# 배열 표현을 이용한 M-힙에서 삽입/삭제 알고리즘

정 해 재†

## 요 약

스케줄링, 정렬, 및 최단 거리 계산 네트워크 문제 등과 같은 응용에 이용될 수 있는 우선 순위 큐 중, 피보나치 힙, 페어링 힙, 및 M-힙은 포인터를 이용하는 자료 구조이다.

본 논문에서는 [1]에서 문제점으로 남겨두었던 M-힙을 배열을 이용하여 표현한 MA-힙(M-heap with an array representation)를 제안한다. MA-힙은 M-힙과 동일한 시간 복잡도인 $O(1)$ 삽입 전이 시간과 $O(\log n)$ 삭제 시간 복잡도를 가지며, 단순한 전통적인 힙에 근거하고 있기 때문에 [5]에서 제안된 힙보다 구현이 매우 용이하다.

**키워드 : 자료 구조, 우선순위 큐, 힙, 전이 시간 복잡도**

# Insertion/Deletion algorithms on M-heap with an array representation

Haejae Jung†

## ABSTRACT

Priority queues can be used in applications such as scheduling, sorting, and shortest path network problem. Fibonacci heap, pairing heap, and M-heap are priority queues based on pointers.

This paper proposes a modified M-heap with an array representation, called MA-heap, that resolves the problem mentioned in [1]. The MA-heap takes $O(1)$ amortized time and $O(\log n)$ time to insert an element and delete the max/min element, respectively. These time complexities are the same as those of the M-heap. In addition, it is much easier to implement an MA-heap than a heap proposed in [5] since it is based on the simple traditional heap.

**Key Words : Data Structure, Priority Queue, Heap, Amortized Time Complexity**

## 1. 서 론

우선 순위 큐(priority queue : PQ)는 운영 체제 스케줄링, 사건 시뮬레이션, 또는 정렬과 같은 응용에 사용되는 자료 구조로서, 어떤 데이터 집합에서 우선순위가 가장 높거나 낮은 것을 빨리 찾는 자료 구조이다. 전자를 최대 우선순위 큐라 하고 후자를 최소 우선순위 큐라 하는데, 본 논문에서는 별다른 언급이 없는 한 최대 우선순위 큐를 우선순위 큐라 한다. 최대 우선순위 큐는 어떤 집합 S에 대해 다음의 삽입 및 삭제 연산을 기본적으로 지원한다.

- insert(e, S) : 집합 S에 새로운 임의의 데이터 e를 추가.
- delmax(S) : 집합 S로부터 우선순위가 가장 높은 데이터를 삭제.

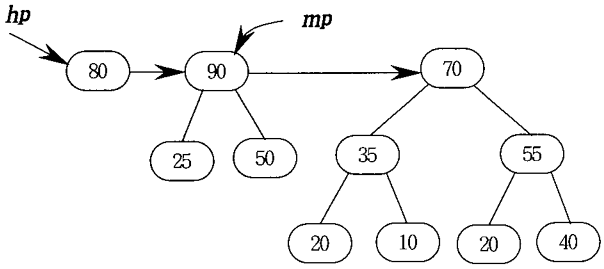
※ 이 논문은 2005학년도 안동대학교 학술연구 조성비에 의해 연구되었음.
† 종신회원 : 안동대학교 공과대학 정보통신공학과 교수
논문접수 : 2006년 3월 14일, 심사완료 : 2006년 5월 8일

우선순위 큐를 구현하기 위해 배열을 이용하는 자료구조 표현은 필요한 메모리를 동적으로 할당 및 회수하는 대신 배열을 사용하기 때문에, 필요한 데이터 최대 수를 알고 있는 응용에 이용된다. 배열을 이용하면, 각 데이터에 대응하는 포인터 공간을 절약할 수 있다. 포인터를 이용한 우선순위 큐에는 $O(1)$ 삽입 및 $O(\log n)$ 삭제 전이 시간 복잡도를 가지는 피보나치 힙, $O(1)$ 삽입 전이 시간 및 $O(\log n)$ 삭제 시간 복잡도를 M-힙, 그리고 $O(\log n)$ 삽입 및 삭제 전이 시간 복잡도를 가지는 페어링 힙 등이 있다. 페어링 힙은 피보나치 힙에 비해 시간 복잡도에 있어서는 나쁘지만, 실질적인 성능에 있어서는 우수한 것으로 나타나 있다[4]. 실질적인 성능 개선을 위해 제안된 [8]과 [9]의 자료 구조는 전통적인 힙을 수정하여 캐쉬 메모리를 효과적으로 이용하도록 하고 있다.

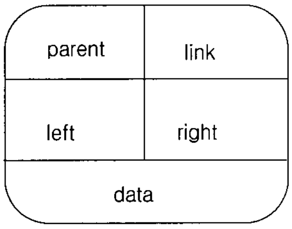
피보나치 힙의 삽입 연산은 새로 삽입될 노드를 존재하는 연결 리스트에 단순히 추가한다[2, 6, 7]. 삭제시에는 최대 키 값을 가지고 있는 노드를 삭제하고, 연결 리스트에 있는 나머지 힙들은 동일한 등급(degree)을 가진 힙끼리 반복하

여 결합한다. 페어링 힙에서의 삽입은 새로 삽입될 노드의 키와 이미 존재하는 힙의 루트 노드의 키 값을 비교하여, 작은 키를 가진 힙을 큰 키를 가진 노드의 자식 노드로 연결한다[3, 4, 7]. 삭제시에는 루트 노드를 삭제하고, 모든 자식 노드를 결합하여 하나의 힙으로 만든다.

본 논문의 주 참조 자료 구조인 M-힙은 최대 $O(\log n)$ 개의 포화(full) 트리를 트리 높이에 대해 오름차순으로 연결한 힙인데, 본 논문에서는 각 포화 트리를 내부힙이라 부른다[1]. (그림 1)과 (그림 2)는 3개의 내부 힙으로 구성된 M-힙의 예와 M-힙의 노드 구조를 각각 보여주고 있다.




(그림 1) M-힙의 예



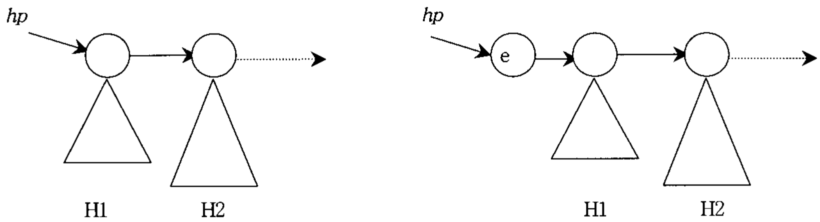

(그림 2) M-힙의 노드 구조

(그림 1)의 헤드 포인터 hp는 가장 낮은 높이를 가진 내부힙을 가리키고, 최대 포인터 mp는 가장 큰 키를 가진 내부힙의 루트 노드를 가리킨다. M-힙에서는 첫 두 내부힙을 제외한 모든 내부힙의 높이는 다르다. (그림 2)의 link 필드는 내부힙의 루트 노드들을 서로 연결하기 위해 사용되고, 루트 노드 이외의 모든 노드에서는 null로 되어 사용되지 않는다. M-힙에서 hp가 가리키는 내부힙을 첫 번째 내부힙이라 하고, 첫 번째 내부힙의 루트 노드의 link 필드에 의해 연결된 힙을 두 번째 내부힙이라한다. (그림 1)에서 첫 번째 내부힙의 루트 노드는 데이터 80을, 두 번째 내부힙 루트는 90을 가지고 있다. parent, left, 및 right 필드는 각각 부모, 왼쪽, 및 오른쪽 자식 노드를 가리키며, 데이터는 data 필드에 저장된다.

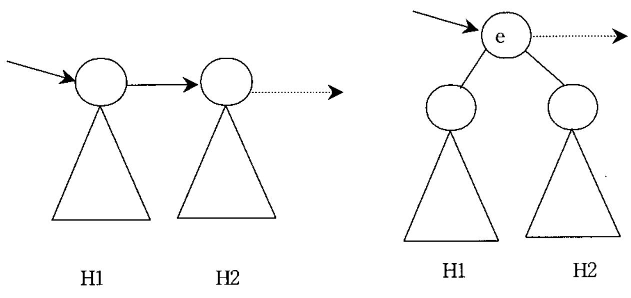
M-힙에서의 삽입은 가장 낮은 높이를 가진 첫 두 내부힙이 동일한 높이를 가진 경우, 그 두 내부힙의 루트 노드를 새로운 노드의 자식으로 만들어 결합한다. 그렇지 않으면, 새로운 노드는 단순히 첫 번째 내부힙으로서 추가된다.

(그림 3)은 hp에 연결된 첫 두 내부힙 H1과 H2의 높이가 서로 다른 경우, 데이터 e가 M-힙의 첫 번째 내부힙으로 삽입된 것을 보이고, (그림 4)는 첫 두 내부힙의 높이가 같은 경우 데이터 e가 첫 두 내부힙의 루트 노드로 삽입되는 것을 보이고 있다.

M-힙에서의 최대 키 삭제는 mp가 가리키는 노드로부터 가장 큰 키 값을 갖는 데이터를 제거하고, 그 자리에 첫 번째 내부힙의 루트에 있는 데이터를 삽입한 후, mp가 가리키는 내부힙에 대해 힙조정(heapify)을 수행하여 힙 순서를 유지한다. 첫 번째 내부힙이 자식을 가지고 있었다면, 왼쪽 및 오른쪽 자식은 각각 첫 번째 및 두 번째 내부힙이 되고, 그렇지 않으면 hp는 두 번째 내부힙의 루트 노드를 가리키게




(그림 3) 첫 두 내부힙 H1과 H2의 높이가 다른 경우의 삽입




(그림 4) 첫 두 내부힙 H1과 H2의 높이가 같은 경우의 삽입

된다. 그 후, hp에 의해 지정되는 첫 번째 노드로부터 link 필드를 따라 가면서 최대 키를 가지고 있는 노드를 찾아, 그 노드를 mp로 하여금 가리키게 한다. 또한, M-힙에서는 미리 지정된 노드의 데이터를 삭제하는 임의의 노드 삭제도 지원된다. 즉, 지정된 노드의 데이터를 삭제한 후, 그 자리에 첫 번째 내부힙의 루트 노드 데이터를 삽입하여 힙조정을 한다. 힙조정은 삽입된 데이터 키가 삭제된 키보다 크면 위쪽으로 작으면 아래쪽으로 이동함으로써 이루어진다.

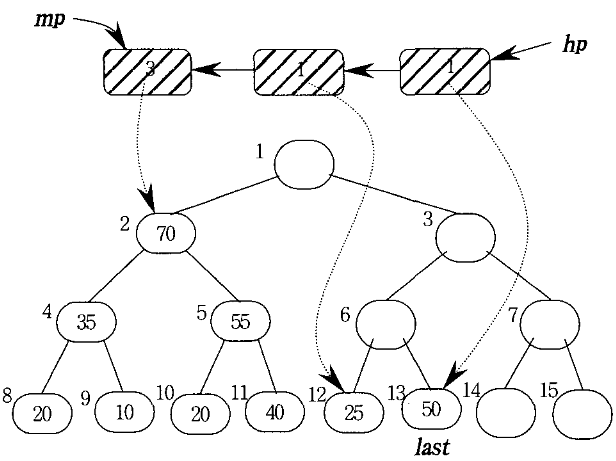
[1]에서는 (그림 2)에 나타난 바와 같이 M-힙의 각 노드마다 4개의 포인터 필드를 포함하고 있기 때문에 배열을 이용한 전통적인 묵시힙에 비하여 메모리 사용에 있어서 비효율적임을 결점으로 지적하고 있다.

본 논문에서는 [1]에서 미해결 문제로 남겨 두었던 배열을 이용하여 M-힙을 표현한 MA-힙(M-heap with an array representation)을 제안한다. 제안된 MA-힙은 M-힙과 동일한 시간 복잡도인 $O(1)$ 삽입 전이 시간과 $O(\log n)$ 삭제 시간 복잡도를 가진다. 또한, MA-힙에서의 삽입 및 삭제 연산시 부모/자식 노드를 찾기 위한 노드 인덱스 계산식은 전통적인 묵시힙(implicit heap)의 단순한 수식을 그대로 사용하므로 구현이 매우 용이하다. [5]의 자료 구조 또한 배열을 이용하고 MA-힙과 동일한 시간 복잡도를 가지지만, MA-힙에 비해 알고리즘이 상당히 복잡하다. MA-힙에서도 M-힙에서 지원하는 임의의 노드 삭제 연산을 M-힙에서와 동일한 $O(\log n)$ 시간에 지원하지만, 그 알고리즘은 최대키 삭제 알고리즘으로부터 쉽게 유추할 수 있으므로 본 논문에서는 더 이상 언급하지 않기로 한다. 다음 절에서는 제안된 MA-힙 자료 구조에 대해 설명하고, 3절에서 MA-힙의 삽입 및 최대 키 삭제 알고리즘을 기술한 후, 4절에서 결론을 맺도록 한다.

## 2. MA-힙 자료 구조

MA-힙에서는 포화 이진 트리를 형성하도록 메모리 공간을 할당하여 데이터의 삽입 및 삭제가 이루어진다. 유효 데이터를 가지는 각 내부힙의 루트 노드는 헤드 포인터 hp로부터 연결된 리스트 노드((그림 5)에서 사선으로 채워진 노드)를 통해 접근되고, 최대 키 값을 가진 루트 노드는 최대 포인터 mp를 통해 접근된다. 내부힙 역시 포화 이진 트리로 유지되고, 내부힙들은 hp로부터 힙 높이에 대하여 오름 차순으로 정렬된 상태로 유지된다. 내부힙들의 높이는 hp에 제일 가까운 두 힙을 제외하고는 모두 다르며, 내부힙의 갯수는 많아야 $O(\log n)$이다.

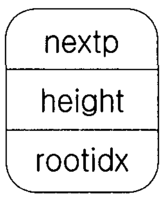
(그림 5)의 MA-힙은 3개의 내부힙으로 구성되어 있고, 각각의 루트 노드는 리스트 노드에 의해 참조된다. 내부힙의 높이는 hp에 가장 가까운 리스트 노드에 연결된 것, 즉 첫 번째 내부힙이 가장 낮고, 멀어질수록 높아진다. 또한, 첫 번째와 두 번째, 즉 hp에 가장 가까운 두 리스트 노드에 연결된 내부힙만이 동일한 높이를 가질 수 있다. (그림 5)의 경우 인덱스 13을 가진 노드가 첫 번째 내부힙이고 인덱스



(그림 5) MA-힙 구조

12인 노드가 두 번째 내부힙인데, 이들 두 내부힙의 높이가 1로서 동일하다. 최대 키 값은 내부힙들의 루트 노드 중 하나가 가지게 되는데, 그 노드를 상수 시간에 접근하기 위해 최대 포인터 mp를 유지한다. mp는 실제로 리스트 노드를 가리키고, 그 리스트 노드를 통하여 최대 키를 가진 내부힙의 루트 노드가 접근된다. (그림 5)에서 mp는 제일 마지막 리스트 노드를 가리키고, 그 리스트 노드를 통하여 접근되는 노드가 최대 키 값인 70을 가지고 있다. 변수 last는 리프 노드 인덱스 값을 가지는데, 첫 번째 내부힙의 가장 오른쪽 리프 노드 인덱스 즉, 데이터를 가지고 있는 리프 노드 중 가장 오른쪽 노드의 인덱스 값을 가진다. 삽입시 첫 두 내부힙의 높이가 다른 경우, last 값이 증가되어 상수 시간에 삽입 위치가 결정된다. (그림 5)에서 last 값은 13인데, last는 리프 노드를 나타내는 8에서 15 사이의 인덱스 중 하나를 가질 수 있다.

(그림 6)는 리스트 노드의 구조를 보여주고 있다. 리스트 노드에서 nextp는 그 다음 리스트 노드를 가리킨다. 리스트 노드의 rootidx 필드에는 내부힙의 루트 노드 인덱스가 저장되고, height는 rootidx에 의해 접근되는 내부힙의 높이를 나타낸다. (그림 5)에서 nextp와 rootidx는 각각 실선과 점선으로 나타나 있고, height는 리스트 노드 내에 정수로 표기되어 있다.



(그림 6) MA-힙의 리스트 노드 구조

## 3. MA-힙 연산

MA-힙에서는 $n$개의 데이터 저장을 위해 크기가 $M = 2^{\lfloor \log n \rfloor + 1}$

인 배열 A[$M$]를 할당하여, 포화 이진 트리가 되도록 한다. 포인터 hp와 mp는 NULL로 초기화되고, 변수 last는 MA-힙의 제일 왼쪽 리프 노드 인덱스보다 1이 적은 $L = 2^{\lfloor log n \rfloor}$ -1로 초기화된다. 데이터 삽입은 가장 왼쪽 리프 노드에서 시작하여 오른 쪽으로 가면서 이루어지는데, 가장 오른쪽 데이터를 가지는 리프 노드 위치는 변수 last에 의해 유지된다. MA-힙에는 여러 개의 내부힙이 존재할 수 있는데, 각 내부힙 높이 또는 크기는 이웃하는 두 내부힙 합병을 통하여 항상 상향식으로 커지고, 루트 노드 제거에 의한 분할에 의해 하향식으로 적어진다. 알고리즘 설명의 편의를 위하여 데이터는 정수형 키 필드로만 구성되어 있다고 가정한다.

삽입과 삭제 알고리즘에서 사용되는 (알고리즘 1)의 heapify() 함수는 루트 노드의 인덱스가 k인 내부힙 루트 노드에 새로운 데이터가 삽입될 때, 힙조정을 하여 그 내부힙을 힙 순서로 유지되도록 하는 함수이다. 힙 조정은 하향식, 즉 루트 노드로부터 리프 노드로 가면서 이루어진다. A[k]에 있는 데이터가 자식 노드들 중의 큰 데이터와 비교하여, 작으면 데이터를 교환한다. 이러한 과정은 A[k]의 값이 자식 노드 데이터보다 크게 될 때까지 또는 A[k]가 리프 노드가 될 때까지 반복된다.

```
void heapify( A[], k )
{
   left = 2*k;  // left child of k
   while( left < M ) {  // 자식이 있는 동안
      max = left;
      if( A[max] < A[left+1] ) max = left+1; // right child of k.
      if( A[k] > A[max] ) break;
      swap( A[k], A[max] );
      k = max;  left = 2*k;
   }
}
```

(알고리즘 1) 힙조정 알고리즘

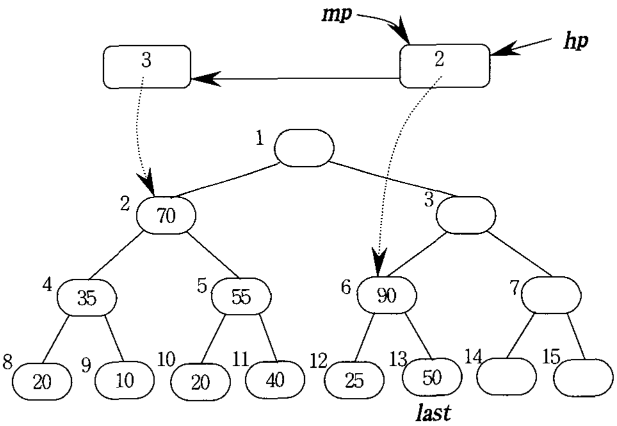
(알고리즘 2)는 MA-힙의 삽입 함수를 보여주고 있는 데, 삽입은 항상 hp에 가장 가까운 내부힙 쪽에서 이루어진다. 우선 MA-힙이 완전히 포화되어 있을 경우 즉, hp.rootidx가 1인 경우, FALSE 값을 반환한다. 그렇지 않고 데이터를 삽입할 공간이 있으면, 먼저 hp에 연결되어 있는 첫 두 내부힙의 높이를 검사한다. 높이가 같으면, 그 두 힙의 루트 노드의 부모 노드 paridx에 입력 데이터 thedata를 삽입한 후, heapify() 함수를 호출하여 힙조정을 한다. 이렇게 합쳐진 내부힙은 첫 번째 리스트 노드 hp로 하여금 가리키도록 하고, 두 번째 리스트 노드였던 sp는 삭제한다. 높이가 같지 않거나 힙이 비어 있거나 또는 힙에 하나의 내부힙 만이 있는 경우, thedata는 1 증가된 last가 나타내는 노드에 삽입되고, 그에 대응하는 새로운 리스트 노드를 생성하여 hp 리스트의 첫 번째 노드로 삽입한다.

예를 들어, (그림 5)의 MA-힙에 insert(90) 및 insert(60)을 순서대로 실행한 결과는 (그림 7) 및 (그림 8)과 같다. 90 삽입 시, 첫 두 내부힙의 높이가 같으므로 90을 그 두 힙

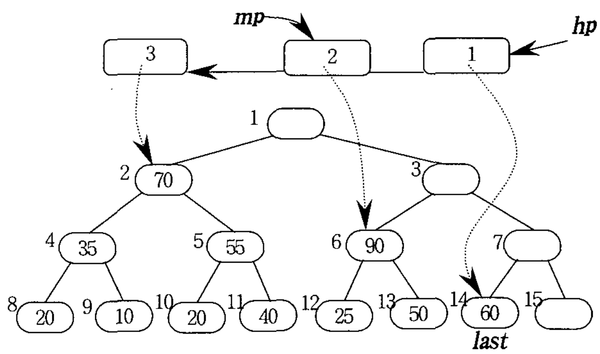
```
bool insert( Element thedata )
{  h1 = -1;  h2 = -2;
   if( hp ) {
      if( hp.rootidx == 1 ) return FALSE;  // MA-힙 is full.
      h1 = hp.height;      // 첫 번째 내부힙 높이
      sp = hp.nextp;       // 두 번째 리스트 노드
      if( sp ) h2 = sp.height; // 두 번째 내부힙 높이
   }
   if( h1 == h2 ) {
      paridx = hp.rootidx / 2;
      A[paridx] = thedata;  heapify( A[], paridx );
      hp.rootidx = paridx;  hp.height++;
      hp.nextp = sp.nextp;
      delete sp;  // 두 번째 리스트 노드 삭제
   } else {
      last++;  A[last] = thedata;
      fp = new ListNode;
      fp.rootidx = last;  fp.height = 1;
      fp.nextp = hp; hp = fp;
   }
   if( mp == NULL ) mp = hp;
   if( thedata > A[mp.rootidx] )  mp = hp;
   return TRUE;
}
```

(알고리즘 2) MA-힙 삽입 알고리즘

의 부모 노드에 삽입하여 병합한다. 60 삽입 시, 첫 두 내부힙의 높이가 다르므로 새로운 내부힙이 생성되는데, 이를 위해 last가 증가되고, 그 자리에 60이 삽입된다.




(그림 7) (그림 5)에 insert(90) 실행 후의 MA-힙




(그림 8) (그림 7)에 insert(60) 실행 후의 MA-힙

최대 키를 가진 데이터의 삭제 알고리즘은 (알고리즘 3)에 나타나 있다. 최대 데이터는 mp에 의해 지정되는 내부힙의 루트 노드로부터 삭제되어 tmpdata에 임시 저장되고, 그 자리에는 첫 번째 내부힙의 루트 노드에 있는 데이터가 삽입되어 heapify() 함수를 통하여 힙조정이 이루어진다. 첫 번째 내부힙이 자식 노드를 가지고 있으면, 그 자식 노드들을 위한 리스트 노드가 추가된다. 이때, 오른쪽 자식 노드가 hp 리스트의 첫 번째 내부힙의 루트가 되고, 왼쪽 자식 노드가 두 번째 내부힙의 루트 노드가 된다. 자식 노드가 없을 경우, 변수 last를 감소시키고 첫 번째 리스트 노드를 삭제한다. 그 후, hp에 연결된 리스트 노드를 탐색하여 최대 키를 가진 내부힙 루트 노드를 찾아 mp를 조정하고, 임시 저장된 최대 데이터 tmpdata를 반환한다. (그림 7)에서 delmax()를 실행하면, (그림 5)의 MA-힙이 된다.

$$T = O(2^h \sum_{i=1}^{h} (i/2^i)) = O(2^h \cdot 2) = O(n)$$

```
int delmax( )
{ if( last <= L ) return -1;  // MA-힙 is empty.
  maxidx = mp.rootidx;
  firstidx = hp.rootidx;
  tmpdata = A[maxidx];
  if( maxidx != firstidx ) {
    A[maxidx] = A[firstidx]; heapify( A[], maxidx );
  }
  left = 2*firstidx;
  if( left < M ) {  // 첫 번째 내부힙 루트에 자식이 있음.
    sp = new ListNode;
    sp.nextp = hp.nextp;  hp.nextp = sp;
    hp.rootidx = left+1; hp.height--;
    sp.rootidx = left;    sp.height = hp.height;
  } else {
    last--;
    tmp = hp;  hp = hp.nextp;
    delete tmp;
  }
  mp = hp;  // mp 조정.
  for( tmp = hp.nextp; tmp != NULL; tmp = tmp->nextp ) {
    if( A[tmp.rootidx] > A[mp.rootidx] ) mp = tmp;
  }
  return tmpdata;
}
```

(알고리즘 3) MA-힙 삭제 알고리즘

**(정리 1)** $n$ 개의 데이터를 가지고 있는 MA-힙에서, 삽입과 삭제 연산은 각각 $O(1)$ 전이 시간 복잡도와 $O(\log n)$ 시간 복잡도가 걸린다.

**(증명)** 리스트 노드의 삽입 및 삭제와 변수 last의 증가 및 감소는 상수 시간 걸린다.

1) 삭제 연산의 경우, 힙조정(heapify)은 내부힙의 높이인 $O(\log n)$ 시간이 걸리고, mp를 재조정하기 위해 hp에 연결된 $O(\log n)$개의 리스트 노드를 따라가면서 최대 키를 가진 내부힙의 루트 노드를 찾는다. 따라서, 삭제는 $O(\log n)$ 시간이 걸린다.

2) 삽입 전이 시간 복잡도를 구하기 위해, 일련의 삽입 연산이 이루어진다고 하자. 일련의 삽입 중, 두 개의 내부힙으로 구성된 MA-힙에 새로운 데이터를 삽입할 경우 하나의 내부힙만으로 구성된 MA-힙이 되는데, 이때 최대 비용이 소요된다. 삽입 시의 힙조정은 항상 이웃하는 두 내부힙의 부모 노드에서 시작하여 리프 노드 쪽으로 이루어지므로, 그 부모 노드의 높이 만큼의 시간이 걸린다. 따라서, 하나의 내부힙으로 구성된 MA-힙의 높이가 h라 할 때, 일련의 삽입 총 비용을 계산하면 $T = \sum_{i=1}^{h} O(i) 2^{h-i}$가 된다. 즉, 트리 레벨 $i$ (리프 노드 레벨을 1로 가정)에는 $2^{h-i}$개의 노드가 있고, 각 노드에 대한 삽입 비용은 $O(i)$가 된다. 따라서 총 비용을 계산하면,

$$T = O(2^h \sum_{i=1}^{h} (i/2^i)) = O(2^h \cdot 2) = O(n)$$

이 되므로, 각 데이터 삽입에 대한 전이 시간 복잡도는 $O(n)/n = O(1)$이 된다.

## 4. 결 론

배열을 이용한 힙은 각 데이터에 포인터와 같은 부가적인 필드를 필요로 하지 않으므로 효율적으로 메모리를 사용할 수 있으며, 배열을 이용하므로 미리 데이터의 최대 개수가 예측되는 응용에 유용하게 이용될 수 있다.

본 논문에서 제안하는 MA-힙은 [1]에서 문제점으로 남겼던 배열을 이용한 M-힙으로서, $n$ 개의 데이터에 대한 삽입과 삭제 연산에 각각 O(1) 전이 시간 복잡도와 $O(\log n)$ 시간 복잡도를 가진다. 또한, MA-힙은 단순한 전통적인 힙에 근거를 두고 있어, 그 구현이 [5]에 비해 매우 용이하다.

## 참 고 문 헌


[1] S. Bansal, S. Sreekanth, and P. Gupta, "M-heap : A Modified heap data structures", International Journal of Foundations of Computer Science, 14(3), pp.491-502, 2003.

[2] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," JACM, 34(3), pp.596-615, 1987.

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

[4] T.J. Stasko and J.S. Vitter, "Pairing heaps : experiments and analysis", Communications of the ACM, 30(3), pp.234-249, 1987.

[5] 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, 318, pp.1-13, July, 1988.
[6] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of Data Structures in C++, W. H. Freeman, San Francisco, 1995.
[7] D. Mehta, and S. Sahni(ed.), Handbook of Data Structures and Applications, Chapman & Hall/CRC, New York, 2005.
[8] A. LaMarca, and R. Ladner, "The Influence of Caches on the Performance of Heaps," ACM Jounrnal of Experimental Algorithmics, 1(4), pp.1-32, 1996.
[9] H. Jung, "The d-deap* : A fast and simple cache-aligned d-ary deap", Information Processing Letters, 93(2), pp.63-67, Jan., 2005.

정 해 재
e-mail : hjjung@andong.ac.kr
1984년 경북대학교 전자계산학과(공학사)
1987년 서울대학교 컴퓨터공학과 (공학석사), DBMS
2000년 플로리다대학교(UF) 컴퓨터정보학과 (공학박사), 알고리즘
1988년~1995년 한국전자통신연구원 선임연구원
2001년~2002년 Numerical Technologies Inc. Staff Engineer.
2003년~2005년 성신여자대학교 컴퓨터정보학부 교수
2005년~현재 안동대학교 공과대학 정보통신공학과 교수
관심분야 : 고성능 정보처리, 자료구조 및 알고리즘, 계산기하, 데이터베이스, 네트워크 알고리즘