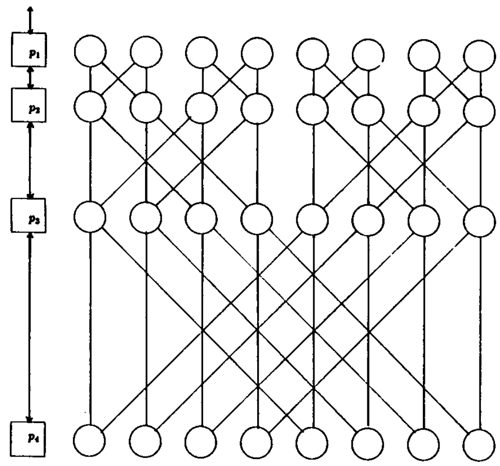


Figure 2




Banyan Heap Machine

Figure 3

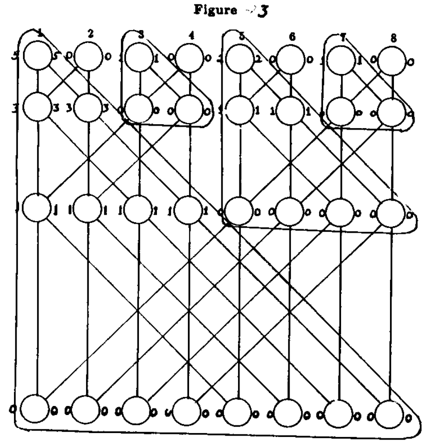


Figure 4