International Journal of Software Engineering and Its Applications
Vol. 9, No. 1 (2015), pp. 201-212
http://dx.doi.org/10.14257/ijseia.2015.9.1.18

Performance Evaluation of Cache-aligned 4-ary AM-heap

Haejae Jung

*Department of Information and Communication Engineering
Andong National University, Andong, South Korea
hjjung@anu.ac.kr*

Abstract

AM-heap is a heap data structure with O(1) amortized insertion time and O(logn) deletion time complexities. This paper presents cache-aligned 4-ary AM-heap in which every node has four children except that the root has three children. In this heap, the leftmost node of each sibling has an index of a multiple of four. Also, every node index matches its corresponding array index and every sibling nodes smoothly fits into a single cache block. Our experimental results show that cache-aligned 4-ary AM-heap is faster than AM-heap as well as post-order heap.

Keywords: *heap, priority queue, data structures, computer algorithm*

1. Introduction

Heaps are fundamental data structures to implement priority queues. They can be used in a wide range of applications such as scheduling, event-driven simulation, sorting, shortest paths computation, and extracting elements with the highest priority. In this paper, the element with the highest priority is defined as the element with the minimum key.

The basic functions of a priority queue(PQ) are insertion and deletion:

- insertion: *insert(x)* puts an arbitrary element *x* into PQ.
- deletion: *removeMin( )* deletes the element with the minimum key from PQ.

Some implicit heaps have been published: the conventional binary heap by J. Williams, implicit binomial queue by S. Carlsson, *et al.*, post-order heap by N. Harvey and K. Zatloukal and AM-heap by H. Jung.

Among them, the conventional heap is the most well-known simple data structure for implementing priority queues [3, 5]. It supports both insertion and deletion operations in O(log*n*) time where *n* is the number of elements in the heap. Even though the time complexities for the two operations are O(log*n*) each, it is very fast practically.

The other three implicit heaps have constant amortized insertion and *O*(log*n*) deletion time complexities. The implicit binomial queue is the first heap with these time complexities but rather complicated to implement [4]. The post-order heap was published by N. Harvey and K. Zatloukal [1] and it is simpler than the implicit binomial queue. The AM-heap by H. Jung is much simpler than the post-order heap due to its simple indexing scheme [2].

For the pointer-based heaps with O(1) amortized insertion time complexity, we would mention binomial heap, Fibonacci heap, pairing heap, and M-heap [10-12]. But, in this paper we are not interested in the heaps using pointers.

In the pointer-based heaps, memory space may be more wasted than data structures utilizing an array since pointers are not used to store data. Also, by using an array to implement an implicit heap, we can utilize cache memory aligned with the array for speedup [6-8]. Therefore, a data structure using an array and its indexes may save memory space as well as time for updating pointers.

This paper proposes a cache-aligned 4-ary AM-heap with constant amortized insertion time complexity to implement priority queues. This data structure uses a variant of 4-ary tree in which only the root has three children and the rest have 4 children, to utilize the regularity of index numbers of nodes and cache memory efficiently. The proposed heap is based on a 4-ary complete tree, not on a 4-ary full tree. In this paper, we define 4-ary *full* tree as a tree in which every leaf is at the same level and every internal node has four children. We also define 4-ary *complete* tree as a 4-ary tree in which there may be some missing leaf nodes on the right from a 4-ary full tree.

In the next section, we review some related works. The Section 3 describes the cache-aligned 4-ary AM-heap for implementing priority queues. In Section 4, we show the experimental results conducted by us. And then, the conclusion of this paper follows.

2. Related Works

In this section, we review some implicit heaps: the conventional heap, post-order heap, and AM-heap.

The conventional binary heap by J. Williams uses a complete binary tree and its insert( ) and removeMin( ) operations take O(log*n*) time each [3]. The nodes in a heap are indexed in top-down and left-to-right level order, possibly starting with the root indexed 1. So, the parent and children from a node can be found by simple expressions: the parent index of node *i* is calculated by $\lfloor i/2 \rfloor$, and the index of the left/right child of node *i* is calculated by $2i/2i+1$, respectively.

For the post-order heap [1], each node is indexed in the postorder that is different from the conventional heap, as can be seen in Figure 1. Figure 1 shows a post-order heap with four component heaps rooted at node 7, 10, 11, 12 and node 12 is the last root. The post-order heap is based on a full binary tree. In this data structure, insert( ) is performed in the postorder and removeMin( ) in the reverse postorder. To implement these two operations, two variables and a bit array are maintained: the array index *n* and the height *h* of the root of the last component heap, and a bit array *D[ ]* with the ancestry bit string of node *n*. The bit value *D[i]* is set to '0'/'1' if node *i* is the left/right child of its parent, respectively. The insert( ) operation puts a new element in node $n+1$ and heapify( ) operation is performed at node $n+1$ in top-down fashion. The right child index of node *n* is simply $n-1$, and the left child index can be calculated by $n-1- s(h_{n-1})$ where h_{n-1} is the height of node $n-1$ and the size $s(h)$ of a subtree with height *h* is equal to $2^{(h+1)} - 1$. The removeMin( ) operation finds node *min* with the minimum element to remove and moves the element of the last node *n* to node *min* and does heapify at node *min*. The traversal of the roots of all component heaps starts from the last root *n* with height *h*. In general, from the root *r* of a component heap with height *h*, the height h_l and the root index r_l of its left neighbor component heap *lnch* are calculated as follows: first increment *h* and move *r* up while *D[h]* is '0' and then assign *h* and $r- s(h_l)$ to h_l and r_l respectively. As explained above, these two operations utilize rather complicated equations, keeping track of an ancestry bit array as well as the index and height of the last node. These complex expressions make difficult to implement and debug this data structure.

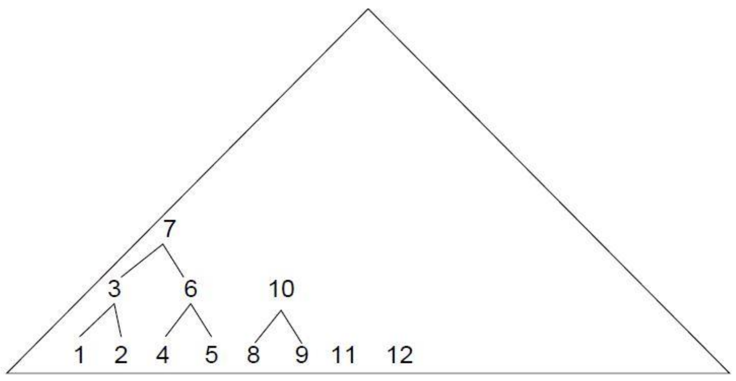


Figure 1. Post-Order Heap

AM-heap uses a complete binary tree, instead of a full binary tree. Also, the AM-heap utilizes the simple indexing scheme of the conventional heap by J. Williams. Elements are inserted in the postorder, starting from the leftmost leaf node 2^t where t is equal to $\lfloor \log_2 MaxIndex \rfloor$ and *MaxIndex* is the largest index of array allocated. The indexes of the parent and child of a node can be calculated by the expressions mentioned in the conventional heap above. Elements are removed in the reverse post-order, as with the post-order heap. In this data structure, two variables *lastRoot* and *lastLeaf* are maintained to indicate the root and rightmost leaf node of the last component heap, respectively. Starting from root *r* of a component heap, we can easily find the root index r_l of its left neighbor component heap by simply utilizing node index: after we repeatedly move *r* up while *r* is even, we can set $r_l = r\text{-}1$.

The next section presents the cache-aligned 4-ary AM-heap.

3. Cache-aligned 4-ary AM-heap

This section presents cache-aligned 4-ary AM-heap in which each node has 4 children except for the root. The 4-ary AM-heap can utilize cache memory for speedup by putting 4 sibling nodes into a cache line so that they can't span a cache line boundary [2, 6, 8].

As can be seen in Figure 2, the 4-ary AM-heap adopts the 4-ary tree structure of [8] to utilize the regularity of its indexes: that is, the leftmost child of a node has an index of a multiple of 4.

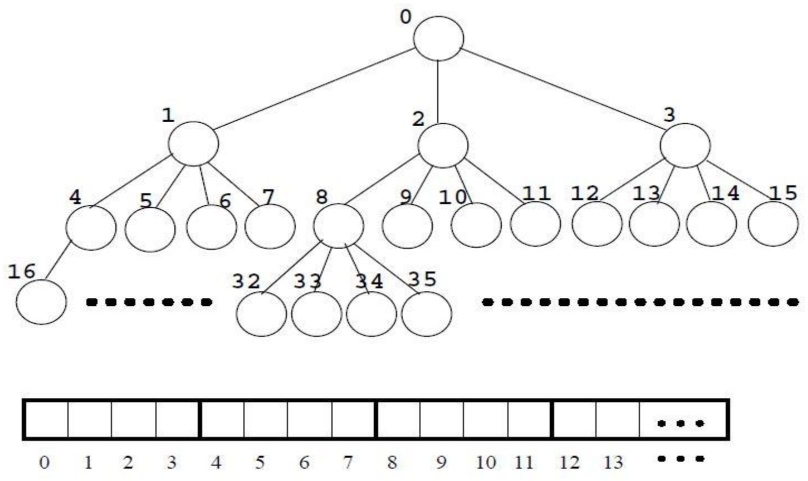


Figure 2. Cache-Aligned 4-ary AM-heap and its Array Representation

In the 4-ary AM-heap, the parent index of node *i* is calculated by a simple expression $\lfloor i/4 \rfloor$ and the 4 children indexes of node *i* are calculated by *4i, 4i+1, 4i+2, 4i+3* each. One exception is that the root has three children whose indexes are *4i+1, 4i+2, 4i+3.* So, the root and its children should be specially handled in an implementation if the root is used.

Figure 2 shows the structure of 4-ary AM-heap and its cache-aligned memory representation in which node indexes match their corresponding array indexes. It is assumed that the size of a cache-line has a multiple of the size of 4 nodes. Without loss of generality, we assume that the size of 4 nodes is equal to that of a cache-line. Then, the root and its three children can be packed into a cache-line. The 4 children of node *1* into another cache-line, and so on. In this way, we can put a 4-ary AM-heap into cache-aligned memory.

Figure 3 shows an implementation structure of the 4-ary AM-heap. The array *a[ ]* and *lastRoot* and *lastLeaf* are changed while insert( ) and removeMin( ) operations are performed. *MaxIndex* and *StartLeaf* are set in the constructor AM4Heap( ). There are two utility functions heapify( ) and resize( ). The function heapify( ) is called by insert( ) and removeMin( ) to make a subtree rooted at *r* heap and the resize( ) operation is called by insert( ) to make the size of array *a[ ]* double when the array is full.

The operation heapify( ) is the same as that of the conventional heap except that each node has 4 children, instead of 2 children. So, its time complexity is O($\log_4 m$) where *m* is the number of elements in a subtree rooted at *r*.

The operation resize( ) allocates an array *b[ ]* whose size is twice bigger than that of *a[ ]*. Then, every element of *a[ ]* is moved to array *b[ ]*, traversing each array in postorder. Whenever an element is moved into *b[i]*, heapify(*i*) operation is performed in top-down fashion. So, the resize( ) operation takes O(*n*) time where *n* is the number of elements in a 4-ary AM-heap (for proof, refer to [2] or Theorem 1 in this section).

```
Class AM4Heap {
    a[ ];
    MaxIndex, StartLeaf;
    lastRoot, lastLeaf;

    AM4Heap( nMax );  // constructor
    insert( x );
    removeMin( );

    heapify( r );
    resize( );  // double array size
}
```

Figure 3. Implementation Structure of 4-ary AM-heap

Figure 4 shows the constructor of MA4Heap that initializes variables and allocates cache-aligned memory for *nMax* elements at line 3. *MaxIndex* is set to have the last internal node have 4 children. *StartLeaf* denotes the leftmost leaf node of a 4-ary AM-heap that is the first node to insert an element. Variables *lastRoot* and *lastLeaf* indicate the root and rightmost leaf node of the rightmost component heap, respectively.

```
AM4Heap( nMax ) {
   MaxIndex = (nMax/4 + 1) * 4 -1;
   a = Element[MaxIndex+1];

   t = log_4(MaxIndex);
   StartLeaf = 4^t;
   lastRoot = -2;   // empty heap
   lastLeaf = StartLeaf - 1;
}
```

Figure 4. Constructor of 4-ary AM-heap

Figure 5 shows insert( ) operation. First, if 4-ary AM-heap is full, resize( ) operation is called to increase the size of array *a[ ]*. In a 4-ary AM-heap, at most 4 component heaps have the same height since elements are inserted in the postorder: a new node is inserted as their parent whenever the rightmost 4 component heaps are of the same height. Specifically, a new element is inserted into the parent node $\lfloor lastRoot/4 \rfloor$ if *(lastRoot+1) % 4* is equal to *0* (lines 4~10). Otherwise, a new component heap with a single element is added, as can be seen in lines 11~15. The heapify( ) operation at node *lastRoot* at line 8 performs in top-down fashion. At each level, all the children of a node are accessed to select the minimum element. Since all the children are in the same cache block, at most one cache miss occurs at each level.

```
1: insert( x )
2: {
3:    if( heap is full ) resize( );

4:    if( (lastRoot+1) == a multiple of 4 ) {
5:       // insert x into the parent of node lastRoot
6:       lastRoot /= 4;
7:       a[lastRoot] = x;
8:       heapify( lastRoot );
9:       return
10:   }

11:   // make a new component heap
12:   lastRoot = ++lastLeaf;
13:   if(lastLeaf > MaxIndex)
14:      lastRoot = lastLeaf = lastLeaf/4;
15:   a[lastLeaf] = x;
16: }
```

Figure 5. Insert Operation

Theorem 1: The insert( ) operation takes constant amortized time to insert an element into a 4-ary AM-heap.

Proof: As an element is inserted closer to the root, it takes more time since heapify( ) percolates it down up to a leaf node. So, it may take the most time when an element is inserted into the root of a 4-ary AM-heap. Therefore, starting with an initially empty heap, we first calculate the total time *T* of a sequence of *n* insertions until the heap becomes full. Then we can obtain the worst case amortized time complexity *O(T)/n* of insert( ). Assuming that tree level starts from 1 at the root, the total time

$$T = \mathrm{h} + 3\left(\sum_{i=0}^{h-2} 4^i (h-i-1)\right) = O\left(4^h\right)$$

where h is the height of a 4-ary AM-heap.
Since the total number of nodes $\mathrm{n} = 1 + 3\left(\sum_{i=0}^{h-2} 4^i\right) = 4^{h-1}$ when the heap is full,

$$\mathrm{T} = O\left(4^h\right) = O(n).$$

Therefore, the amortized time complexity of the insert( ) operation is *O(n)/n = O(1)*.

To remove the minimum element from a 4-ary AM-heap, as can be seen in Figure 6, we should first find node *minRoot* with the minimum element by traversing the roots of all the component heaps in the heap, starting with node *lastRoot* (lines 4 through 8). In general, starting from the root *r* of a component heap, the root of its left neighbor component heap can be found by following up its parent while the node index is a multiple of *4* and by decrementing its index by one. At lines 9 and 10, the element in node *lastRoot* is moved to node *minRoot* and heapify( ) operation is performed at node *minRoot*. Now that we have to remove node *lastRoot* physically, the two variables *lastRoot* and *lastLeaf* should be adjusted as at lines 12 through 25. If node *lastRoot* has children, its rightmost child becomes *lastRoot*. The heap becomes empty if there was only one element in the heap. Otherwise, we find the root of the left neighbor component heap to set *lastRoot*. Node *lastLeaf* is just the rightmost leaf node of the component heap rooted at *lastRoot*.

```
 1: removeMin( )
 2: {
 3:    if( heap is empty ) return;

 4:    minRoot = curRoot = lastRoot;
 5:    while( curRoot != a power of 4 ) {
 6:      curRoot = root of left neighbor component heap;
 7:      if( a[minRoot] > a[curRoot] )minRoot = curRoot;
 8:    }
 9:    a[minRoot] = a[lastRoot];
10:    heapify( minRoot );

11:    // update variables  lastRoot and lastLeaf;
12:    if( lastRoot has a child )
13:       lastRoot  = lastRoot*4 + 3; // rightmost child
14:    else if( lastLeaf == StartLeaf ) {
15:       // only a single element was in the heap.
16:       lastLeaf--;
17:       lastRoot = -2;  // heap is empty now.
18:    } else {
19:       lastLeaf--;
20:       if( (lastLeaf*4) < MaxIndex )
21:          lastLeaf = MaxIndex;

22:       // lastRoot = the root of left component heap.
23:       lastRoot = lastLeaf;
24:       while( (lastRoot+1) % 4 == 0 ) lastRoot /= 4;
25:    }
26:}
```

Figure 6. RemoveMin Operation

Theorem 2: The removeMin( ) operation takes *O*(log*n*) time to delete the minimum element where *n* is the number of element in a 4-ary AM-heap.

Proof: First, notice that both the number of component heaps and the height of the largest component heap are *O*(log*n*) [1,2]. To delete the minimum element, node *minRoot* with the minimum element is searched which takes *O*(log*n*) time (lines 4~8). The heapify( ) operation at line 10 takes *O*(log*n*). Updating *lastRoot* at line 24 also takes *O*(log*n*). Therefore, the removeMin( ) operation has *O*(log*n*) time complexity.

4. Experimental Results

We implemented the AM-heap, 4-ary AM-heap, cache-aligned 4-ary AM-heap and post-order heap in C++ to obtain experimental evaluation of the cache-aligned 4-ary AM-heap relative to AM-heap. The codes were compiled using GNU g++ with maximum optimization option –O3 and run on a PC with Q8200 CPU and 3GB(Giga Bytes) of main memory.

For run time tests, we used four test models, as shown in Figure 7.

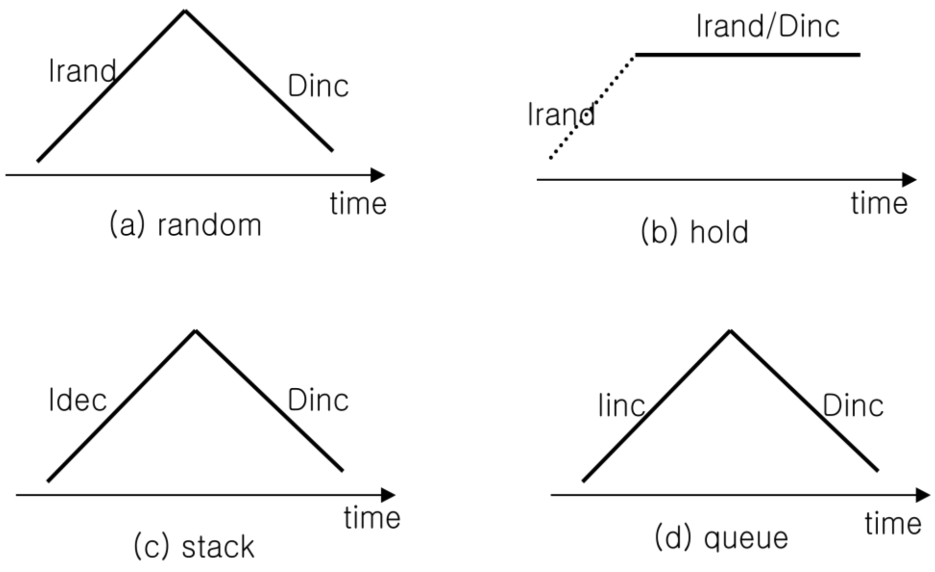


Figure 7. Test Models (I: Insert, D: Delete, Rand: Random Order, inc: Increasing Order, dec: Decreasing Order)

- Hold model [9]: An initially empty heap is initialized with *n* random elements and an intermixed sequence of 2*n* insert and delete operations is performed. During this sequence of *2n* operations is performed, the number of elements in the heap remains at roughly *n*. The time taken for the intermixed sequence of 2*n* insert and delete operations is measured.
- Random model: In this test model, *n* random elements are inserted into an initially empty heap and *n* elements are deleted in increasing order. The time taken for the *2n* operations is measured.
- Stack model: *n* elements are inserted into an initially empty heap in decreasing order. Then all these elements are deleted in increasing order. The time to perform these *2n* operations is measured.
- Queue model: n elements are inserted into an initially empty heap in increasing order and then all these elements are removed in increasing order. The time taken for the *2n* operations is measured.

Figures 8 through 15 show the run-times and speedup relative to HeapAM for the four test models. The figures use the following abbreviations: HeapAM(AM-heap), HeapAM04CU (cache-unaligned 4-ary AM-heap), HeapAM04 (cache-aligned 4-ary AM-heap), HeapPO(post-order heap), K(kilo), and M(mega).

For every test model, the cache-aligned 4-ary AM-heap showed the best results: its maximum speedup relative to AM-heap was about 25 percent in the hold model. Also, it was slightly faster than AM-heap for the stack and queue models. On the other hand, the post-order heap showed the worst performance for all the test models. The performance of the cache-unaligned 4-ary AM-heap was in between AM-heap and cache-aligned 4-ary AM-heap.

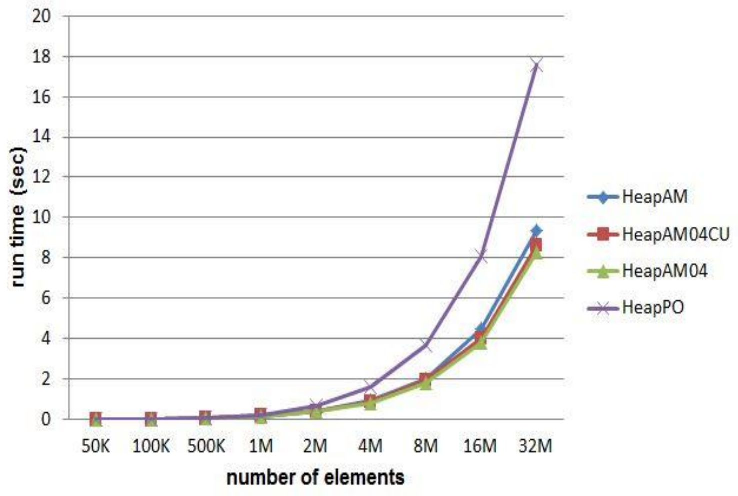


Figure 8. Run Time in Hold Model

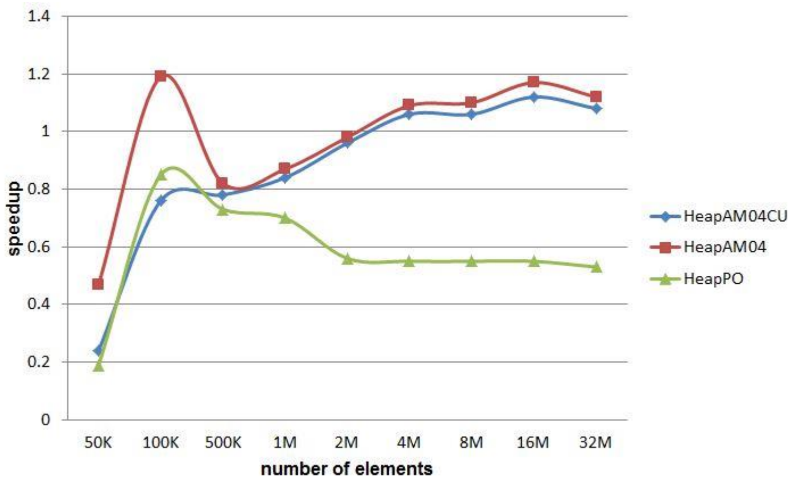


Figure 9. Speedup Relative to AM-heap in Hold Model

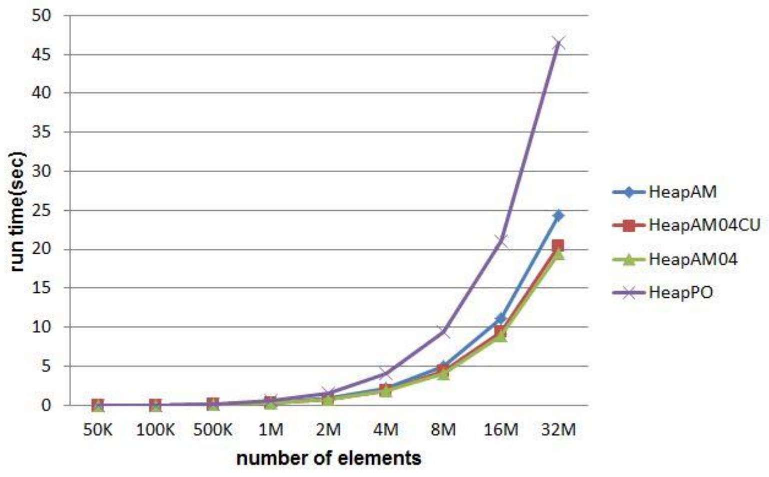


Figure 10. Run Times in Random Model

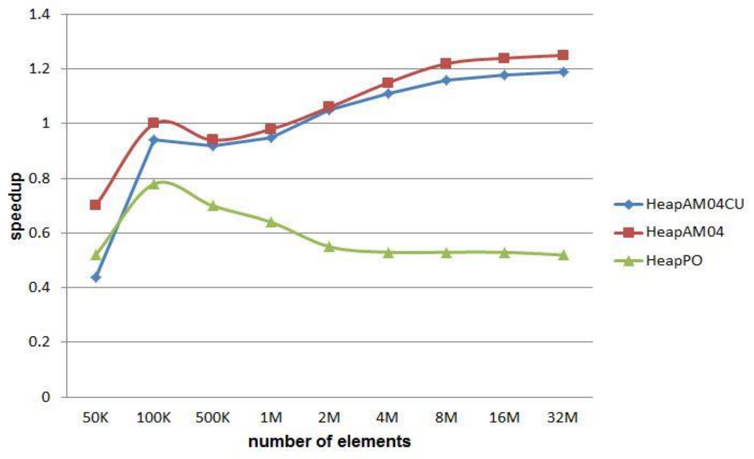


Figure 11. Speedup Relative to AM-heap in Random Model

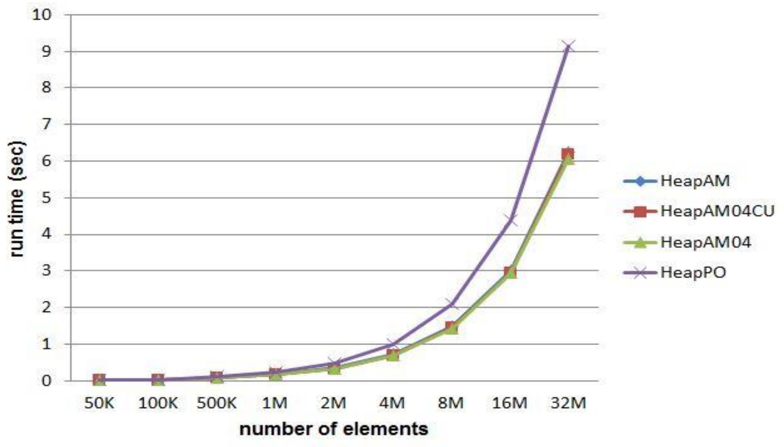


Figure 12. Run Times in Stack Model

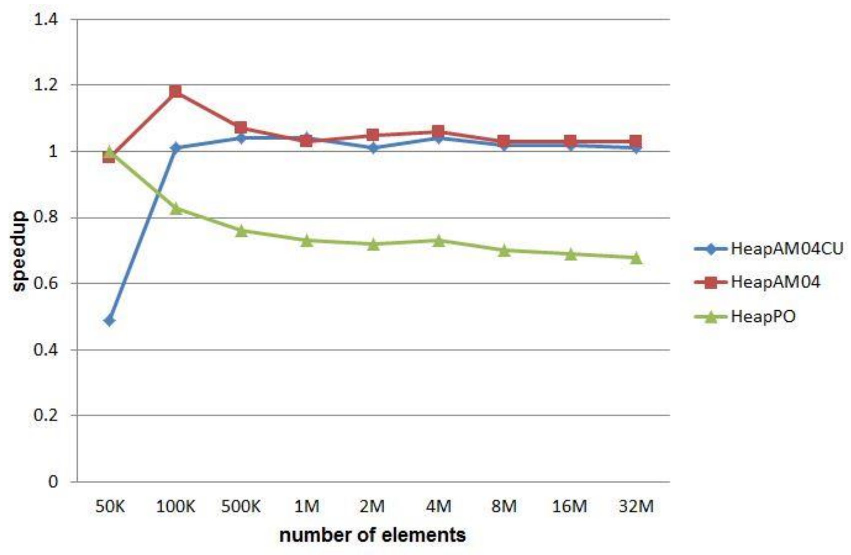


Figure 13. Speedup Relative to AM-heap in Stack Model

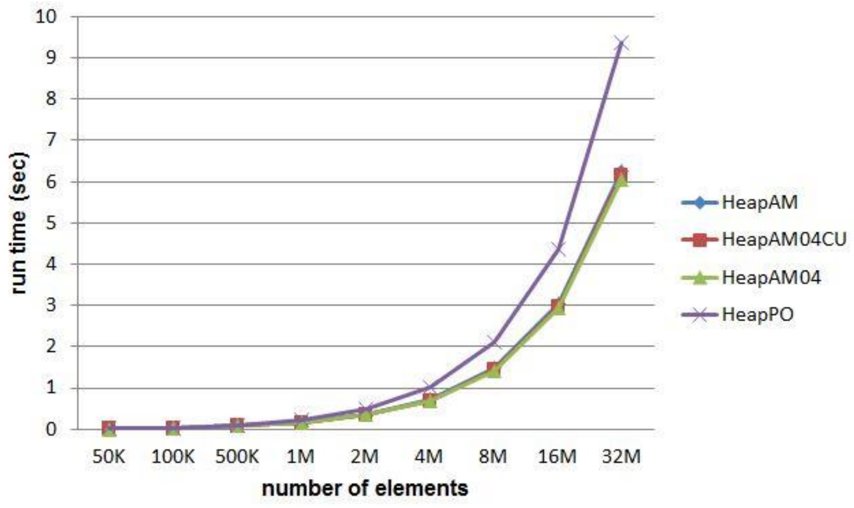


Figure 14. Run Times in Queue Model

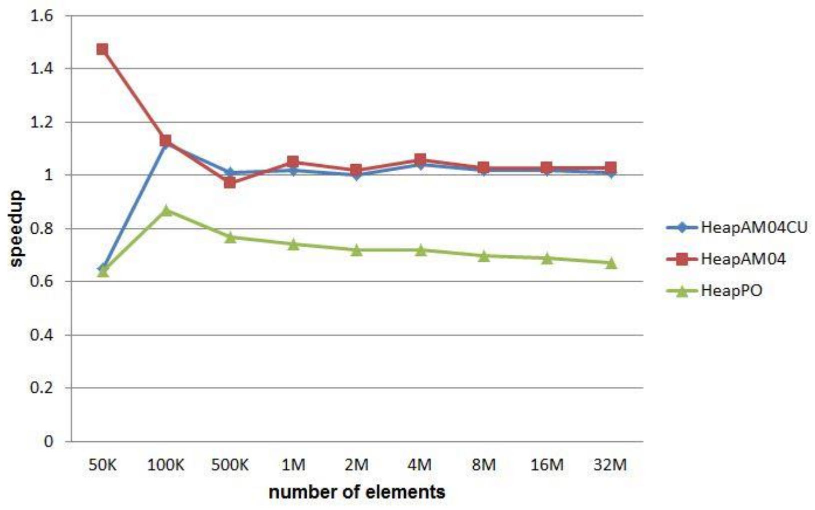


Figure 15. Speedup Relative to AM-heap in Queue Model

5. Conclusion

We have proposed the cache-aligned 4-ary AM-heap. It uses a simple indexing scheme in which the first child of every node has an index of a multiple of 4. Also, it efficiently utilizes cache memory by fitting all the children of a node into a single cache block. Our experimental results show that the performance of the cache-aligned 4-ary AM-heap is the best among implicit heaps with constant amortized insertion and logarithmic deletion time complexities.

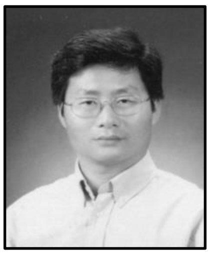
Acknowledgements

This work was supported by a grant from 2014 Research Fund of Andong National University.

References

[1] N. Harvey and K. Zatloukal, "The Post-order heap", Proceedings of the Third International Conference on Fun with Algorithms(FUN), **(2004)** May.
[2] H. Jung, "A Simple Array Version of M-Heap", International Journal of Foundations of Computer Science, vol. 25, no. 1, **(2014)**.
[3] J. Williams, "Algorithm 232 Heapsort", Communications of the ACM, vol. 7, no. 1, **(1964)**.
[4] S. Carlsson, P. Poblete and J. Munro, "An implicit binomial queue with constant insertion time", Lecture Notes in Computer Science, vol. 318, **(1988)**, pp. 1-13.
[5] E. Horowitz, S. Sahni and D. Mehta, "Fundamentals of Data Structures in C++ (2nd Edition)", Silicon Press, Summit, **(2007)**.
[6] A. LaMarca and R. Ladner, "The influence of Caches on the Performance of Heaps", ACM Journal of Experimental Algorithmics, vol. 1, no. 4, **(1996)**.
[7] H. Jung and S. Sahni, "Supernode binary search trees", International Journal of Foundations of Computer Science, vol. 14, no. 3, June **(2003)**, pp. 465-490.
[8] H. Jung, "The d-deap*: A fast and simple cache-aligned d-ary deap", Information Processing Letters, vol. 93, no. 2, **(2005)** January, pp. 63-67.
[9] D. Jones, "An empirical comparison of priority-queue and event-set implementation", Communications of the Association for Computing Machinery, vol. 29, no. 4, **(1986)**, pp. 300–311.
[10] M. Fredman, R. Sedgewick, R. Sleator and R. Tarjan, "The pairing heap: A new form of self-adjusting heap", Algorithmica, vol. 1, **(1986)** March, pp. 111-129.
[11] T. J. Stasko and J. S. Vitter, "Pairing heaps: experiments and analysis", Communications of the ACM, vol. 30, no. 3, **(1986)** March, **(1987)**, pp. 234-249.
[12] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", Journal of the ACM, vol. 34, no. 3, **(1987)**, pp. 596-615.
[13] J.-L. Yu, C.-H. Choi, D.-S. Jin, J. Ruth Lee and H.-J. Byun, "A Dynamic Virtual Machine Allocation Technique Using Network Resource Contention for a High-performance Virtualized Computing Cloud", IJSEIA, vol. 8, no. 9, **(2014)** September, pp. 17-28.
[14] J. Lee and G.-L. Park, "Cluster-based Vehicle Redistribution scheme based on Genetic Algorithms for Electric Vehicle Sharing Systems", IJSEIA, vol. 8, no. 9, **(2014)** September, pp. 147-158.
[15] Z. Rizal M Azmi, K. Abu Bakar, M. Shahir Shamsir, W. Nurulsafawati Wan Manan and A. Hanan Abdullah, "Scheduling Grid Jobs Using Priority Rule Algorithms and Gap Filling Techniques", IJAST, vol. 37, **(2011)** December, pp. 61-76.
[16] H. Jung, "The 4-ary AM-heap", FGCN 2014(The 8th International Conference on Future Generation Communication and Networking), **(2014)** to appear.

Author



Haejae Jung, he received Ph.D. degree in Computer & Information Science & Engineering from University of Florida in 2000. He has been with the department of information and communication engineering at Andong National University since 2005. His research interests are in the design of efficient computer algorithms.