KIPS Trans. Comp. and Comm. Sys.
Vol.8, No.2 pp.29~34 pISSN: 2287-5891
https://doi.org/10.3745/KTCCS.2019.8.2.29

IMI-Heap: An Implicit Double-Ended Priority Queue with Constant Insertion Amortized Time Complexity

Haejae Jung†

ABSTRACT

Priority queues, one of the fundamental data structures, have been studied for a long time by computer scientists. This paper proposes an implicit double-ended priority queue, called IMI-heap, in which insert operation takes constant amortized time and each of removal operation of the minimum key or the maximum key takes O(logn) time. To the author's knowledge, all implicit double-ended priority queues that have been published, perform insert, removeMin and removeMax operations in O(logn) time each. So, the proposed IMI-heap is superior than the published heaps in terms of insertion time complexity.The abstract should concisely state what was done, how it was done, principal results, and their significance.

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

IMI-힙: 상수 삽입 전이 시간 복잡도를 가진 묵시 양단 우선순위 큐

정 해 재†

요 약

우선순위 큐은 근본적인 자료 구조 중의 하나이며 오랫동안 많은 연구가 이루어여 왔다. 본 논문에서는 IMI-힙이라고 하는 묵시 양단 우선순위 큐를 제안한다. 제안된 IMI-힙에서는 삽입에 O(1) 전이시간이 걸리고 최소값과 최대값 삭제 연산에 각각 O(logn) 시간이 걸린다. 기존에 발표된 묵시 양단 우선순위 큐는 삽입과 최소/최대값 삭제에 모두 O(logn) 시간이 걸리는 것으로 본 저자는 알고 있다. 따라서 제안된 IMI-힙은 삽입 시간 복잡도에 있어서 기존의 힙보다 우수하다.

키워드 : 양단 우선순위 큐, 묵시 힙, 전이 시간 복잡도, 자료 구조

1. 서 론

우선순위 큐(priority queue)는 기본적인 자료구조 중의 하나로서 많은 연구가 이루어져 왔다. 우선순위 큐에는 일단 우선순위 큐(single-ended priority queue)와 양단 우선순위 큐(DEPQ: double-ended priority queue)로 나눌 수 있고, 그 구현은 트리 구조에서 부모-자식 관계를 나타내기 위해 포인터를 이용하는 방법과 포인터를 사용하지 않고 배열의 인덱스만을 이용하는 묵시적인 방법이 있다.

※ 이 논문은 2016학년도 안동대학교 연구비에 의하여 연구되었음.
† 종신회원 : 안동대학교 정보통신공학과 교수
Manuscript Received : August 13, 2018
Accepted : December 18, 2018
* Corresponding Author : Haejae Jung(hjjung@anu.ac.kr)

묵시(implicit) 일단 우선순위 큐로는 O(logn) 삽입 및 삭제 시간을 가지는 전통적인 힙(이하 전통힙이라 함)이 잘 알려져 있다. 상수 삽입 전이시간(amortized time)과 O(logn) 삭제 시간 복잡도를 가지는 묵시 일단 우선순위 큐로는 묵시 이항 큐(implicit binomial queue), 후위힙(postorder heap), 및 AM-힙이 있다[1-4]. 후위힙은 트리 노드에 대한 인덱스 순서가 후위순이고 AM-힙은 전통힙과 같이 레벨순으로 이루어지며, 키 삽입은 두 힙 모두에서 후위순으로 이루어진다.

묵시 양단 우선순위 큐에는 최대-최소 힙(min-max heap), 딥(deap), d-딥*(d-deap*), 대칭 최대-최소 힙(symmetric min-max heap), 구간 힙(interval heap) 등이 있으며, 이들은 모두 O(logn) 삽입 및 삭제 시간 복잡도를 가진다[1, 5-7].

양단 우선순위 큐는 외부 정렬(external sorting)과 같은 응

※ 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.

용에 사용될 수 있다[8]. 즉 내부 퀵정렬(quick sort)과 같은 방법으로, DEPQ의 키들을 중간 그룹(pivot)으로 하고, 중간 그룹의 제일 작은 키보다 더 작은 키들을 왼쪽 그룹으로 중간 그룹의 제일 큰 키보다 더 큰 키를 오른쪽 그룹으로 배정하는 것을 반복함으로써 디스크에 있는 모든 키를 정렬할 수 있다.

본 논문에서는 O(1) 삽입 전이 시간과 O(logn) 최대값/최소값 삭제 시간을 가지는 묵시 양단 우선순위 큐인 IMI-힙(Implicit M-heap and Interval-heap)을 제안하며, 제안된 자료 구조는 지금까지의 O(logn) 삽입과 O(logn) 최대값/최소값 삭제 시간 복잡도를 가지는 묵시 양단 우선순위 큐에 비해 삽입 시간 복잡도에 있어서 우수하다.

2. 관련 연구

일단 우선순위 큐 중 상수 삽입 전이시간과 Ologn) 삭제 시간이 걸리는 자료구조에 대해 먼저 살펴본다. M-힙은 전통힙의 구조를 수정하여 O(logn)개의 성분힙을 가지는 자료 구조로서 상수 삽입 전이 시간과 O(logn) 삭제 시간이 걸리며, 포인터를 사용하는 자료구조이다[9]. 후위 힙은 포화 이진 트리(모든 리프 노드가 동일한 레벨에 있는 트리) 구조를 가지며, 전통힙과는 달리 배열에 대응하는 트리 노드 인덱스가 후위 순서로 매겨지며, 삽입 역시 이 순서로 이루어진다[3]. M-힙을 묵시힙으로 만든 AM-힙은 완전 이진 트리(complete binary tree) 구조를 가지며, 트리 인덱스는 전통힙과 같이 레벨 순서로 매겨지고 삽입은 후위 순서로 이루어진다[4]. 후위힙과 AM-힙에서의 실질적인 삭제는 후위순서의 반대로 이루어진다.

묵시 양단 우선순위 큐 중 구간 힙은 전통힙과 같은 완전 이진 트리 구조를 가지지만, 각 노드에는 구간의 왼쪽값과 오른값을 나타내는 두 개의 값 <lKey, rKey>을 유지하고, lKey <= rKey의 관계를 가진다[6, 7]. 또한, 구간 힙은 부모 노드의 구간이 자식 노드의 구간을 포함하는 특성을 가지며, 삽입과 삭제에 모두 O(logn) 시간이 걸린다. 구간힙을 M-힙 구조로 수정한 MI-힙은 상수 삽입 전이 시간과 O(logn) 최소키/최대키 삭제 시간을 가지는 자료 구조이지만 완전한 묵시힙은 아니다[10]. 왜냐하면, 키를 저장하기 위해서 배열을 사용하고 있지만 모든 성분힙의 루트를 서로 연결하기 위해 O(logn)개의 노드를 포인터로 연결하는 연결 리스트를 사용하고 있기 때문이다.

본 논문에서 제안하는 IMI-힙은 MI-힙에서 사용한 포인터를 모두 없애고 순수하게 배열 인덱스만을 이용하여 모든 연산을 지원하는 묵시 양단 우선순위 큐이며, 삽입과 삭제 연산의 시간 복잡도는 MI-힙과 동일하다.

3. IMI-힙 구조

IMI-힙은 완전 이진 트리로 표현되며, AM-힙이나 MI-힙과 같이 최대 O(logn) 개의 성분힙(component heap)으로 구성된다[4, 10]. Fig. 1은 노드 2와 노드 6에 루트를 가진 두 개의 성분힙으로 구성된 IMI-힙의 예를 나타낸다. IMI-힙에서는 삽입과 삭제를 하기 위해서 두 개의 변수 lastLeaf와 lastRoot를 유지한다. lastLeaf는 키를 가진 노드중 가장 오른쪽 리프 노드를 나타내고, lastRoot는 마지막 성분힙 즉 가장 오른쪽 성분힙의 루트 노드의 인덱스 값을 가진다.

IMI-힙에서는 각 노드에 좌단키(lKey)와 우단키(rKey)를 나타내는 두 개의 키값<lKey, rKey>을 저장하여 구간을 나타내는데, 그 두 값의 관계는 lKey <= rKey 이다. 예를 들면, Fig. 1에서 노드 4와 5의 구간은 노드 2의 구간에 포함된다. IMI-힙에 홀수개의 키가 있을 경우, lastRoot 노드의 rKey 값은 실제 저장될 수 있는 어떠한 키 값보다 큰 값 INF을 가지며, Fig. 1의 노드 6에 ∞로 표시되어 있다. IMI-힙의 특성은 다음과 같다.

1. IMI-힙은 완전 이진 트리(complete binary tree)이다.
2. IMI-힙은 최대 O(logn)개의 성분힙을 가지며, n은 힙에 있는 노드의 개수이다. 제일 오른쪽 즉, 마지막 두 개의 힙을 제외하고는 모든 성분힙의 높이가 서로 다르다.
3. 각 노드는 <lKey, rKey>인 두 개의 키 값을 가지며, 구간을 나타낸다. 즉, lKey <= rKey.
4. 각 노드에 대해, 부모 노드의 구간은 자식 노드의 구간을 포함한다.

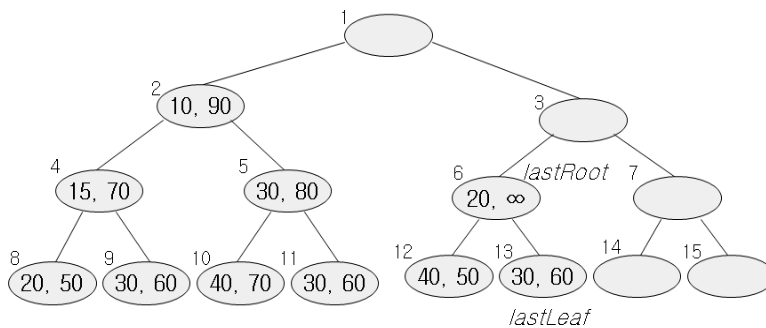

Fig. 1. An IMI-heap with Two Component Heaps

특성 2에서 lastRoot가 홀수이면 마지막 두 성분힙은 동일한 높이를 가진다. 위의 특성 3과 4에 의하여 각 성분힙의 모든 lKey는 최소힙을 형성하고, 모든 rKey는 최대힙을 형성한다.

4. IMI-힙 연산

4.1 초기화

IMI-힙에서는 maxN 개의 키를 저장하기에 충분한 크기의 배열 arr[ ]를 할당한다. 힙이 공백이라는 것을 표현하기 위해 마지막(가장 오른쪽) 성분힙의 루트를 나타내는 변수 lastRoot를 0으로 초기화하고, 변수 lastLeaf는 IMI-힙의 제일 왼쪽 리프 노드 인덱스 startLeaf보다 1 적은 값으로 초기화한다.

```
void initialize(maxN) // maxN: max. number of keys
{
   size = ⌈maxN/2⌉;
   maxIndex = size;
   if( size % 2 == 0 )
      maxIndex++;  // every node has 2 children

   // the root has index 1.
   arr = new keyType[maxIndex+1];
   startLeaf = 2^⌊log2(maxIndex)⌋; // leftmost leaf node

   // root of last(rightmost) component heap
   lastRoot = 0;
   lastLeaf = startLeaf - 1;
}
```

Fig. 2. Initialization of IMI-heap

4.2 힙조정

임의의 키가 노드 r의 lKey 또는 rKey로 삽입될 경우, 힙 조정은 노드 r로부터 하향식으로 이루어지는데, 그 알고리즘은 각각 전통힙의 최소힙 또는 최대힙에서의 힙조정과 유사하다.

키가 rKey로 삽입될 경우, rKey로만 형성되는 우단힙을 따라 하향식으로 힙조정이 이루어지는데, 그 알고리즘은 Fig. 3에 나타나 있다. 함수 heapifyRight( )의 인수 r은 새로운 rKey 값을 가지고 있는 노드 인덱스를 나타낸다. 함수 cmpSwap( )는 두 인수를 비교하여 첫 번째 키가 두 번째 키보다 크면 서로 교환해 주는 함수이다. while 루프에서 노드 r의 두 자식 노드의 rKey 중 큰 값을 가지는 노드를 c로 두고, 노드 c의 rKey가 부모 노드 r의 rKey보다 크면 두 rKey를 서로 교환한다. 이러한 교환은 부모 노드의 rKey가 두 자식노드의 rKey보다 크게 될 때 종료된다(라인 8).

임의의 키가 노드 r의 lKey로 삽입되는 경우, 각 노드의 lKey로만 구성되는 좌단힙을 따라 하향식으로 힙조정을 하는 heapifyLeft( )도 이와 유사하게 작성할 수 있다.

```
1  // parameter r : node to start to heapify
2  void heapifyRight( r )
3  {
4     cmpSwap( arr[r].lKey, arr[r].rKey );
5     c= 2*r;  // c: left child of r
6     while( c < maxIndex ) {  // while node c is valid
7        if( arr[c].rKey < arr[c+1].rKey ) c++; // c: right child of r.
8        if( arr[r].rKey > arr[c].rKey ) break;
9        swap( arr[r].rKey, arr[c].rKey );
10       cmpSwap( arr[c].lKey, arr[c].rKey );
11       r = c;  c = 2*r;
12    }
13 }
```

Fig. 3. Heapify Algorithm of Right-end

4.3 삽입

Fig. 4는 IMI-힙 삽입 알고리즘을 보여주고 있다. 인수로 전달된 theKey를 삽입하기 위하여 마지막 성분힙의 루트 노드 lastRoot가 한 개의 데이터만 가지고 있는지 알아본다. 이 경우 인수로 전달된 theKey로 rKey인 INF(∞)값을 대체하고 heapifyRight( ) 함수를 호출하여 힙조정을 한다(라인4~8).

마지막 성분힙의 루트 인덱스 lastRoot가 짝수인 경우(라인 12), lastLeaf를 1 증가시켜 새로운 성분힙을 생성하고, 이 노드의 lKey와 rKey 필드에 theKey와 INF 값을 각각 삽입한다. lastRoot는 lastLeaf 노드와 동일한 노드를 가리킨다.

마지막 두 성분힙의 루트가 같은 수준(level)에 있는 경우에는 마지막 성분힙 루트노드의 부모노드를 lastRoot로 하고, 인수 theKey와 INF 값을 각각 lastRoot 노드의 lKey과 rKey로 저장한 후, heapifyLeft( )을 호출하여 힙조정을 한다(라인 21~25).

```
1  bool insert( theKey )
2  {
3     // check if the rKey of the last root is infinite value.
4     if( arr[lastRoot].rKey == INF ) {
5        arr[lastRoot].rKey = theKey;
6        heapifyRight( lastRoot );
7        return true;
8     }
9
10    if( lastRoot == 1 ) return false;  // heap is full.
11
12    if( lastRoot % 2 == 0 ) {  // create a new component heap
13       lastRoot = ++lastLeaf;
14       if( lastLeaf > maxIndex )
15          lastRoot = lastLeaf = lastLeaf/2;
16       arr[lastRoot].lKey = theKey;
17       arr[lastRoot].rKey = INF;
18       return true;
19    }
20
21    lastRoot /= 2;
22    arr[lastRoot].lKey = theKey;
23    arr[lastRoot].rKey = INF;
24    heapifyLeft( lastRoot );
25    return true;
26 }
```

Fig. 4. Insertion Algorithm

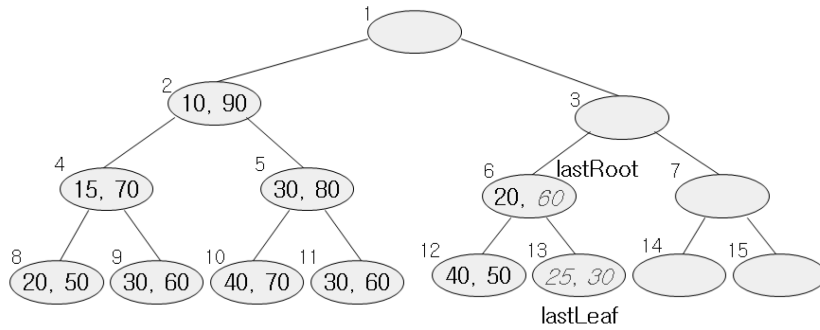

Fig. 5. IMI-heap after inserting key 25 into Fig. 1

Fig. 5는 Fig. 1에 키 25를 삽입한 후의 IMI-힙을 보여주고 있다. 즉 키 25로 lastRoot의 INF값을 대체한 후 힙조정한 결과를 보여주고 있다. Fig. 5에 키 35를 추가로 삽입하는 경우, (35, INF)가 노드 14에 삽입되고, lastRoot와 lastLeaf는 노드 14를 가리키게 된다.

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

4.4 최소키 삭제

최소키는 성분힙 루트의 lKey 중 가장 작은 값이므로, 먼저 모든 성분힙의 루트를 조사하여 가장 작은 lKey 값을 가지는 노드를 찾고, 그 노드의 lKey 값을 lastRoot 노드의 lKey로 대체한다. 대체된 노드를 루트로 하는 성분힙에 대해 좌단 힙조정을 수행하고, lastRoot와 lastLeaf 값을 조정함으로서 삭제가 완성된다.

Fig. 6의 삭제 알고리즘에서 먼저 최소키를 가지고 있는 minRoot를 찾는다(라인 5-10). 찾은 minRoot의 lKey를 lastRoot의 lKey로 대체하고 minRoot로부터 좌단 lKey에 대해 힙조정을 한다. 여기서, 어떤 성분힙 루트 r에서 인접하는 왼쪽 성분힙의 루트는 r이 짝수가 아닐때까지 부모 노드를 따라 올라가서 1을 감소시킴으로써 찾을 수 있다. 이를 이용하여 IMI-힙에서 최대 또는 최소 값을 가진 노드를 $O(\log n)$ 시간에 찾을 수 있다.

```
 1 keyType removeMin( )
 2 {
 3    if( lastRoot == 0 ) throws EmptyException; // heap is empty.
 4    // find min node (minRoot)
 5    curRoot = minRoot = lastRoot;
 6    while( curRoot != firstRoot ) { // firstRoot: root of leftmost CH
 7       while( curRoot%2 == 0 ) curRoot /= 2;
 8       curRoot--;
 9       if( arr[curRoot].lKey < arr[minRoot].lKey ) minRoot = curRoot;
10    }
11    removedKey = arr[minRoot].lKey;
12    arr[minRoot].lKey = arr[lastRoot].lKey; // replace
13    heapifyLeft( minRoot );
14
15    if( arr[lastRoot].rKey != INF ) { // lastRoot had no INF
16       arr[lastRoot].lKey = arr[lastRoot].rKey;
17       arr[lastRoot].rKey = INF;
18       heapifyLeft( lastRoot );
19       return removedKey;
20    }
21    // Now, lastRoot node is empty: update lastRoot and lastLeaf
22    if( lastRoot != lastLeaf ) {
23       lastRoot = 2*lastRoot + 1;
24    } else { // lastRoot is a leaf
25       lastLeaf--;
26       if( 2*lastLeaf < maxIndex ) lastLeaf = maxIndex;
27       lastRoot = 0;
28       if( lastLeaf != startLeaf - 1 ) { // not empty
29          lastRoot = lastLeaf;
30          while( lastRoot%2 == 1 ) lastRoot /= 2;
31       }
32    }
33    return removedKey;
34 }
```

Fig. 6. Remove Algorithm of Minimum Key

lastRoot의 rKey 값이 INF가 아닌 경우, 그 rKey로 lKey를 대체하고, rKey에는 INF 값을 넣고 좌단힙조정을 한다(라인 15~20). 그렇지 않은 경우, lastRoot와 lastLeaf를 수정한다(라인 22~32). lastRoot가 내부노드인 경우 그 오른쪽 자

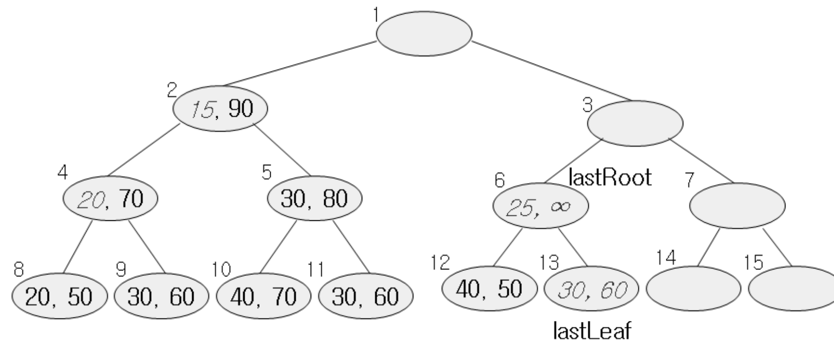

Fig. 7. IMI-heap after Deleting Minimum Key from Fig. 5

식이 lastRoot가 되고(라인 23), 리프노드인 경우 인접한 왼쪽 성분힙의 루트가 lastRoot가 된다(라인 29~30).

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

(증명) IMI-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 heapifyRight( )은 힙의 높이만큼의 시간이 걸리므로 $O(\log n)$ 시간이 걸린다. 또한 라인 30의 lastRoot 변수를 수정하는데 트리 높이인 $O(\log n)$ 시간이 걸린다. 따라서, 최소값 삭제 알고리즘은 $O(\log n)$ 시간 복잡도를 가진다.

Fig. 7은 Fig. 5로부터 최소키 10을 삭제한 후의 IMI-힙을 보여주고 있다. 먼저 노드 2의 키 10을 lastRoot의 키 20으로 대체하고 노드 2에 대해 좌단 힙조정을 한다. 그 후, lastRoot의 rKey인 60을 lKey로 대체하고 rKey에는 INF로 대체한 후 노드 6에 대해 좌단 힙조정을 한다.

최대키 삭제 알고리즘도 최소키 삭제와 유사하게 만들 수 있다. IMI-힙에서의 최대키 삭제는 최대키를 가지고 있는 성분힙의 루트를 찾은 후, 그 최대키를 마지막 성분힙의 루트인 lastRoot 노드의 rKey 값으로 대체하고, 대체된 성분힙에 대해 우단 힙조정을 한다. 그 후, lastRoot와 lastLeaf 값를 조정함으로써 삭제가 완성된다. 최대키를 찾을 때 lastRoot의 rKey 값이 INF일 경우에는 lastRoot의 두 자식 노드의 rKey 값도 비교하여야 한다.

5. 결 론

본 고에서 제안한 IMI-힙은 배열을 이용하여 묵시적으로 부모-자식 관계를 표현하는 묵시 양단 우선순위 큐 자료구조이다.

포인터를 사용하는 양단 우선순위 큐를 제외한 기존의 묵시 양단 우선순위 큐에서는 삽입과 최소값/최대값 삭제를 하는데 모두 $O(\log n)$ 시간이 걸렸지만, 본 논문에서 제안한 묵시 IMI-힙에서는 삽입과 최대값/최소값 삭제 연산에 각각 O(1) 전이 시간과 $O(\log n)$ 시간이 걸린다.

References

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

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

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

[4] 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.

[5] H. Jung, "The d-deap*: A fast and simple cache-aligned d-ary deap," *Information Processing Letters*, Vol.93, No.2, pp. 63-67, Jan. 2005.

[6] J. van Leeuwen and D. Wood, "Interval heaps," *The Computer Journal*, Vol.36, No.3, pp.209-216, 1993.

[7] Y. Ding and M. Weiss, "On the Complexity of Building an Interval Heap," *Information Processing Letters*, Vol.50, pp.143-144, 1994.

[8] S. Sahni, Data Structures, Algorithms, & Applications in Java: Double-Ended Priority Queues [Internet], https://www.cise.ufl.edu/~sahni/dsaaj/enrich/c13/double.htm.

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

[10] H. Jung, "A double-ended priority queue with O(1) insertion amortized time," *KIPS Journal A*, Vol.16, No.A(3), pp. 217-222, Jun. 2009.

정 해 재

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

e-mail : hjjung@anu.ac.kr

1984년 경북대학교 전자공학과(학사)

1987년 서울대학교 컴퓨터공학과(석사)

2000년 CISE, Univ. of Florida(Ph.D.)

1988년~1995년 ETRI 선임연구원

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

2005년~현 재 안동대학교 정보통신공학과 교수

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