1989

Efficient Heap Implementation with a Fixed-Size Linear Systolic Array

Jyh-Jong Tsay

Report Number:
89-911

911

Efficient Heap Implementation With A Fixed-Size Linear Systolic Array

Jyh-Jong Tsay

Department of Computer Science

Purdue University

West Lafayette, IN 47907.

Abstract

The heap is a data structure used in many applications and provides a funfamental technique to solve many problems efficiently. In this paper, we show that a sequence of n INSERT and EXTRACT_MIN heap operations can be performed in time $O(n \log m / \log p)$ with space $O(m)$ on a random access machine to which a linear systolic array of p processors is attached, provided that, at any time instance, there are at most m ($m \leq n$) data elements in the heap. The algorithm can be easily to modified to handle DELETE operation with time $O(n \log n / \log p)$ and space $O(n)$.

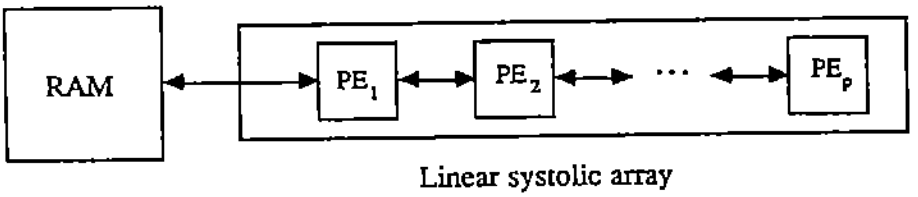

Figure 1: The Machine Model

1 Introduction

The machine model used in this paper consists of a random access machine to which a linear systolic array of p processing elements is attached (see Figure 1). Each of the p processing elements has only $O(1)$ local memory. The same machine model has been considered for sorting [3, 8] and geometric problems [4] and optimal algorithms have been given. In this paper, we consider the problem of performing a sequence of n INSERT, EXTRACT_MIN operations on this machine model. We present an algorithm which takes $O(\log m/\log p)$ amortized time for each operation and $O(m)$ space, provided that, at any time instance, there are at most m ($m \leq n$) data elements existing. The algorithm can be easily modified to handle DELETE operation with time $O(\log n/\log p)$ and space $O(n)$. The time bound is optimal since sorting can be reduced to a sequence of n INSERT operations followed by n EXTRACT_MIN operations and the time performance of $O(n \log n/\log p)$ is the best possible for sorting [1].

Section 2 outlines our algorithm, Section 3 and 4 gives details and analysis, and Section 5 concludes.

We assume throughout that $n = p^k$ and $m = p^l$ for some positive integers k, l. Our scheme can be easily modified to handle the case of arbitrary n and m.

2 Outline of the algorithm

It is known that a linear systolic array of p processing elements can be used to store $2p$ data elements such that each INSERT and EXTRCAT_MIN operation can be performed in $O(1)$ time [7]. Our algorithm is mainly based on a way, which will be described later, to store

most of the m data elements (except no more than p ones) in the random access machine such that their minimum can be obtained and deleted efficiently. Only a small portion (no more than $2p$) of the m data elements are stored in the linear systolic array. The condition maintained throughout our algorithm is that the smallest one is always stored in the linear systolic array. We then use the algorithm in [7] for INSERT and EXTRACT_MIN operations to insert a data element into or extract the minimum from the linear systolic array in $O(1)$ time. We next explain how the data elements are stored in the random access machine.

We store the data elements in the random access machine in $p \log m / \log p$ sorted lists as follows. View the memory of the random access machine as consisting of $p \log m / \log p$ *blocks* $b_1, b_2, \ldots, b_{p \log m / \log p}$, such that the size of block b_i is $p^{\lceil i/p \rceil}$. Each block consists of consecutive memory locations. The data elements are stored in blocks and each block contains an increasing sorted sequence.

The $p \log m / \log p$ blocks are grouped into p groups $G_1, G_2, \ldots, G_p$, according to their indices, such that group G_i consists of $k(= \log m / \log p)$ blocks $b_{(i-1)k+1}, b_{(i-1)k+2}, \ldots, b_{ik}$. For any group G_i, define the *group minimum* of G_i to be the smallest one among those which are stored in blocks of G_i. If there is no data element stored in G_i, the group minimum of G_i is defined to be ∞. The data elements stored in the linear systolic array are the p group minimums and no more than p most recently inserted ones (which will be clear from the details of INSERT operation). We next outline our algothms and review some previous results which will used in our algorithms later.

Define *level* C_i to be the set of blocks of size p^i, i.e. $C_i = \{b_{(i-1)p+1}, b_{(i-1)p+2}, \ldots, b_{ip}\}$. When a data element is inserted, it will be first inserted into the linear systolic array until the linear array holds $2p$ data elements. (At that time, p of them are group minimums and p of them are from INSERT operations.) Those p ones which are not group minimums are then moved to a block of C_1. When all the blocks in C_1 are occupied, we then move all the data elements stored in blocks of C_1 to a block of C_2. This needs to merge the p sorted lists of size no more than p, which are stored in C_1, into a sorted list of size no more than p^2. We call such a merge *level-merge*. The level-merge will be performed on any level C_i if all the blocks of C_i are occupied. To report and delete the smallest element, we first extract the smallest element from the linear systolic array and report it. If the extracted element is a group minimum of some G_i, we then delete that element from G_i, find and insert the

new group minimum of G_i into the linear systolic array.

It is easy to verify that the space of the algorithm outlined as above is $O(n)$. We adopt the following strategy to reduce the space to $O(m)$: for each level C_i, at most one of its occupied blocks is allowed to contain less than $p^i/2$ data elements. If there are two blocks in C_i containing less than $p^i/2$ elements, we then merge them into one sorted list. We call such a merge *local-merge*. It is well known that a local-merge takes linear time.

Lemma 2.1 *The space complexity of above outlined algorithm is $O(m)$ if in each level C_i, at most one of its occupied block contains less than $p^i/2$ data elements.*

Proof: To count the space, we count the number of data elements stored in the random access machine when a block b in C_i is first used as follows. Suppose there are $w < p$ other blocks in C_i used before b. Note that b will be used to store at least $(p-2)p^{i-1}/2$ data elements as b is used to store the result of a level-merge on C_{i-1}. Since b is used first time, all the blocks in C_j for $j > i$ have not yet been used. Thus, the space used up to now is $O((w+2)p^i)$ (including b). Since there may be at most one used blocks other than b in C_i containing less than $p^i/2$ data elements, the total number of data elements stored in the random access machine is at least $(w-1)p^i/2 + (p-2)p^{i-1}/2$, which is $O(wp^i)$. Thus, the space is $O(m)$. □

We next review a lemma which is given in [3].

Lemma 2.2 *A level-merge on C_i can be performed in time linear to the number of data elements stored in C_i on our machine model.*

Proof: In [3], Atallah, Frederickson and Kosaraju show that, given p sorted lists, we can merge them into one sorted lists in time linear to the number of data elements involved on our machine model by doing the following steps: (i) select the pth element from the p sorted lists (each list is in one block) in $O(p)$ time, using the selection algorithm of Frederickson and Johnson [6], (ii) identify and delete the group of the first p elements from the p sorted lists in $O(p)$ time by examining each sorted list, (iii) sort the identified group of the first p elements in $O(p)$ time by use of the linear systolic array, (iv) if the p sorted lists are not all empty, repeat (i), (ii) and (iii). □

We next give details of INSERT and EXTRACT_MIN operations.

3 The INSERT operation

Our algorithm for INSERT operation consists of the following steps.

1. Insert the new data element into the linear systolic array.

2. If there are less than $2p$ data elements stored in the linear systolic array, then stop. Otherwise go to Step 3.

3. Read the data elements from the linear systolic array in sorted order. Identify those p ones which are not group minimums and store them in a block of C_1. All can be done in $O(p)$ time. (The implementation of element identification and block allocation is simple and omitted.)

4. If now all the blocks in C_1 are occupied, then perform a level-merge on C_1 and store the result in a block of C_2. This level-merge process will be continued on C_2, C_3, ..., up to C_i, where i is the smallest integer such that there are still some blocks in C_{i+1} not occupied after the result of level-merge on C_i is stored. Now, if there are two blocks in C_{i+1} containing less than $p^{i+1}/2$ data elements, then do a local-merge to merge them into one sorted list.

5. Regroup blocks into $G_1, G_2, \ldots, G_p$ according to m. Find the group minimum of each G_i, and insert it into the linear systolic array. The linear systolic array now contains p data elements in increasing sorted order. The group minimun of each G_i is obtained by examining the first element of each occupied block in G_i.

The correctness of above algorithm is established as follows. Step 1 inserts the new data element into the linear systolic array and ensures that the data elements which are not in the memory of the random access machine are in the linear systolic array. Step 3 and 4 ensures that the data elements stored in each block are in increasing sorted order and that, for each C_i, at most one of its occupied blocks contains less than $p^i/2$ data elements. Step 5 ensures that the p group minimums are stored in the linear systolic array. Thus, the minimum of the data elements stored in the linear systolic array is the minimum of the current data. This gives the correctness of above algorithm.

To establish that the total time for all the INSERT operations is $O(n \log m / \log p)$, we count the time as follows. It is obvious that the time for Step 1 and 2 is $O(1)$ [7]. We

analyze the time for Step 3, 4 and 5 with an amortized scheme. We charge the cost of Step 3 and 5 to the p INSERT operations which insert those p elements which are stored in the linear systolic array but not group minimums. Since the space is $O(m)$ (hence the number of blocks is $O(p \log m / \log p)$), Step 5 takes $O(p \log m / \log p)$ time to regroup the blocks and find the group minimums. It is easy to perform Step 3 in $O(p)$ time. Thus, the time charged to each INSERT operations due to Step 3 and 5 is $O(\log m / \log p)$.

To analyze the time for Step 4, we count the time to do level-merges and the time to do local-merges separately. We charge the time of each level-merge to the data elements involved. Since each level-merge can be performed in time linear to the number of data elements involved (Lemma 2.2), the time charged to each data element due to level-merges in Step 4 is $O(\log m / \log p)$ as each element can be involved in one level-merge while in each level. We now count the time for local-merges. Consider a local-merge of Step 4 which is to merge two sorted lists stored in blocks b_{j_1} and b_{j_2} of C_{i+1}. We know that one of b_{j_1} and b_{j_2} is used to store the result of the level-merge on C_i. Let it be b_{j_1}. We charge the cost of the local-merge to the data elements stored in b_{j_1}. Since b_{j_1} contains at least $(p-2)p^i/2$ data elements and the local-merge takes $O(p^{i+1})$ time, the time charged to each element due to local-merges in Step 4 is $O(\log m / \log p)$ ($O(1)$ while in each level). Since there are only $O(n)$ INSERT operations and hence $O(n)$ data elements, the total time for all the INSERT operations is therefore $O(n \log m / \log p)$. We then have the following lemma.

Lemma 3.1 *Above algorithm performs all the INSERT operations in $O(n \log m / \log p)$ time.*

4 The EXTRACT_MIN operation

The details of EXTRACT_MIN operations are as follows.

1. Extract the minimum from the linear systolic array and report it.
2. If the minimum is not a group minimum then stop. Otherwise, go to Step 3.
3. Let the minimum be in $b_j \in G_i$. Delete it from b_j. If there are two blocks in G_i containing less than $p^i/2$ data elements, then do a local-merge to merge them into one sorted list. Find the new group minimum of G_i and insert it into the linear systolic array.

The correctness of above algorithm is straightforward. We only analyze its time complexity as follows. Step 1 and Step 2 take $O(1)$ time. For Step 3, deletion of the minimum from a block can be done in $O(1)$ time by changing the index of the first element in that block, and finding the group minimum of G_i can be done in $O(\log m/\log p)$ time since G_i has only such many blocks. We next count the time to do local-merges of Step 3.

Consider a forest which records the history of local-merges in Step 3. Each occupied block b_j is associated with a history tree T_j, and each node v of T_j is associated with a sorted list $I(v)$ which is a result of some local-merge or level-merge. Assume b_j is in C_i. Tree T_j is formed as follows. If block b_j is used to stored the result of a local-merge on b_{j_1} and b_{j_2}, then T_j is a tree with the roots of T_{j_1} and T_{j_2} as the only children of its root. For any tree T, let $root(T)$ denote the root of T. List $I(root(T_j))$ is then the result of the local-merge on b_{j_1} and b_{j_2}. After the local-merge on b_{j_1} and b_{j_2}, trees T_{j_1} and T_{j_2} are cleared to be empty. Otherwise, block b_j is used to store the result of a level-merge on C_{i-1}, and T_j is a tree with only one node and $I(root(T_j))$ is the result of that level-merge. After a level-merge on C_{i-1}, the p history trees associated blocks of C_i are cleared to be empty. We then have the following simple lemma.

Lemma 4.1 *Let b_j be a block of C_i. Then,*

1. *for any node v of T_j, $|I(v)| \geq (p-2)p^{i-1}/2$, and*
2. *if v_1 and v_2 are two children of v, and $v_{1,1}$, $v_{1,2}$ and $v_{2,1}$, $v_{2,2}$ are children of v_1 and v_2, then $|(I(v_{1,1}) \cup I(v_{1,2}) \cup I(v_{2,1}), \cup I(v_{2,2})) - I(v)| \geq (p-4)p^{i-1}$.*

Based on above lemma, we then count the cost of local-merges in Step 3 with an amortized scheme as follows. Consider a local-merge of Step 3 which is to merge blocks b_{j_1} and b_{j_2} of C_i. If one of $root(T_{j_1})$ or $root(T_{j_2})$, say $root(T_{j_1})$, is a leaf, we then charge the cost of the local-merge to the data elements which are in $I(root(T_{j_1}))$. Since $|I(root(T_{j_1}))| \geq (p-2)p^{i-1}$, the cost so charged to each element is $O(\log m/\log p)$ ($O(1)$ while in each level). Otherwise, let $v_{j_{1,1}}$, $v_{j_{1,2}}$ be the two children of $root(T_{j_1})$ and $v_{j_{2,1}}$, $v_{j_{2,2}}$ be the two children of $root(T_{j_2})$. We then chage the cost to the data elements in $(I(v_{j_{1,1}}) \cup I(v_{j_{1,2}}) \cup I(v_{j_{2,1}}) \cup I(v_{j_{2,2}})) - I(root(T_j))$. Since $|(I(v_{1,1}) \cup I(v_{1,2}) \cup I(v_{2,1}) \cup I(v_{2,2})) - I(v)| \geq (p-4)p^{i-1}$ and each element is charged at most twice while in each level, the cost so charged to each element is $O(\log m/\log p)$ ($O(1)$ while in each level). Since there are no more than n data elements, the time to perform all the EXTRACT_MIN operations is $O(n \log m/\log p)$.

Lemma 4.2 *Above algorithm performs all the EXTRACT_MIN operations in time* $O(n \log m / \log p)$.

Therefore, we have the following theorem.

Theorem 4.1 *A sequence of* n *INSERT and EXTRACT_MIN operations can be performed in time* $O(n \log m / \log p)$ *with space* $O(m)$ *on a random access machine to which a linear systolic array of* p *processing elements is attached, provided that, at any time instance, there are no more than* m *data elements existing.*

5 Conclusion

We have shown that a sequence of n INSERT and EXTRACT_MIN operations can be performed in time $O(n \log m / \log p)$ with space $O(m)$ on a random access machine to which a linear systolic array of p processing elements is attached. The algorithm can be easily modified to handle DELETE operations with time bound $O(n \log n / \log p)$ and space $O(n)$. The idea is to view each DELETE as an INSERT but mark the data element as "deleted". Deletions are performed only when a deleted element meets an inserted element with the same value. This gives a general scheme to obtain $O(n \log n / \log p)$ solutions for many problems which can be reduced to a sequence of INSERT, DELETE and EXTRACT_MIN operations. One of such problems is to schedule n unit-length jobs on q identical processors to meet deadlines [5].

Acknowledgment

The author would like to thank M.J. Atallah for his support and helpful comments of this work.

References

[1] A. Aggarwal and J.S. Vitter, The Input/Output Complexity of Sorting and Related Problems, *Comm. ACM 31*, pp. 1116-1127, Sept. 1988.

[2] A. V. Aho and J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[3] M.J. Atallah and G.N. Frederickson and S.R. Kosaraju, Sorting With Efficient Use of Special-Purpose Sorters, *Info. Process. Lett.*, 27, pp. 13-15, 1988.

[4] M.J. Atallah and J.-J. Tsay, On the Parallel-Decomposability of Geometric Problems, *ACM Proceedings of the fifth annual symposium on Computational Geometry*, pp. 104-113, June 1989.

[5] , J. Blazewicz, Simple algorithm for multiprocessor scheduling to meet deadlines, *Info. Process. Lett.*, 21, pp. 162-164, 1977.

[6] G.N. Frederickson and D.B. Johnson, The Complexity of Selection and Ranking in $X+Y$ and Matrices with Sorted Columns, *J. of Computer and System Science 24(1982) 197-208.*

[7] D.T. Lee and H. Chang and C.K. Wong, An on-chip compare steer bubble sorter, *IEEE Trans. Computers*, Vol. 6, c-30, pp. 396-405, 1981.

[8] H. Mueller, Sorting Numbers Using Limited Systolic Coprocessors, *Info. Process. Lett.*, 24, pp. 351-354, 1987.