KIPS Trans. Comp. and Comm. Sys.
Vol.7, No.12 pp.289~294 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2018.7.12.289

Implicit D-Ary AM-Heap

Haejae Jung†

ABSTRACT

This paper proposes an implicit d-ary priority queue, called AM(d)-heap that is a generalized version of AM-heap, in which insert operation takes constant amortized time and remove operation takes O(logn) time. According to our experimental results, the best performance was shown when d is 4 or 8. Also, AM(d)-heap is about 1.5~1.8 times faster than the postorder heap.

Keywords : Priority Queue, Implicit Heap, Amortized Time Complexity, Data Structures

묵시 다원 AM-힙

정 해 재†

요 약

본 논문에서는 AM-힙의 다원 버전인 AM(d)-힙이라고 하는 묵시 다원 우선순위 큐를 제안하며, 제안된 AM(d)-힙에서는 삽입에 O(1) 전이시간이 걸리고 삭제 연산에 O(logn) 시간이 걸린다. 실험 결과에 따르면, 전체적으로 d가 4 또는 8일 때 가장 우수한 성능을 나타내었다. 기존의 후위힙과 비교하면 AM(d)-힙이 약 1.5~1.8배 빠른 것으로 나타났다.

키워드 : 우선순위 큐, 묵시 힙, 전이 시간 복잡도, 자료 구조

1. 서 론

힙은 스케줄링, 시뮬레이션, 검색 결과로부터의 우선순위에 따른 추출, 정렬, 최단 거리 계산 등에 활용할 수 있는 근본적인 자료구조 중의 하나로서, 지금까지 오랜 기간 동안 많은 연구가 이루어져 왔다. 우선순위 큐에는 최소키 삭제를 지원하는 최소 우선순위 큐와 최대키 삭제를 지원하는 최대 우선순위 큐가 있는데, 본 논문에서는 별다른 언급이 없는 한 최소 우선순위 큐에 대해 이야기하는 것으로 간주한다. 최소힙은 임의의 키 삽입과 최소키 삭제를 빠르게 지원하는 우선순위 큐이며, 원소 집합 S에 대해 다음의 기본 연산을 효율적으로 지원한다.

- insert(S, e) : 집합 S에 임의의 원소 e를 삽입.
- removeMin(S) : 집합 S로부터 최소키를 삭제.

※ 이 논문은 안동대학교 기본연구지원사업에 의하여 연구되었음.
† 종신회원 : 안동대학교 정보통신공학과 교수
Manuscript Received : May 8, 2018
First Revision : August 13, 2018
Accepted : September 27, 2018
* Corresponding Author : Haejae Jung(hjjung@anu.ac.kr)

힙은 포인터를 사용하는 힙과 포인터를 사용하지 않는 묵시힙(implicit heap)으로 분류할 수 있다.

포인터를 사용하는 힙 중에서 상수 삽입 전이 시간복잡도를 가지는 힙으로는 피보나치 힙, 페어링 힙, M-힙 등이 있다[1-4]. 이들 자료 구조는 배열만을 사용하는 묵시힙에 비해 포인터 사용공간이 더 필요하고, 삽입 및 삭제 연산시의 포인터 수정 시간이 더 필요하게 된다.

대표적인 묵시힙(implicit heap)으로는 전통힙(conventional binary heap), 묵시 이항 큐 (implicit binomial queue), 후위힙(postorder heap), 및 AM-힙(AM-heap)이 있다[5-9]. 전통힙은 삽입과 삭제에 모두 O(logn) 시간이 걸리고, 나머지 세 힙은 삽입과 삭제에 각각 상수 전이 시간과 O(logn) 시간이 걸린다.

후위힙은 모든 리프 노드가 동일한 레벨에 있는 포화 이진트리 구조를 가지며, 전통힙과는 달리 배열에 대응하는 트리 노드 인덱스가 후위 순서(post-order)로 매겨진다[8]. 키 삽입 역시 후위 순서로 이루어지며, 실질적인 키 삭제는 후위 순서의 반대로 이루어진다.

AM-힙은 이진트리 구조를 가지며, 전통힙과 같이 레벨 순서로 노드 인덱스가 부여된다[9]. 따라서 AM-힙은 삽입

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/ licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

및 삭제 시간 복잡도에 있어서 후위힙과 동일하나, 노드 인덱싱 구조가 간단하여 구현하기 더 용이하고 다원 힙으로 확장도 용이해진다. 또한 포화 이진 트리 구조를 가지는 후위힙과는 달리 AM-힙은 완전 이진 트리(complete binary tree) 구조를 가져 메모리 활용도가 높게 된다. AM-힙에서도 키 삽입은 후위 순서로 이루어지고, 실질적인 키 삭제는 그 반대로 이루어진다. 키 삽입과 삭제의 순서로 인하여 AM-힙은 O(logn) 개의 성분힙(CH: component heap)으로 구성된다. 성분힙의 높이는 가장 오른쪽 두 개를 제외하고는 모두 다르게 되고, 가장 왼쪽에서 오른쪽으로 가면서 성분힙의 높이가 낮아진다.

본 논문에서는 AM-힙을 일반화한 묵시 다원 AM-힙(implicit d-ary AM-heap)인 AM(d)-힙을 제안하고, 트리 차수 d에 따른 성능 평가 결과를 보인다. AM(d)-힙도 AM-힙과 같이 O(1) 삽입 전이 시간과 O(logn) 삭제 시간을 가진다.

2. AM(d)-힙

AM(d)-힙은 완전 트리로 표현되며, AM-힙 또는 MA-힙과 같이 최대 O(logn) 개의 성분힙(component heap)으로 구성된다 [9,10]. Fig. 1은 노드 1, 8, 9, 10에 루트를 가진 4개의 성분힙으로 구성된 4원 AM-힙, 즉 AM(4)-힙의 예를 보이고 있다. 그림에서 lastLeaf는 키를 가진 노드중 가장 오른쪽 리프 노드인 노드 10를 가리키고, lastRoot는 가장 오른쪽 즉 마지막 성분힙의 루트 노드 10을 가리킨다.

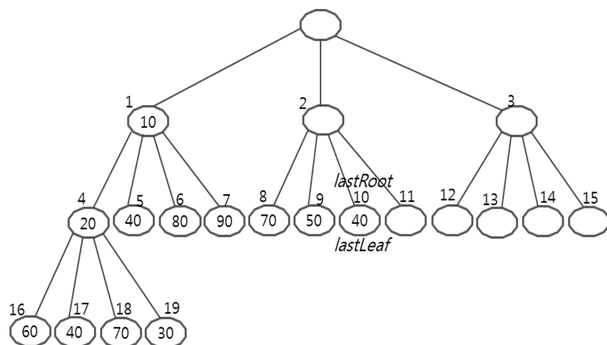

Fig. 1. AM(4)-heap

AM(d)-힙에서 루트는 최대 d-1개의 자식을 가지고, 그 외의 모든 내부 노드는 최대 d개의 자식을 가진다. 이러한 구성으로 인하여 어떤 노드의 가장 왼쪽 자식은 d의 배수 인덱스를 가지고, 각 수준(level)의 가장 왼쪽 노드는 d의 멱승 인덱스를 가지게 된다.

최소 AM(d)-힙의 특성은 다음과 같다.

1. 완전 트리(complete tree)이다.
2. 루트는 d-1개까지의 자식을 가질 수 있고, 그 외의 모든 내부 노드는 d개까지 가질 수 있다.
3. n개의 노드를 가진 d-ary AM-힙은 최대 $O(\log_d n)$개의 성분힙을 가진다.
4. 각 성분힙의 높이는 왼쪽에서 오른쪽으로 가면서 낮아진다. 높이가 동일한 성분힙의 개수는 최대 d개이며, 이들은 제일 오른쪽 즉, 마지막에 위치한다.
5. 각 성분힙은 최소 전통힙이다.

2.1 초기화

Fig. 2에서 보이고 있는 AM(d)-힙 초기화 코드에서는 maxN 개의 키를 저장하기에 충분한 크기의 배열 nar[ ]을 할당하고, 제일 마지막 노드 인덱스를 maxIndex로 둔다. 이때, 배열 nar[ ]의 크기는 maxN개의 키를 수용할 수 있도록 하면서 마지막 내부 노드가 d개의 자식을 갖도록 하고 있다. 힙이 공백이라는 것을 표현하기 위해 마지막(가장 오른쪽) 성분힙의 루트를 나타내는 변수 lastRoot를 상수 HEAP_EMPTY으로 초기화하고, 변수 lastLeaf는 AM(d)-힙의 제일 왼쪽 리프 노드 인덱스인 startLeaf보다 1 적은 값으로 초기화 한다.

```
1: Algorithm initialize( maxN )
2: {
3:     HEAP_EMPTY = -2;
4:     HEAP_FULL = d-1;  // node 0 is unused.
5:
6:     maxIndex = (maxN/d + 1) * d - 1;
7:     nar = new keyType[maxIndex+1];  // node
   array
8:     t = log_d(maxIndex);
9:     startLeaf = d^t;
10:
11:     lastRoot = HEAP_EMPTY;    // -2: empty
    heap
12:     lastLeaf = startLeaf - 1;
13: }
```

Fig. 2. Initialization of AM(d)-heap

2.2 힙조정

인수로 받은 키 theKey가 노드 r에 삽입될 경우, 하향식으로 힙조정이 이루어지는데, 그 알고리즘은 Fig. 3에 나타나 있다. 함수 heapify( )의 인수 r은 새로운 키 theKey를 삽입할 노드 인덱스를 나타낸다. while 루프에서 노드 r의 자식들 중 가장 작은 키를 가진 노드 min을 찾고, 그 키가 theKey보다 크면(라인 9) 노드 r에 theKey를 삽입한다(라인 13). 그렇지 않으면, 노드 min의 키를 부모 노드에 복사해 올리고 한 단계 내려와서(라인 10-11) 위의 과정을 반복한다.

```
1: Algorithm heapify(r, theKey) // starting from
   node r
2: {
3:     min = r * d;    // min: leftmost child
4:     while( min < maxIndex ) {
5:          rmc = min + (d-1);  // rmc: rightmost
   child
6:          for( i = min+1; i <= rmc; i++ )
7:               if( nar[min] > nar[i] ) min = i;
8:
9:          if( theKey <= nar[min] ) break;
10:          nar[r] = nar[min];
11:          r = min; min = d*r;
12:     }
13:     nar[r] = theKey;
14: }
```

Fig. 3. Heapify operation of AM(d)-heap

2.3 삽입

Fig. 4는 AM(d)-힙 삽입 알고리즘을 보여주고 있다. 인수로 전달된 theKey를 삽입하기 위하여 (lastRoot+1)이 d의 배수인지 알아본다. d 배수인 경우 인수로 전달된 theKey를 lastRoot의 부모 노드에 삽입하고 힙조정을 하기 위해 heapify( ) 함수를 호출한다(라인 7-8). 그렇지 않으면, lastRoot와 lastLeaf를 하나의 노드로 구성된 새로운 성분힙을 가리키게 하고 theKey를 이 새로운 노드에 삽입한다(라인 11-14).

```
1: Algorithm insert( theKey )
2: {
3:    if(     lastRoot==HEAP_FULL     )     throws
   FullException;
4:
5:    if( (lastRoot+1) == a multiple of d ) {
6:        // insert theKey into the parent of node
   lastRoot
7:        lastRoot /= d;
8:        heapify( lastRoot, theKey );
9:    } else {
10:        // make a new component heap
11:        lastRoot = ++lastLeaf;
12:        if(lastLeaf > maxIndex)
13:            lastRoot = lastLeaf = lastLeaf/d;
14:        nar[lastLeaf] = theKey;
15:    }
16: }
```

Fig. 4. Insertion of AM(d)-heap

Fig. 5는 Fig. 1에 키 55와 45를 연속으로 삽입한 후의 AM(4)-힙을 보여주고 있다. 키 55가 삽입된 후에는 lastRoot와 lastLeaf 모두 노드 11을 가리키고, 키 45 삽입시 lastRoot는 2가 되고 heapify( )에 의해 노드 2와 노드 10의 키가 교환된다.

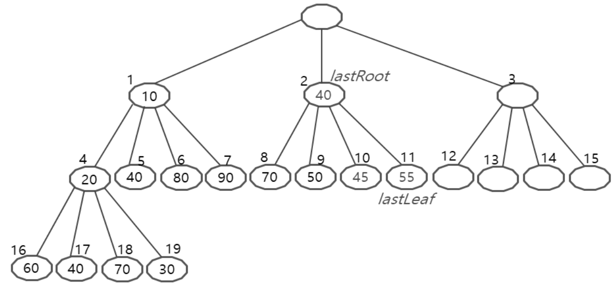

Fig. 5. AM(4)-heap after Insertions of 55 and 45 into Fig. 1 in Sequence

(정리 1) n 개의 키를 가지고 있는 AM(d)-힙에서, 삽입 연산은 상수 전이 시간 복잡도 (amortized time complexity)를 가진다.

(증명) 삽입 전이 시간 복잡도를 구하기 위해, 일련의 삽입 연산이 이루어진다고 하자. 일련의 삽입 중, 같은 높이의 이웃하는 d 개의 성분힙으로 구성된 AM(d)-힙에 새로운 키를 삽입할 경우 하나의 성분힙만으로 구성된 AM(d)-힙이 되는데, 이때 최대 비용이 소요된다. 힙조정은 이웃하는 d 성분힙의 부모 노드에서 시작하여 리프 노드 쪽으로 이루어지므로, 그 부모 노드를 루트로 하는 서브트리의 높이만큼의 시간이 걸린다. 따라서, 하나의 성분힙으로 구성된 AM(d)-힙의 높이가 h라 할 때, 일련의 삽입 총 비용을 계산하면 $T=\sum_{l=1}^{h}(h-l+1)d^{l-1}$ 가 된다. 즉, 루트 레벨 1로 시작하는 트리 레벨 l에는 d^{l-1} 개의 노드가 있고, 각 노드의 키 삽입 비용은 $(h-l+1)$이 된다. 여기서 $i=h-l+1$로 두면 $T=\sum_{i=1}^{h}(i)d^{h-i}=d^{h}\sum_{i=1}^{h}(i/d^{i})=O(d^{h}\cdot 2)=O(n)$이 된다. 따라서, 키 각각에 대한 삽입 전이 시간 복잡도는 $O(n)/n=O(1)$이 된다.

2.4 최소키 삭제

최소키는 모든 성분힙의 루트 키 중 가장 작은 값이므로, 먼저 모든 성분힙의 루트를 조사하여 가장 작은 키를 가지는 노드를 찾고, 그 노드의 키를 lastRoot 노드의 키로 대체한다. 대체된 노드를 루트로 하는 성분힙에 대해 힙조정을 수행하고, lastRoot와 lastLeaf 값을 적절하게 수정함으로서 삭제가 완성된다.

Fig. 6의 삭제 알고리즘에서 먼저 최소키를 가지고 있는 minRoot를 찾는다(라인 5-9). minRoot는 lastRoot로부터 시작하여 왼쪽 인접 성분힙의 루트를 찾는 것을 반복함으로서

이루어진다. 왼쪽 인접 성분힙의 루트는 현재 성분힙의 루트 인덱스를 d로 나눈 나머지가 0이 아닐 때까지 반복한 후 1을 빼면 된다. 또한 각 레벨의 가장 왼쪽 성분힙의 루트 firstRoot는 d의 멱승이므로 상수 시간에 검사할 수 있다. 이렇게 찾은 minRoot의 키를 lastRoot의 키로 대체하고 삽입하기 위해 힙조정 함수 heapify( )를 호출한다.

최소키 삭제 후, lastRoot와 lastLeaf를 수정한다. lastRoot가 내부노드인 경우 그 노드의 가장 오른쪽 자식이 lastRoot가 되고(라인 15), 리프노드인 경우 인접한 왼쪽 성분힙의 루트가 lastRoot가 된다(라인 21-26).

```
1:  Algorithm  removeMin( )
2:  {
3:      if(lastRoot==HEAP_EMPTY)throws EmptyException;
4:
5:      minRoot = curRoot = lastRoot;
6:      while( curRoot != firstRoot ) {
7:        curRoot = root of left neighbor CH;
8:        if( a[minRoot] > a[curRoot] )minRoot = curRoot;
9:      }
10:     removedKey = nar[minRoot]; // save to return
11:     heapify( minRoot, nar[lastRoot] );
12:
13:     // update variables lastRoot and lastLeaf;
14:     if( lastRoot has a child )
15:         lastRoot = lastRoot*d + (d-1); //rightmost child
16:     else if( lastLeaf == startLeaf ) {
17:       // only a single CH was in the heap.
18:         lastLeaf--;
19:         lastRoot = HEAP_EMPTY; // empty heap now.
20:     } else {
21:         lastLeaf--;
22:         if( (lastLeaf*d) < maxIndex )
23:             lastLeaf = maxIndex;
24:         // lastRoot= root of left neighbor CH.
25:         lastRoot = lastLeaf;
26:         while( (lastRoot+1) % d == 0 ) lastRoot /= d;
27:     }
28:     return removedKey;
29: }
```

Fig. 6. Remove Algorithm of AM(d)-heap

(정리 2) AM(d)-힙에서 최소키 삭제는 $O(\log n)$ 시간 걸린다. 여기서 n은 AM(d)-힙에 있는 키의 개수이다.

(증명) AM(d)-힙에 최대 $O(\log n)$ 개의 서로 다른 높이의 성분힙이 존재할 수 있으므로 최소키를 가진 성분힙의 루트 노드를 찾는데 $O(\log n)$ 시간이 걸리고, 힙조정 알고리즘 heapify( )는 힙의 높이만큼의 시간이 걸리므로 $O(\log n)$ 시간이 걸린다. 또한 lastRoot 변수를 수정하는데 트리 높이인 $O(\log n)$ 시간이 걸린다. 따라서, 최소값 삭제 알고리즘은 $O(\log n)$ 시간 복잡도를 가진다. 여기서 log함수의 밑수 d는 상수이므로 점근 표기법에서 생략했다.

Fig. 7은 Fig. 5로부터 최소키 10을 삭제한 후의 AM(4)-힙을 보여주고 있다. 먼저 minRoot인 노드 1의 최소키 10을 lastRoot의 키 40으로 대체하고 노드 1에 대해 힙조정을 한다. 그 후, lastRoot는 노드 11을 가리키도록 수정된다.

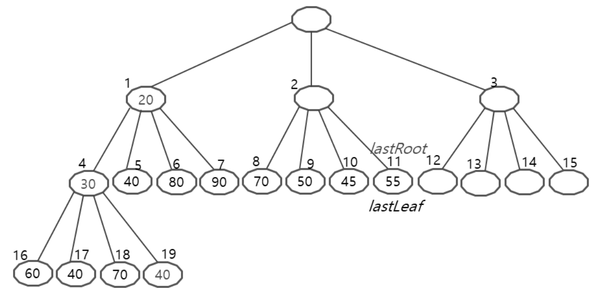

Fig. 7. AM(4)-heap after Removemin from Fig. 5

3. 실험 결과

성능 시험을 위하여 상수 삽입 전이 시간과 O(logn) 삭제 시간이 걸리는 묵시힙인 후위힙과 AM(d)-힙 (d = 2, 4, 8, 16, 32)을 구현하였다. 모든 프로그램은 GNU g++의 최고 최적화 수준 옵션 O3를 주고 컴파일 하였다. 구현은 노드를 캐쉬 정합(cache-alignment)이 되도록 구현하였다. 실험은 4GB 메모리를 가지고 있는 윈도우7 PC 환경에서 유지(hold), 스택(stack), 큐(queue), 및 편향(skew) 4가지 성능 시험 모델에 대해 이루어졌다[11, 12]. 유지모델에서는 n개의 키로 힙을 초기화 한 후, 힙에 n개의 키를 유지하면서 n개의 키 삽입과 삭제를 무작위로 실행한 시간을 측정하였다. 스택모델에서는 n개의 키를 내림차순으로 삽입하고 그 n개의 키를 오름차순으로 삭제한 시간을 측정하였다. 큐모델이서는 n개의 키를 오름차순으로 삽입하고 그 n개의 키를 오름차순으로 삭제하는 시간을 측정하였다. 편향모델에서는 n개의 키를 무작위로 삽입하고 그 중 5%을 오름차순으로 삭제하는 시간을 측정하였다. 여기서, 스택과 큐 모델은 최악의 데이터 분포에 대한 실험이고, 나머지 모델은 무작위 분포에 대한 실험이다. 실험은 키의 개수 n을 10K(K=1,000), 50K, 100K, 500K, 1M(M=1,000,000), 2M, 4M, 8M, 16M, 32M로 변경하면서 이루어졌고, 각각에 대해 9회 반복한 평균 시간을 측정하였다.

Fig. 8은 실험 결과를 보여주고 있다. 유지모델에서 d가 2, 4, 16일 때, 스택과 큐 모델에서는 d가 4, 8일 때, 편향모델에서는 4, 8, 16일 때 우수한 성능을 나타내었다. 편향모델에서 d가 16일 때 가장 우수한 성능을 보여 삽입 연산이 가장 우수한 것으로 나타났다. 전체적으로 보면 d가 4 또는 8일 때

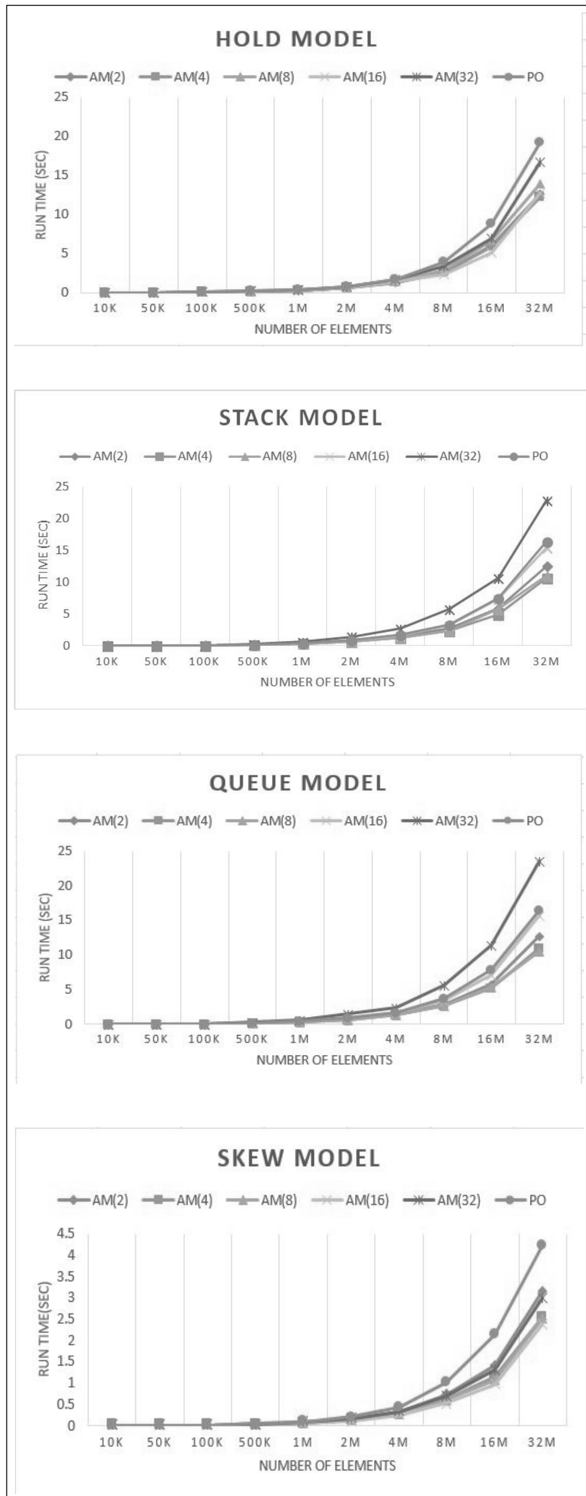

Fig. 8. Experimental Results (PO: Post-order Heap)

우수한 성능을 보이고 있다. 또한 AM(d)-힙이 후위힙에 비해 1.5~1.8배 빠른 것으로 나타났다. 후위힙에서의 왼쪽 이웃 성분힙의 루트를 찾을 때나 자식 노드를 찾을 때의 복잡한 계산식이 성능 저하의 요인이 되고 있다.

위와 같은 실험 결과는 삽입과 삭제 연산시 키 비교 회수와 캐쉬 정합 효과에 기인하는 것으로 분석된다. 각 레벨에서 부모와 자식 간의 비교 회수는 d이므로 d가 커지면 비교회수가 많아진다. 따라서 d가 32인 경우 성능이 저하된 것을 볼 수 있다. 또한 노드가 캐쉬 정합되어 있으면 최대 1번의 캐쉬 미스가 발생하여 최고의 성능을 가져올 수 있다. 캐쉬 라인 크기가 32바이트이고 4바이트 크기의 키의 경우 d가 8일 때 최선의 성능을 가진다.

4. 결 론

본 고에서는 AM-힙을 일반화 한 묵시 다원 AM-힙인 AM(d)-힙 구조 및 알고리즘을 제안하고, AM(d)-힙에서의 삽입은 상수 전이 시간이 걸리고 삭제는 O(logn) 시간이 걸림을 보였다.

또한, 여러 d 값과 다양한 성능 시험 모델을 이용하여 성능 평가를 실시하였다. 실험 결과, 삽입 연산이 많은 경우 d가 16일 때 가장 우수한 성능을 보이고, 전체적으로는 d가 4 또는 8일 때 가장 우수한 성능을 나타내었다. 이러한 성능은 후위힙에 비해 1.5~1.8배 우수한 것으로 나타났다.

References

[1] M. Fredman and R. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *Journal of the ACM*, Vol.34, No.3, pp.596-615, 1987.

[2] M. Fredman, R. Sedgewick, R. Sleator, and R. Tarjan, “The pairing heap: A new form of self-adjusting heap,” *Algorithmica*, Vol.1, pp.111-129, Mar. 1986.

[3] T. J. Stasko and J. S. Vitter, “Pairing heaps: experiments and analysis,” *Communications of the ACM*, Vol.30, No.3, pp.234-249, 1987.

[4] S. Bansal, S. Sreekanth, and P. Gupta, “M-heap: A Modified heap data structure,” *International Journal of Foundations of Computer Science*, Vol.14, No.3, pp.491-502, 2003.

[5] J. Williams, “Algorithm 232 Heapsort,” *Communications of the ACM*, Vol.7, No.1, pp.347-348, 1964.

[6] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*, W. H. Freeman, San Francisco, 1995.

[7] S. Carlsson, J. Munro, and P. Poblete, “An implicit binomial queue with constant insertion time,” *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science*, Vol.318, pp.1-13, Jul. 1988.

[8] N. Harvey and K. Zatloukal, “The Post-order heap,” in *Proceedings of the Third International Conference on Fun with Algorithms (FUN)*, May 2004.

[9] H. Jung, “A Simple Array Version of M-heap,” *International Journal of Foundations of Computer Science*, Vol.25, No.1,

pp.67-88, Jun. 2014.

[10] H. Jung, "Insertion/Deletion Algorithms on M-heap with an array representation," *The KIPS Transations: Part A*, Vol.13, No.A(3), pp.261-266, Jun. 2006 (written in Korean).

[11] H. Jung, "A Performance Evaluation of Priority Queue Algorithms," *The Journal of Korean Institute of Information Technology*, Vol.8, No.6, pp.97-103, Jun. 2010.

[12] D. Jones, "An empirical comparison of priority-queue and event-set implementation," *Communications of the Association for Computing Machinery*, Vol.29, No.4, pp.300-311, 1986.

정 해 재

https://orcid.org/0000-0002-6538-168X

e-mail : hjjung@anu.ac.kr

1984년 경북대학교 전자공학과(학사)

1987년 서울대학교 컴퓨터공학과(석사)

1988년∼1995년 ETRI 선임연구원

2000년 CISE Univ. of Florida(Ph.D.)

2001년∼2002년 Numerical Tech, Inc. USA, Staff Engineer

2005년∼현 재 안동대학교 정보통신공학과 교수

관심분야 : Data Structures and Algorithms, Database and Web, Data Mining