1991

A Mergeable Double-Ended Priority Queue

S. Olariu
Old Dominion University

Z. Wen
Old Dominion University

A Mergeable Double-ended Priority Queue

S. OLARIU, C. M. OVERSTREET AND Z. WEN

Department of Computer Science, Old Dominion University, Norfolk, VA 23529–0162, USA

An implementation of a double-ended priority queue is discussed. This data structure referred to as min–max–pair heap can be built in linear time; the operations Delete-min, Delete-max and Insert take O(log n) time, while Find-min and Find-max run in O(1) time. In contrast to the min-max heaps, it is shown that two min–max–pair heaps can be merged in sublinear time. More precisely, two min–max–pair heaps of sizes n and k can be merged in time O(log (n/k) * log k).

Received November 1990, revised February 1991

1. INTRODUCTION

A priority queue is a data structure whose elements are assigned a label representing their priority. In this context, the natural order of the elements in such a structure is dictated by their respective priority. Priority queues are widely used in software engineering,[7] simulation,[4,7] external sorting[1] and operating systems,[6] to name a few (see Refs. 2 and 6 for relevant discussion).

More formally, a priority queue can be viewed as an abstract data type maintaining a set of keys from a totally ordered universe and supporting the following atomic operations: *Find-max*: find the maximum; *Delete-max*: delete the maximum; *Insert*(x): insert key x into the structure. (Of course, instead of finding or deleting the maximum we could just as well insist on maintaining the structure such that the minimum is operated upon.)

Typically, *heaps* are used to implement priority queues in computer systems. In essence, a heap is a binary tree featuring the *heap-shape property*: all the leaves occur on at most two adjacent levels in the structure, with the leaves on the last level being confined to the leftmost position; and a *max-ordering*: every element is no less than the largest of its children. It is well known that in the heap implementation of priority queues *Find-max* takes constant time, while both *Delete-max* and *Insert* take $O(\log n)$ time. Furthermore, an n-element heap can be constructed in $O(n)$ time, which is trivially seen to be optimal (see Ref. 2 for details).

In fact, the idea of a priority queue can be naturally extended 'to a *double-ended priority queue* where, in addition to *Find-max*, *Delete-max*, the operations of *Find-min* and *Delete-min* are also of interest. Motivated by this concept, Atkinson *et al.*[1] have recently proposed an interesting variation on the idea of a heap: they define the *min-max heap* as a binary tree with the *heap-shape property*, and also *min-max ordered*, that is, elements on even levels are less than or equal to their descendant, and elements on odd levels are greater than or equal to their descendants.

Max–min heaps are defined completely analogously: such a structure begins with the maximum element at the root and then the heap condition alternates between minima and maxima.

A nice feature of min–max heaps is that they can be implemented in situ, with no need for additional pointers. As it turns out,[1] when the double-ended priority queue is implemented as a min–max heap, *Find-min* and *Find-max* can be performed in constant time, while *Delete-min*, *Delete-max*, and *Insert* take $O(\log n)$ time. In addition, Atkinson *et al.*[1] have presented a linear time, and thus optimal, algorithm to construct a min–max heap.

An interesting problem arising in fault-tolerant-distributed simulation[8] is the following: assume that several (computationally active) sites in a distributed system are simulating a process. It is sometimes desirable to implement these event lists as double-ended priority queues. Basic fault-tolerant requirements require that if one of these sites, say S_i, suddenly becomes computationally inactive, another one must continue the simulation performed by S_i. For this purpose we need to elect a site S_j $(i \neq j)$, which will then import the event list of S_i and will merge it with its own event list. Surprisingly, it has recently been proved[5] that merging two min–max heaps of sizes n and k, respectively, cannot be done in less than $\Omega(n+k)$ time. This negative result motivates us to investigate a different data structure to implement efficiently a double-ended priority queue. This data structure, which was first proposed in a different form by Williams[10] is herewith referred to as the min–max–pair heap (see Fig. 1). In essence, a min–max–pair heap is a binary tree H featuring the heap-shape property, such that every node in H has two fields, called the *min field* and the *max field*, and such that H has a *min–max ordering*: for every I $(1 \leqslant i \leqslant n)$, the value stored in the min field of $H[i]$ is the smallest of all values in the subtree of H rooted at $H[i]$; similarly, the value stored in the max field of $H[i]$ is the largest key stored in the subtree of H rooted at $H[i]$. We show that min–max–pair heaps can be implemented *in situ*, with no need for additional pointers. As it turns out, when the double-ended priority queue is implemented as a MinMaxPairHeap, *Find_Min* and *Find_Max* can be performed in constant time, while *Delete_Min*, *Delete_Max*, and *Insert* take $O(\log n)$ time.

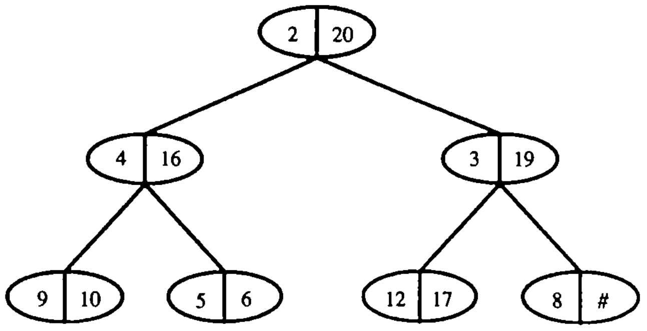

Figure 1. A min–max–pair heap.

However, what really distinguishes min–max–pair heaps from min–max heaps is the fact that min–max–pair heaps can be merged efficiently in sublinear time. More precisely, we show that two min–max–pair heaps with n and k nodes can be merged in time $O(\log n/k * \log k)$.

Recently, Carlsson[3] proposed a new data structure called the *deap*, which provides an efficient implementation of a double-ended priority queue. Formally, a deap is a data structure featuring the heap–shape property, with the left sub-tree of the non-existing root organised as a *min-heap*, the right sub-tree of the non-existing root a *max-heap*, and with each leaf in the min-heap smaller than a corresponding leaf in the max-heap. Specifically, a leaf at location t in the min-heap is smaller than the element at location $t+2^{[\log t]-1}$, in case $t+2^{[\log t]-1} \leqslant n+1$ or the element at location $[t+2^{[\log t]-1}/2]$, otherwise. It turns out that deaps can be implemented in situ and can be constructed in linear time.[3] To the best of our knowledge, however, it is an open question whether the deaps can be merged in sublinear time.

2. OPERATIONS ON MIN–MAX–PAIR HEAPS

Consider an array $H[1..n]$ as input. For $1 \leqslant i \leqslant n$, each element $H[i]$ of H has two fields $H[i]$.min and $H[i]$.max. (Therefore, the array H can be viewed as containing $2n-1$ or $2n$ keys altogether; in the case where H contains $2n-1$ keys, the max field of $H[n]$ contains a special symbol, namely #.)

The construction algorithm for min–max–pair heap resembles the construction of the standard heap structure.[1] Let $H[i]$ be an arbitrary node of the array to be made into a min–max–pair heap. Assume, further, that for all j $(i \leqslant j)$, the subtrees rooted at the children of $H[j]$, namely $H[2j]$ and $H[2j+1]$, provided they exist, have been made into min–max–pair heaps. First, we ensure that the key in $H[i]$.min is no larger than the key stored at $H[i]$.max. Next, we restore the min–max–pair heap property along the min fields of the nodes in the subtree rooted at $H[i]$, by trickling down larger keys. Similarly, we restore the min–max–pair heap property along the max fields by trickling down smaller keys. The purpose of this is to ensure that the $H[i]$.min and $H[i]$.max contain the smallest and largest keys in the subtree rooted at $H[i]$, respectively. The details are given below.

```
Procedure Create (H[1..n]);
  begin
    for i ← n downto 1 do
      Siftdown(H[i]);
  end {Create}
Procedure Siftdown(H[i]);
{we assume that the subtrees rooted at H[2i] and H[2i+1]
are min–max–pair heaps}
  begin
    Trickledown-min-field(H[i]);
    Trickledown-max-field(H[i]);
  end; {Siftdown}
Procedure Trickledown-min-field(H[i]);
  begin
    p ← H[i];
    if p.max < p.min then Swap(p.min, p.max);
    if p is a leaf then return;
    p1 ← child of p with smallest min field;
    if p1.min < p.min then begin
      Swap(p1.min,p.min); Trickledown-min-field(p1)
  end; {Trickledown-min-field}
```

Procedure *Trickledown-max-field* is similar. The following result establishes the correctness and the time complexity of our procedure.

Theorem 1. Procedure *Create* correctly induces a min–max–pair heap structure over $2n$-1 or $2n$ keys in $O(n)$ time.

Proof. To settle the correctness we proceed as follows: assume that for all values of i $(2 \leqslant i \leqslant n)$, when *Trickledown-min-field*($H[i]$) (resp. *Trickledown-max-field*($H[i]$)) terminates, $H[i]$.min (resp. $H[i]$.max) contains the smallest (resp. largest) key in the subtree rooted at $H[i]$, while the subtrees rooted at $H[2i]$ and $H[2i+1]$ (provided they exist) are min–max–pair heaps. It is easy to confirm that when *Siftdown*($H[1]$) terminates, the whole structure is made into a min–max–pair heap.

To address the complexity, consider what happens in procedure *Trickledown-min-field* when node $H[i]$ is being processed. To ensure that $H[i]$.min $\leqslant$ $H[i]$.max and to determine the child of $H[i]$ with the smallest min field three comparisons are required. Consequently, the total number of comparisons to perform *Siftdown* is

$$\sum_{i=1}^{n} 3[\lfloor\log n\rfloor - \lfloor\log i\rfloor],$$

which is easily seen to be $O(n)$. □

Next, we show that performing the standard operations *Insert*(x) and *Delete-min* as well as *Delete-max* can be done in $O(\log n)$ time. Basically, the idea of inserting a new element x into a min–max–pair heap is the same as the insertion of a new element in a standard heap. We first place the new key at the bottom of the structure and then perform the well-known bubble-up operation. Just as in the case of heaps, the time complexity of the *Insert*(x) operation for the min–max–pair heap is dominated by the cost of the bubble-up, which is easily seen to be $O(\log n)$ as shown in the following procedures.

```
Procedure Bubbleup(H[i]);
  begin
    p ← H[i];
    b ← false;
    if p.min > p.max then
        swap(p.min,p.max);
    if p is the root then return;
    p1 ← the parent of p;
    if p1.max < p.max then begin
      swap(p1.max,p.max);
      b ← true;
      end;
    if p1.min > p.min then begin
      swap(p1.min,p.min);
      b ← true;
      end;
    if b then
      Bubble(p1);
  end; {Bubbleup}
```

```
Procedure Insert(x,H[1..n]);
  begin
    if H[n].max = '#' then
      H[n].max ← x;
    else begin
      n ← n+1;
      H[n].min ← x;
      H[n].max ← '#';
    end;
    Bubbleup(H[n]);
  end.
```

Similarly, the idea of *Delete-min* and *Delete-max* resembles the well-known delete operation on heaps. The details are spelled out in the following procedures. It is an easy matter to confirm that both these operations can be executed in O(log n) time, while *Find-min* and *Find-max* take O(1) time.

```
Procedure Delete-min(H[1..n]);
  begin
    if H[n].max = '#' then begin
      H[1].min ← H[n].min; n ← n−1;
      end
    else begin
      H[1].min ← H[n].max;
      H[n].max ← '#'
    end;
    Trickledown-min-field(H[1]);
  end;
```

3. MERGING MIN–MAX–PAIR HEAPS

Recently, Sack and Strothotte[9] have proposed an efficient algorithm to merge two heaps in sublinear time. Specifically, merging two heaps of size n and k can be done in O(log $(n/k) * \log k$) time. The general case of the heap-merging algorithm in Ref. 9 reduces, in stages, to that of merging perfect heaps. (A heap H is *perfect* if the leaves occur at the last level only.) The idea in Ref. 9 is very elegant: first, to merge two perfect heaps $H1$ and $H2$ of equal size, make the rightmost leaf of $H2$ into the new root, whose children become the old roots of $H1$ and $H2$. After this, the new root is sifted down to restore the heap property.

Next, let $H1$ and $H2$ be two perfect heaps of sizes n and k, respectively, with $k < n$. Start at the root of $H1$ and compare it to the root of $H2$. If the root of $H2$ is smaller than the root of $H1$, exchange the two roots and perform a sift-down on $H2$. This operation is repeated along the path (Walk-down) in $H1$ from the root down to the leftmost leaf of $H1$ for $\log n - \log k$ steps.

We propose to show that the heap-merging algorithm in Ref. 9 can be easily adapted to merge two min–max–pair heaps in sublinear time. We shall therefore focus on merging perfect min–max–pair heaps, that is, min–max–pair heaps whose leaves occur at the last level only. We refer the interested reader to Ref. 8, where the tedious details are documented.

Just as in Ref. 9, to reduce the amount of data movement during the execution of our merging algorithm, we shall assume a pointer-based implementation. In this context, a min–max–pair heap node v contains the following fields:

- *v.min* and *v.max* fields;
- *v.lchild* contains a pointer to the left child of v in the min–max–pair heap;
- *v.rchild* contains a pointer to the right child of v in the min–max–pair heap.

It is convenient to assume that *depth*(H) returns the depth of the min–max–pair heap H in constant time. The details of our merging algorithms are as follows.

```
Procedure Merge-perfect-equal(H1, H2);
{H1 and H2 are two min–max–pair heaps of same size}
  begin
    p ← the last node in H2;
    remove p from H2;
    p.lchild ← H1;
    p.rchild ← H2;
    Siftdown(p);
    H1 ← p;
  end;
Procedure Walk-down(Hn,Hk,from,to); {Hn is a
min–max–pair heap with n nodes; Hk is a min–max–pair
heap with k nodes; 'from' is the starting location of
current operation on the path from the root to Hn to the
leftmost leaf; 'to' is the ending position of the operation}
begin
  if Hk.min < from.min then swap(Hk.min,from.min);
  if Hk.max > from.max then
  swap(Hk.max,from.max);
  Siftdown(Hk);
  if from = to then return
  else begin
    next ← next node on the walk-down after from;
    Walk-down(Hn,Hk,next,to)
    end
  end; {Walk-down}
Procedure Merge-perfect(Hn,Hk);
begin
  p ← node on the path from the root to the leftmost leaf
  in Hn, such that the subtree rooted at p has k nodes;
  r ← root of Hn;
  Walk-down(Hn,Hk,r,p);
  p1 ← parent of p;
  merge-perfect-equal(p,Hk);
  if p1 ≠ nil then
    p1.lchild ← p
  else Hn ← p
end; {Merge-perfect}
```

(For a detailed example refer to Figure 3.) It is easy to see that the complexity of our algorithm is exactly the same as that of the heap-merging algorithm in Ref. 9. Specifically, we can merge two min–max–pair heaps with n and k nodes, respectively, in O(log $(n/k) * \log k$) time.

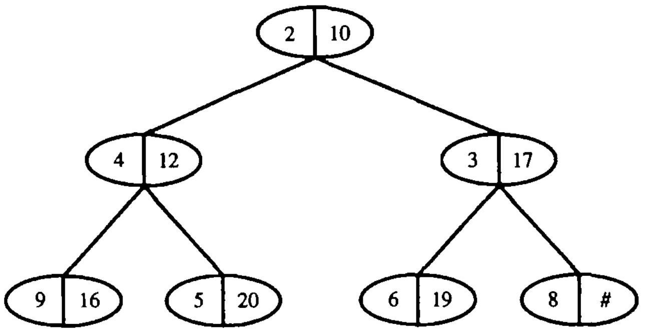

Figure 2. A min–min–pair heap.

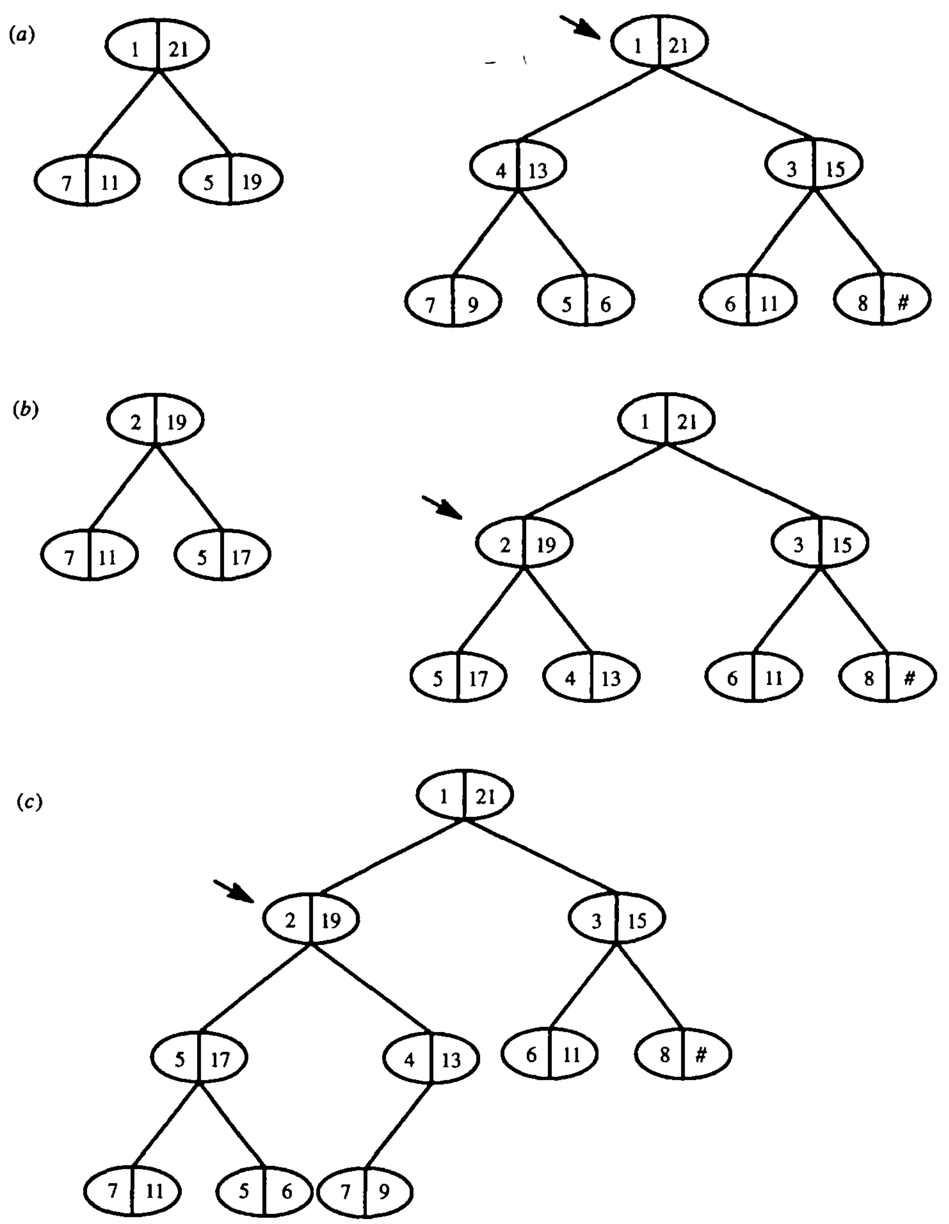

Figure 3. Merging two min–max–pair heaps. (*a*) Two min–max–pair heaps to be merged; (*b*) after one iteration (one level of recursion) of procedure 'Walk-down'; (*c*) the merge min–max–pair heap.

4. CONCLUSIONS AND FURTHER WORK

We have shown that the techniques in Ref. 9 can be readily adapted to handle two merging min–max–pair heaps. It is easy to see that the idea that led to the min–max–pair heap can be further expanded. As an example, we define a min–min–pair heap as a heap-shaped binary tree with each node containing two fields called *min*1 and *min*2, respectively. The value of $p.min1$ is the smallest of all the value stored in the subtree rooted at p; $p.min2$ contains the smallest of all the values stored in the *min*2 fields of all the nodes in the subtree rooted at p. Finally, for every node q in the subtree rooted at p, $p.min2 \geqslant 2.min1$ (see Fig. 2).

One interesting feature of a min–min–pair heap is that the *min*1 field of the root contains the minimum value in the whole structure, while *min*2 of the root contains the median of the whole structure. As it turns out,[8] a min–min–pair heap containing $2n$-1 or $2n$ keys can be constructed in O(n) time. Clearly, the operations *Find-min* and *Find-median* can be performed in O(1) time. Similarly, *Insert*(x), *Delete-min* and *Delete-median* can be done in O(log n) time.[8] Similarly, one can define a max-max-pair heap and a max-min-pair heap.[8] Unfortunately none of these variations of the min–max–pair structure is mergeable in sublinear time.

Finally, an interesting open problem is whether or not deaps are mergeable in sublinear time. In particular, it would be interesting to see if the techniques in Ref. 9 can be extended to deaps.

Acknowledgement

The authors would like to thank an anonymous referee for a number of constructive comments. A preliminary version of this paper has appeared in the *Proceedings* of the 29th Annual ACM SE Regional Conference, Auburn, Alabama, 1991. The first author was supported, in part, by NSF under grant CCR-8909996.

REFERENCES

1. M. D. Atkinson, J. R. Sack, N. Santoro and T. Strothotte, Min-max heaps and generalized priority queues. *Comm. ACM* **29**, 996–1000 (1986).
2. S. Baase, *Computer Algorithms – An Introduction to Design and Analysis*. Addison–Wesley, Reading, MA (1988).
3. S. Carlsson, The deap – a double ended heap to implement double ended priority queues. *Information Processing Letters* **26**, 33–36 (1987).
4. G. H. Gonnet, Heaps applied to event-driven mechanisms. *Comm. ACM* **19**, 417–418 (1976).
5. A. Hasham and J. R. Sack, Bounds for min-max heaps. *BIT* **27**, 315–323 (1987).
6. K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. McGraw-Hill, New York (1984).
7. O. Nevalainen and J. Teuhola, Priority queue administration by sublist index. *The Computer Journal* **22**, 220–224 (1977).
8. S. Olariu and Z. Wen, *The Min–max-pair Heap and its Variations*. Department of Computer Science, Old Dominion University, Technical Report TR-89-33 (revised October 1990).
9. J.-R. Sack and T. Strothotte, An algorithm for merging heaps. *Acta Informatica* **22**, 171–186 (1985).
10. J. W. J. Williams, Algorithm 232. *Comm. ACM* **7**, 347–348 (1964).

Correspondence

Sir,
In many applications of computer engineering,[1] we are often in need of converting decimal data into its equivalent data of other base(r) system such as binary, octal, hex etc. One of the usual techniques of such conversion is known as 'dibble-double' technique.[1–3] But the 'dibble-double' technique has two major limitations as listed below.

(1) The technique uses two different algorithms[1–4] for conversion of real data. One algorithm is used for conversion of the integer part of the data and another algorithm is used for conversion of the fractional part of the data.

(2) The order of the bits of integer part of the data is the reverse to that of the fractional part of the data on conversion. Such nature of order of bits of two parts of data becomes a common source of error particularly for paper and pencil work.[3]

The other usual technique of data conversion is known as 'table-look-up'.[3] But the 'table-look-up' technique is efficient only for conversion of any decimal data to its equivalent binary data (D–B). The technique is not efficient for the cases of conversion of D–O (decimal to octal) and of D–H (decimal to hex) etc. This is because the tables used in the table-look-up are very difficult to design in these cases. The size of the table increases in term of columns with increase of base in which data is to be converted. For example, the required number of columns of the table in case of D–B conversion of data up to the maximum decimal integer of eight is two; while the same for the case of D–O conversion goes up to eight. While using this technique in conversion of data, the required searching time[3] of the table for conversion increases with the increase of the size of this table. Hence the technique is a slow one, particularly for the cases of conversion of D–O and of D–H etc. Moreover, the table increases its size towards row as the data under conversion increases.

The above-stated limitations of existing techniques of conversion of data may be removed, if the new algorithm given below is used for the same purpose.

The new algorithm is based on the fact that any decimal integer, I in any other base system, r can be expressed as:

$$a_n \gamma^n + a_{n-1}\gamma^{n-1} + \ldots + a_0 \gamma^0,$$

where a_i for $i = n, n-1, \ldots, 0$ are the bits representing I in base, r.

But

$$(a_n \gamma^n + a_{n-1}\gamma^{n-1} + \ldots + a_0 \gamma^0) \geqslant a_n \gamma^n \ldots \quad (1)$$

Thus the subtraction of r^n from I for a_n number of times must meet the inequality (1). At the next such subtraction the inequality (1) must not be satisfied as:

$$(a_n \gamma^n + a_{n-1}\gamma^{n-1} + \ldots + a_0 \gamma^0) < (a_n + 1)\gamma^n \ldots \quad (2)$$

Thus counting the number of times for which r^n is subtracted from I in satisfying the inequality (1) will provide the value of a_n. Similarly any other bit, a_i can be evaluated.

For the fractional part (F) of any decimal data the related inequalities are:

$$(a_{-1}\gamma^{-1} + a_{-2}\gamma^{-2} + \ldots + a_{-m}\gamma^{-m}) \geqslant a_{-1}\gamma^{-1}$$

and

$$(a_{-1}\gamma^{-1} + a_{-2}\gamma^{-2} + \ldots + a_{-m}\gamma^{-m}) < (a_{-1} + 1)\gamma^{-1}.$$

In case of the conversion of fractional part of data, in some cases the number of required bits on conversion may be very high. In such situations a limit in the number of bits may be included. This introduces the so called conversion error.[1]

The above-stated idea forms a basis of new algorithm for conversion of data.

A numerical example to convert decimal data 5.8 in binary is given below to show the difference between the 'dibble-double' algorithm and the new algorithm.

Under dibble-double algorithm

Integer part:

```
2|5|2                2|2|1    2|1|0 (indicates end
  4                    2        0    of operation)
  -                    -        -
  1 (LSB-Least         0        1   (MSB-Most
     significant                     significant bit)
     bit)
```

Fractional part up to 4th bit:

$$0.8 \times 2 = 1.6 \rightarrow 1 \text{ (MSB)},$$
$$0.6 \times 2 = 1.2 \rightarrow 1,$$
$$0.2 \times 2 = 0.4 \rightarrow 0,$$
$$0.4 \times 2 = 0.8 \rightarrow 0 \text{ (LSB)}.$$

Observation

(1) Two separate algorithm for conversion of integer and that of fractional part are used.

(2) Order in which MSB to LSB is evaluated in reverse to each other in two cases.

Under new algorithm

Integer part:

$$5 - 2^2 = 1 > 0 \rightarrow 1 \text{ (MSB)},$$
$$1 - 2^1 = -1 < 0 \rightarrow 0,$$
$$1 - 2^0 = 0 = 0 \rightarrow 1 \text{ (LSB)}.$$

(Indicates end of operation).

Fractional part

$$0.8 - 2^{-1} = 0.3 > 0 \rightarrow 1 \text{ (MSB)},$$
$$0.3 - 2^{-2} = 0.05 > 0 \rightarrow 1,$$
$$0.05 - 2^{-3} = -0.075 < 0 \rightarrow 0,$$
$$0.05 - 2^{-4} = -0.0125 < 0 \rightarrow 0 \text{ (LSB)}.$$

Observations

(1) Single algorithm is used for both the cases of conversion of integer and fractional part of data.

(2) Order in which MSB to LSB is obtained is same in both the cases of the integer and the fractional part of data.

C. T. BHUNIA

Department of Computer Science, North Bengal University (and NERIST), Dist. Darjeeling, Pin 734 430, West Bengal, India

References

1. T. L. Booth, *Introduction to Computer Engineering*, Chapter 2, John Wiley and Sons, New York (1978).
2. J. J. F. Cavanagh, *Digital Computer Arithmetic*, Chapter 1, McGraw-Hill, Singapore (1985).
3. W. H. Gothmann, *Digital Electronics*, Chapter 2: Prentice-Hall, N.J., U.S.A. (1987).
4. Gibson and Liu, *Microcomputers for Engineers & Scientists*, Chapter 2, Prentice-Hall, N.J. (1980).