캐쉬 효과를 고려한 확장된 Pairing Heap 알고리즘

(Extended Pairing Heap Algorithms Considering Cache Effect)

정 균 락 † 김 경 훈 ††
(Kyun-Rak Chong) (Kyoung-Hoon Kim)

요 약 빠른 프로세서 속도에 비해 메모리 접근(access)하는 시간이 상대적으로 느려짐에 따라, 대부분의 시스템은 격차를 줄이기 위하여 캐쉬(cache)라는 매우 빠른 메모리를 사용하고 있으며 캐쉬 메모리를 얼마나 효과적으로 사용하는 가 하는 문제는 알고리즘의 성능에 있어서도 결정적인 영향을 미치게 된다. 블록을 사용하는 방법은 캐쉬의 효율성을 향상시키는 방법으로 잘 알려져 있으며 행렬곱셈이나 d-heap과 같은 탐색트리에 사용되어 좋은 결과를 내고 있다. 그러나 삽입과 삭제 연산시 트리의 회전(rotation)이 필요한 자료구조에서는 블록을 사용하면 블록사이에 데이터의 이동이 필요해서 실행시간이 증가하게 된다. 본 논문에서는 블록을 사용하는 pairing heap에서 개선된 삽입과 삭제 알고리즘을 제안하였고 실험을 통해 우수성을 입증하였다. 또 블록을 사용하는 경우 여러 개의 데이터를 한 블록에 저장하므로 포인터의 개수가 줄어들게 되어 메모리를 적게 사용하게 된다.

키워드 : 페어링힙, 우선순위큐, 캐쉬메모리, 자료구조, 알고리즘

Abstract As the memory access time becomes slower relative to the fast processor speed, most systems use cache memory to reduce the gap. The cache performance has an increasingly large impact on the performance of algorithms. Blocking is the well known method to utilize cache and has shown good results in multiplying matrices and search trees like d-heap. But if we use blocking in the data structures which require rotation during insertion or deletion, the execution time increases as the data movements between blocks are necessary. In this paper, we have proposed the extended pairing heap algorithms using block node and shown by experiments that our structure is superior. Also in case of using block node, we use less memory space as the number of pointers decreases.

Key words : *Paring Heap, Priority Queue, Cache Memory, Data Structure, Algorithm*

1. 서 론

CPU 성능이 비약적으로 발전함에 따라 연산자를 처리하는데 드는 비용보다는 메모리에 접근(access)하는 비용이 상대적으로 증가되었고, 메모리 접근에 드는 비용을 줄이기 위하여 캐쉬(cache)라는 매우 빠른 메모리가 사용되고 있다.

캐쉬 메모리를 얼마나 효과적으로 사용하는가 하는 문제는 알고리즘의 성능에 있어서도 결정적인 영향을 미치게 된다. 캐쉬 효과를 고려한 알고리즘에 관한 연구는 많은 이들에 의해 다양한 방법으로 진행되어 왔다. Lam과 Rothberg 그리고 Wolf는 블록을 이용한 행렬의 곱셈 알고리즘에 대하여 연구하였는데, 블록을 사용하는 방법은 메모리 계층 구조의 효율성을 향상시키는 방법으로 잘 알려져 있다[1]. Lamarca와 Ladner는 캐쉬 메모리와 알고리즘과의 관계에 대해서 연구하였고[2], 또 heap 알고리즘에서의 캐쉬 효과 연구를 통하여 캐쉬 메모리 블록을 최대한으로 이용할 수 있는 d-heap 알고리즘을 제안하였다[3]. D-heap은 하나의 부모 노드가 두 개의 자식 노드를 갖는 기존의 heap 알고리즘과는 달리 d개의 자식 노드를 가질 수 있다. 그들은 "Collective Analysis" 방법을 이용하여 이론적으로 캐쉬 메모리를 효과적으로 사용하고 있음을 보였고 실험을 통하여 d-heap의 성능이 기존의 heap보다 뛰어남을 보였다. 또

† 종신회원 : 홍익대학교 컴퓨터공학과 교수
chong@cs.hongik.ac.kr
†† 비 회 원 : 포스데이타 SW개발부
kimkh@posdata.co.kr
논문접수 : 2002년 6월 12일
심사완료 : 2003년 1월 22일

한 그들은 정렬 알고리즘[4]과 탐색과 무작위 접근 알고리즘[5]에서의 캐쉬 효과에 대해서도 연구하였다. Chilombi와 Hill 그리고 Larus는 캐쉬 메모리에서의 데이터의 지역성을 높이기 위한 캐쉬 메모리 안에서의 연속적인 데이터 배치에 대한 점진적인 접근 방법을 제시하였다[6].

우선 순위 큐(priority queue)는 효율적인 삽입 연산과 최소 값(또는 최대 값) 삭제 연산을 지원하는 자료구조이다. [7]에서 S. Cho와 S. Sahni는 대표적인 우선순위 큐들에 대한 실행시간을 비교하였는 데 splay 트리가 가장 우수한 실행시간을 나타냈으며, 효율적인 합병(merge)을 제공하는 자료구조에서는 leftist 트리가 좋은 결과를 나타냈다. 그러나 블록을 사용한다고 모든 탐색트리에서 항상 성능이 향상되는 것이 아니며 AVL 트리나 leftist 트리, splay 트리 등과 같이 삽입과 삭제 연산시 트리의 회전(rotation)이 필요한 자료구조에서는 블록을 사용하면 블록사이에 데이터의 이동이 필요해서 실행시간이 증가하게 된다.

Pairing Heap(이하 PH)은 Freedman, Sedgewick 그리고 Sleator[8]에 의해 1986년에 제안되었는 데, O(1) 삽입 시간과 O(n) 최소 원소 삭제 시간 (amortized 시간은 O(logn))을 가지며 두 개의 PH을 O(1) 시간에 합병할 수 있는 자료 구조이다.

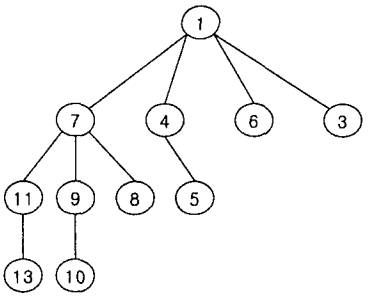
본 논문에서는 블록을 사용하는 PH에서 개선된 삽입과 삭제 알고리즘을 제안하였고 실험을 통해 splay 트리와 leftist 트리 알고리즘과 비교하였다. 또 PH은 블록을 사용하는 경우 여러 개의 데이터를 한 블록에 저장하므로 포인터의 개수가 줄어들게 되어 메모리를 적게 사용하게 된다.

2. Pairing Heap 알고리즘

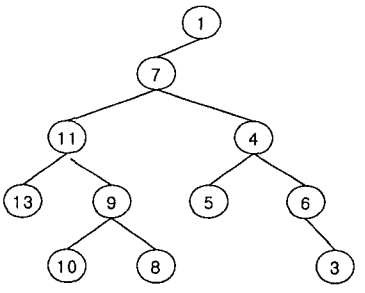
PH은 heap 구조에 기반한 우선 순위 큐(priority queue)로 삽입 연산과 최소 값 삭제 연산을 지원한다. PH에서 각 노드의 값은 자식 노드 값보다 작거나 같다. 그러므로 루트 노드의 값은 트리에 있는 값들 중 가장 작은 값이다. PH은 보통 트리의 이진 트리 표현을 사용하여 나타낸다. 그림 1(a)에 PH의 한 예가 나타나 있고, 그림 1(b)는 그림 1(a)의 예를 이진 트리로 표현한 것이다. 이진 트리로 표현한 경우 한 노드의 값은 그 노드를 루트로 하는 왼쪽 부 트리에 있는 모든 노드들의 값보다 작거나 같지만, 오른쪽 부 트리에 있는 노드들의 값보다 클 수가 있다.

PH의 삽입과 삭제연산은 비교-연결(Comparison-Link) 연산을 사용한다[8]. 비교-연결은 두 트리의 루트를 비교하여 크기가 작은 쪽이 크기가 큰 쪽의 부모가 되게 두 트리를 연결하는 것이다. 두 개의 PH를 P_A와 P_B라 하고 루트 노드를 각각 R_A와 R_B라 하자. R_A의 데이터 값을 x라 하고 R_B의 데이터 값을 y라 할 때, x<y이면 R_A의 왼쪽 자식이 R_B가 되고 원래 R_A의 왼쪽 자식은 R_B의 오른쪽 자식이 된다. x>y이면 R_B의 왼쪽 자식이 R_A가 되고 원래 R_B의 왼쪽 자식은 R_A의 오른쪽 자식이 된다(그림 2 참조).

PH의 삽입 연산은 비교-연결을 사용한다. v를 삽입

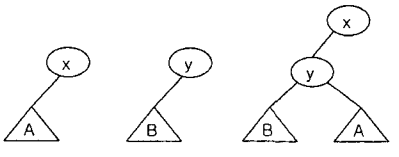


(a) 트리 표현

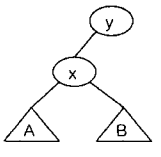


(b) 이진 트리 표현

그림 1 pairing heap의 예



(a) PH P_A와 P_B (b) x<y인 경우



(c) x>y인 경우

그림 2 비교-연결 연산

하는 경우에는 v를 루트로 하는 PH과 기존의 PH과 비교-연결 연산을 수행하면 된다.

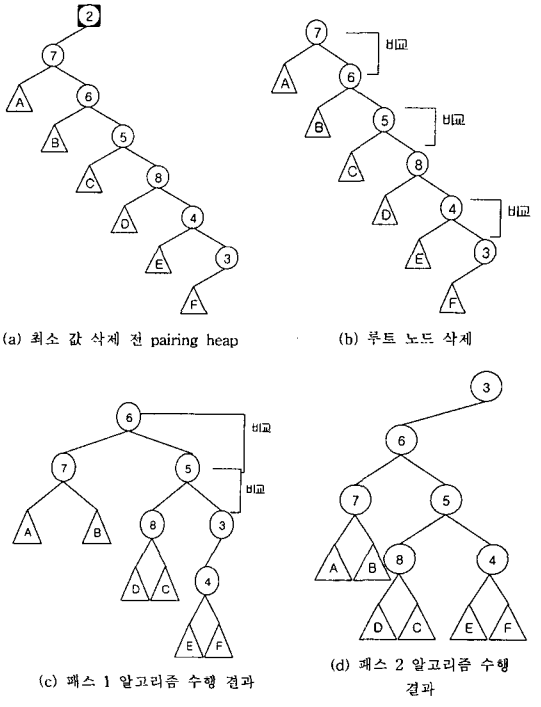
PH에서의 최소 값 삭제 연산은 이중 패스(two pass) 알고리즘이나 다중 패스(multi pass) 알고리즘을 이용하여 수행되어지는 데[9], 본 논문에서는 이중 패스 알고리즘을 사용하였기 때문에 이중 패스 알고리즘에 대해서만 간단히 설명하고자 한다. PH은 최소 값을 항상 루트 노드에 유지하고 있다. 최소 값 삭제를 위해서는 우선 루트 노드를 삭제해야 한다. 루트 노드를 삭제하고 난 다음에는 최소 값을 포함하고 있는 노드가 새로운 루트 노드가 될 수 있도록 트리를 재구성하여야 한다. 트리를 재구성하는 과정에서 이중 패스 알고리즘을 사용한다. 이진 트리로 표현된 PH에서 패스 1 알고리즘은 루트 노드에서 잎 노드까지 오른쪽 자식 노드를 따라서 이루어진다. 제거된 루트 노드의 왼쪽자식을 루트로 하는 PH을 S_1이라 하고, S_1의 오른쪽자식을 루트로 하는 PH을 S_2, ..., S_{n-1}의 오른쪽자식을 루트로 하는 PH을 S_n이라 하자. 편의상 n을 짝수라 하자(n이 짝수가 아니면 n-1번째 PH까지 비교-연결을 수행하고 n번째 PH은 그대로 둔다). 그러면 일련의 PH들 S_1, S_2, ..., S_n에서 앞에서 두 개씩 차례로 비교-연결 연산을 수행한다. 즉 S_1과 S_2, S_3과 S_4, ..., S_{n-1}과 S_n 대해 비교-연결을 수행하고 그 결과를 각각 T_1, T_2, ..., T_k(k=n/2)라 하면 패스 2 알고리즘은 다음과 같이 수행된다. 먼저 T_{k-1}과 T_k에 대해 비교-연결을 수행한다. 그 결과 만들어진 PH을 P_{k-1}라 하면 P_{k-1}과 T_{k-2}에 대해 비교-연결을 수행한다. 그 결과 만들어진 PH을 P_{k-2}라 하면 P_{k-2}과 T_{k-3}에 대해 비교-연결을 수행한다. 이런 식으로 하면 P_2과 T_1과 비교-연결을 수행한 결과 만들어진 PH이 우리가 원하는 PH이 된다. 그림 3에 이중패스 알고리즘이 나타나 있다. 여기서 C=Comparison-Link(A, B)는 PH A와 PH B에 대해 비교-연결 연산을 수행한 후 그 결과를 PH C로 넘겨주는 함수이다.

```
TwoPass(S1)
    Let S1 be the left child of the root and
    Si+1 be the right child of Si, 1 ≤ i ≤n-1.
    for (i = 1 ; i ≤ n/2; i += 2)
        Ti = Comparison-Link(S2i-1, S2i);
    Pn/2 = Tn/2;
    for (i = n/2 ; i ≤ 2; i--)
        Pi-1 = Comparison-Link(Pi, Ti-1);
    return (P1);
```

그림 3 Two Pass 알고리즘

그림 4는 PH에서의 최소 값 삭제 연산의 예를 보여준다. 그림 4(a)는 삭제 연산이 수행되기 전의 트리이고, A. B, C, D, E, F는 왼쪽 부 트리로 부모 노드의 값보다 큰 값들을 유지하고 있다.

최소 값을 삭제하기 위해서 우선 PH에서의 최소 값인 '2'를 포함하는 루트 노드를 삭제한다(그림 4(b)). 그리고 난 뒤 패스 1 알고리즘을 수행한다. 패스 1 알고리즘은 루트 노드에서부터 잎 노드까지 수행되어 지며, 그림에서처럼 '7'과 '6', '5'와 '8' 그리고 '4'와 '3'을 비교하여 '7', '8', '4'를 포함하는 노드가 '6', '5', '3'을 포함하는 노드의 왼쪽 자식 노드가 된다. 그리고 '6', '5', '3'의 왼쪽 부 트리인 B, C, F는 '7', '8', '4'를 포함하는 노드의 오른쪽 부 트리가 된다(그림 4(c)). 마지막으로, 패스 2 알고리즘을 수행한다. 패스 1 알고리즘과는 달리 잎 노드에서부터 루트 노드까지 수행되어 진다. 우선 '3'과 '5'를 비교한 후, '5'를 포함하는 노드가 '3'을 포함하는 노드의 왼쪽 자식 노드가 되고 '4'를 루트 노드로 하는 '3'의 왼쪽 부 트리는 '5'의 오른쪽 부 트리가 된다. 그리고 난 뒤, '3'과 '6'을 비교한다. '6'을 포함하는 노드가 '3'을 포함하는 노드의 왼쪽 자식 노드가 되고, '5'를 루트 노드로 하는 '3'의 왼쪽 부 트리는 '6'의 오른쪽 부 트리가



(a) 최소 값 삭제 전 pairing heap (b) 루트 노드 삭제

(c) 패스 1 알고리즘 수행 결과 (d) 패스 2 알고리즘 수행 결과

그림 4 Pairing heap에서 최소 값 삭제 연산의 예

된다. 그 결과로 '3'을 포함하는 노드를 새로운 루트 노드로 하는 PH이 얻어진다(그림 4(d)).

3. Heap 기반 K-블록노드를 이용한 확장된 Pairing Heap

Pairing heap에서 최소 값 삭제 연산의 경우 비교-연결을 사용하게 되는 데 한 번에 오직 두 개의 노드의 값만을 비교하기 때문에 비교 회수가 너무 많아지고, 노드들이 동적으로 할당되어 있기 때문에 캐쉬 메모리의 활용율을 떨어뜨린다. 캐쉬 메모리를 효율적으로 사용하기 위한 보편적인 방법은 여러 개의 데이터를 블록에 저장하는 것이다.

먼저 heap 기반 K-블록노드를 다음과 같이 정의하기로 한다. K-블록 노드는 K개의 데이터를 저장할 수 있는 일차원 배열과 두 개의 자식 블록노드를 가리키는 포인터로 이루어진다. 블록노드 안의 K개의 데이터는 자식 노드의 데이터 값이 항상 부모 노드의 데이터 값보다 큰 heap 구조를 갖는다. 그림 5는 K=6일 때의 heap 기반 블록노드의 일차원 배열 구조와 heap으로 표현된 구조가 나타나 있다.

블록노드를 사용하게 되면 데이터를 삽입하거나 최소 값을 삭제할 때 데이터의 이동이 발생하게 된다. 그러면 꽉 차지 않은 블록노드들이 생기게 되고 삽입과 삭제가 진행됨에 따라 메모리의 손실이 점점 더 커지게 된다. 본 논문에서는 이러한 문제를 해결하기 위하여 버퍼를 사용한다. 데이터 삽입 시 데이터를 버퍼에다 삽입한 다음 버퍼가 차면 버퍼를 PH에 삽입한다.

본 연구에서 제안하는 확장된 pairing heap(이하 BPH)은 버퍼와 트리로 이루어져 있고, 버퍼와 트리의 각 노드는 heap 기반 K-블록노드들이다. 그림 6은 K = 6인 heap 기반 K-블록노드를 이용한 BPH의 예를 나타낸다.

BPH은 K-블록노드의 데이터 배열의 첫 번째 데이터를 가지고 이진 트리를 유지한다. 배열의 첫 번째 데이터가 블록노드 안의 데이터들 중 최소 값이므로 두 개의 블록노드의 첫 번째 데이터 값만을 비교하고도 이 두 개의 블록노드에서의 최소 값을 결정할 수 있다.

블록을 사용할 때 비교-연결은 PH에서의 비교-연결과 유사한 데, 다른 점은 BPH에서는 두 블록노드의 첫 번째 원소(최소 원소)들을 비교하여 부모 자식 관계를 결정한다. 두 개의 BPH를 P_A와 P_B라 하고 루트 노드를 각각 R_A와 R_B라 하자. R_A의 최소 원소를 x라 하고 R_B의 최소 원소를 y라 할 때, x<y이면 R_A의 왼쪽 자식이 R_B가 되고 원래 R_A의 왼쪽 자식은 R_B의 오른쪽

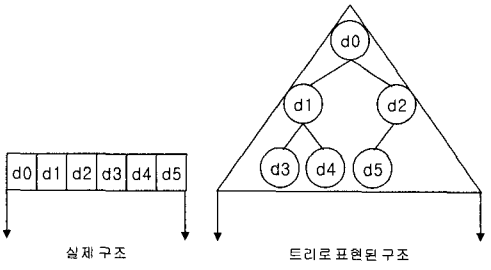


그림 5 Heap 기반 블록노드

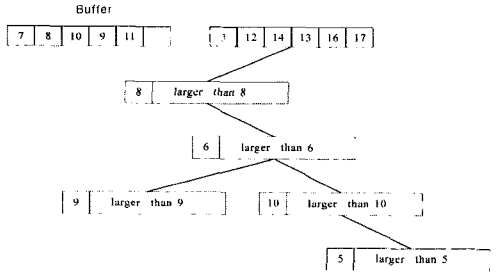


그림 6 확장된 pairing heap의 예

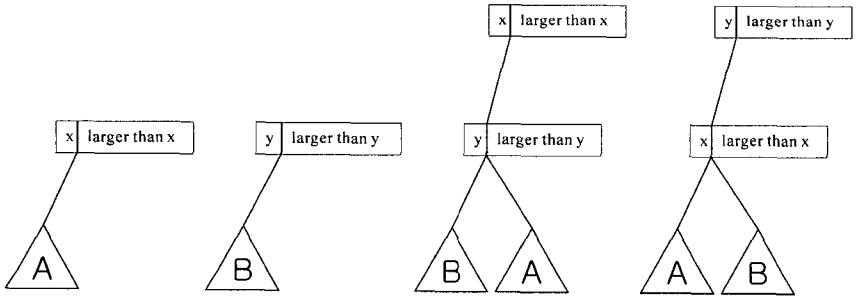


그림 7 블록을 이용한 비교-연결

자식이 된다. x>y이면 R_B의 왼쪽 자식이 R_A가 되고 원래 R_B의 왼쪽 자식은 R_A의 오른쪽 자식이 된다(그림 7 참조).

3.1 삽입 연산

삽입 연산은 우선 삽입하고자하는 데이터 v를 버퍼에 넣는데 heap에서 삽입 알고리즘을 사용하여 버퍼에 넣는다. 이 때 버퍼가 가득 차 있다면 버퍼와 트리와의 비교-연결을 수행하여 새로운 BPH를 만들어 주면 삽입 연산이 완료된다. 그림 8에 삽입 알고리즘이 나타나 있다. 여기서 InsertHeap(x, buffer)는 heap 구조를 갖는 buffer에 원소 x를 삽입하는 알고리즘[7]이고, Comparison-Link는 K-블록 노드를 사용하는 비교-연결 연산을 의미한다. 크기가 상수(K)인 버퍼에 삽입하는 시간과 비교-연결 시간이 모두 O(1)이므로 삽입시간은 O(1)이다.

```
void Insert(element x)
{
    InsertHeap(x, buffer);
    if ( buffer is not full )  return;
    else {
        root = Comparison-Link(root, buffer);
        GetNode(buffer);
    }
}
```

그림 8 삽입 연산 알고리즘

3.2 최소 원소 삭제

최소 원소 삭제는 버퍼와 트리가 비어 있는 가에 따라 세 가지 경우로 나누어진다. 트리와 버퍼가 모두 비어 있는 경우는 제외한다.

1) 트리가 비어있는 경우

버퍼로부터 heap에서의 최소 원소 삭제 알고리즘을 사용하여 버퍼의 첫 번째 데이터를 삭제하면 된다.

2) 버퍼가 비어있는 경우

트리의 루트 블록노드에서 heap에서의 최소 원소 삭제 알고리즘을 사용하여 첫 번째 데이터를 삭제한 후, 루트 블록노드를 버퍼로 한다. 루트 블록 노드를 제외한 나머지 트리에 pairing heap에서의 패스 1 알고리즘과 패스 2 알고리즘을 수행하면 된다.

3) 트리와 버퍼 모두 비어있지 않은 경우

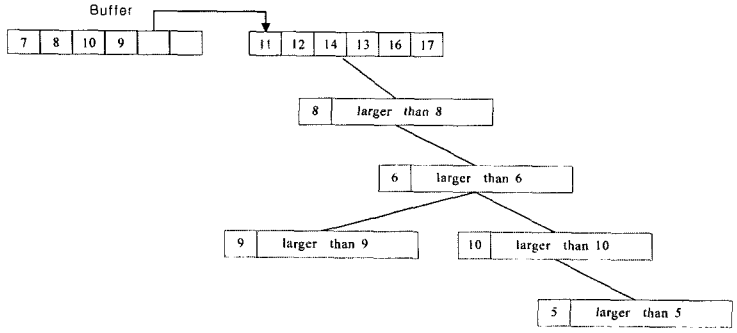
버퍼의 첫 번째 데이터와 루트 블록노드의 첫 번째 데이터 값을 비교한 후, 버퍼의 첫 번째 데이터 값이 작다면 버퍼의 첫 번째 데이터를 삭제하면 된다. 반대로, 루트 블록노드의 첫 번째 데이터 값이 작다면 루트 블록노드의 첫 번째 데이터 값을 삭제하고 버퍼의 맨 끝에 있는 데이터를 가져와서 이 데이터와 루트 블록노드의 나머지 데이터를 가지고 heap을 새로 구성한다, 루트 블록노드의 첫 번째 데이터가 변했기 때문에 이제 루트 블록노드의 왼쪽 자식 블록노드의 첫 번째 데이터보다 작다는 것을 보장할 수 없으므로 루트 블록노드의 왼쪽 자식 노드를 오른쪽 자식노드로 옮긴다. 그리고 나서 루트 블록노드에서부터 패스 1 알고리즘과 패스 2 알고리즘을 수행하면 최소 값 삭제 연산이 완료된다.

BPH에서 최소 원소 삭제 알고리즘이 그림 9에 나타나 있다. 여기서 DeleteMinHeap(B, x)는 K-블록노드 B에서 최소 원소 x를 삭제하는 알고리즘이다[7]. BPH에서 최소 원소 삭제 시간은 최악의 경우 n/K개의 노드에 대해 비교-연결을 수행해야 하므로 O(n)이고 armortized 시간은 O(logn)이다. 그림 10은 그림 6의 BPH 예에서 heap 기반 K-블록노드를 이용한 BPH에서의 최소 값 삭제 연산의 과정을 보여준다. 최소 값을 삭제하기 위해서 우선 버퍼의 첫 번째 데이터 값 '7'과 트리의 루트 블록노드의 첫 번째 데이터 값 '3'을 비교한다. '3'이 작으므로 '3'을 삭제하고 버퍼의 맨 끝 데이터 값 '11'과 루트 블록노드의 나머지 데이터 값 '12', '14', '13', '16', 그리고 '17'을 가지고 heap을 구성한 후, 루트 블록노드의 왼쪽 자식 노드를 오른쪽 자식 노드로 만들기 위해서 포인터를 이동한다(그림 10(a)). 그리고 난 뒤 패스 1 알고리즘을 수행한다. 패스 1 알고리즘은 루트 노드에서부터 잎 노드까지 수행되어 지며, 그림에서처럼 '11과 '8', '6'와 '10'을 비교하여 '11', '10'을 포함하는 블록노드가 '8', '6'을 포함하는 블록노드의 왼쪽 자식 노드가 된다(그림 10(b)). 마지막으로, 패스 2 알고리

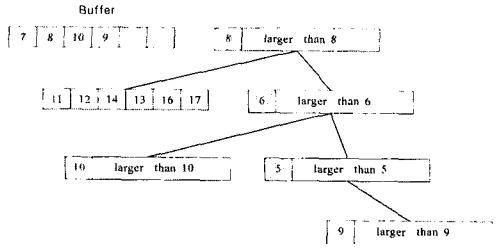
```
void delete_min( void) {
  if ( buffer is empty and tree is empty) error return ;
  if ( tree is empty ) {
      DeleteMinHeap(buffer, x)
      return;
  }
  if ( if buffer is empty ) {
      DeleteMinHeap(root, x);
      buffer <- root;
      root = TwoPass(root->left);

  } else if ( min element in buffer < min element in root ) {
      DeleteMinHeap(buffer, x);
  } else {
      DeleteMinHeap(root, x);
      InsertHeap(the last element in buffer, root);
      root->right = root->left;
      root->left = NULL;
      root = TwoPass(root);
  }
```

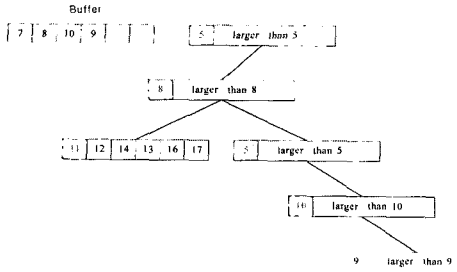
그림 9 최소 값 삭제 연산 알고리즘



(a)버퍼의 데이터를 루트 블록노드로 옮기고 포인터 이동



(b) 패스 1 알고리즘 수행 결과



(c) 패스 2 알고리즘 수행 결과

그림 10 최소 값 삭제 연산 예

즘을 수행한다. 우선 '5'와 '6'을 비교한 후, '6'을 포함하는 블록노드가 '5'를 포함하는 블록노드의 왼쪽 자식 노드가 된다. 그리고 난 뒤, '5'와 '8'을 비교한다. '8'을 포함하는 블록노드가 '5'를 포함하는 블록노드의 왼쪽 자식 노드가 된다. 최종적으로 '5'를 포함하는 블록노드를 새로운 루트 블록노드로 하는 트리가 얻어진다(그림 10(c)).

4. 실험 결과

제안된 알고리즘은 C로 구현되어 SUN ENTERPRISE 3000과 ULTRA60에서 실험되었다. 데이터는 4바이트 정수를 사용하였고, 따라서 K-블록노드의 크기는 (K+2) ×4 바이트이다. 실험은 정지 모델(hold model)을 사용하였는데 먼저 각각의 자료구조에 N개의 데이터를 삽입한 후 확률 0.5로 삽입과 삭제 연산을 각각 M번 실행시킨다. 삽입과 삭제의 확률이 0.5이기 때문에 자료 구조의 크기는 대략 N을 유지하게 된다. 먼저 블록노드의 크기에 대한 실행시간을 비교하기 위해서 여러 개의 K 값에 대해 실험하였다. 표 1은 정지 모델에서 $M=N=100$ 만을 사용하고, K=6, 14, 30, 62, 126 즉 K-블록노드의 크기가 32, 64, 128, 256, 512 바이트일 때 결과이고 실행 시간의 단위는 ms이다.

표 1 K값에 대한 실행 시간 비교

K	블록노드의 크기	실행 시간
6	32	2054000
14	64	1682000
30	128	1840000
62	256	1788000
126	512	1810000

실험 결과에 의하면 K=14일 때 즉 블록노드의 크기가 64바이트일 때 heap 기반 K-블록노드를 이용한 확장된 pairing heap의 성능이 가장 뛰어남을 알 수 있다. 이것은 블록노드의 크기를 캐쉬 메모리 블록의 크기로 만들었을 때 성능이 가장 뛰어남을 보여주는 것이다.

다음에 heap 기반 14-블록노드를 이용한 확장된 pairing heap 알고리즘을 기존의 pairing heap, splay 트리와 leftist 트리와 비교했다. 그림 11은 pairing heap(PH), leftist 트리(LT), splay 트리(ST), 그리고 K =14인 heap 기반 K-블록노드를 이용한 확장된 pairing heap(BPH) 알고리즘을 비교한 결과이다. 실행 시간의 단위는 ms이다. 실험은 앞에서와 마찬가지로 정지 모델을 사용하였는데 초기 자료 구조의 크기 N은 1K, 10K,

100K, 500K, 1000K를 사용하였고 M은 1000K를 사용하였다. N이 크기가 작을 때는 splay 트리 알고리즘이 가장 빠르지만 N이 10만보다 커지면 heap 기반 K-블록노드를 사용한 확장된 pairing heap 알고리즘이 가장 빠르게 된다. 위에서 살펴보았듯이, heap 기반 K-블록 노드를 사용한 확장된 pairing heap 알고리즘은 데이터의 수가 많아질수록 캐쉬 효과가 많이 나타나게 된다.

그림 12에는 정지 모델에서 삽입과 삭제 연산시 원소를 하나씩 삽입하거나 삭제하는 것이 아니라 100개씩 삽입하거나 삭제할 경우 실행시간이 나타나 있다. 실행은 SUN ULTRA 60 워크스테이션을 사용하였다. 결과를 보면 원소하나씩 삭제하거나 삽입하는 경우와 유사한 것을 알 수 있다. 그림 13에는 삽입과 삭제 연산의 비율이 각각 0.75와 0.25인 경우 즉 삭제에 비해 삽입의 비율이 더 많은 경우에 각 자료 구조의 실행시간이 나타나 있다. 실행은 SUN ULTRA 60 워크스테이션을 사용하였다. 이 경우에는 pairing heap과 heap 기반의 pairing heap이 leftist 트리나 splay 트리에 비해 월등히 좋은 것을 알 수 있는 데, 그 이유는 전자의 경우 삽입 시간이 O(1)이나 후자의 구조들은 삽입 시간이 O(logn)이기 때문이다.

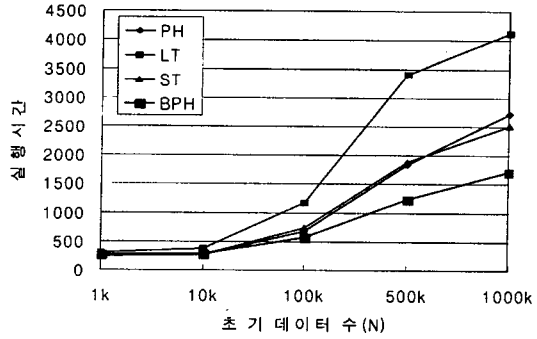


그림 11 삽입과 삭제 확률이 0.5인 경우 실행시간 비교

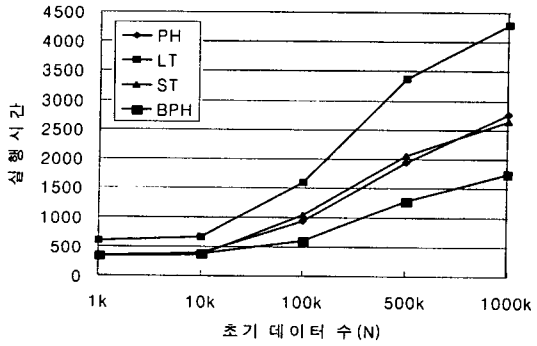


그림 12 100개씩 삽입과 삭제를 할 경우 실행 시간 비교

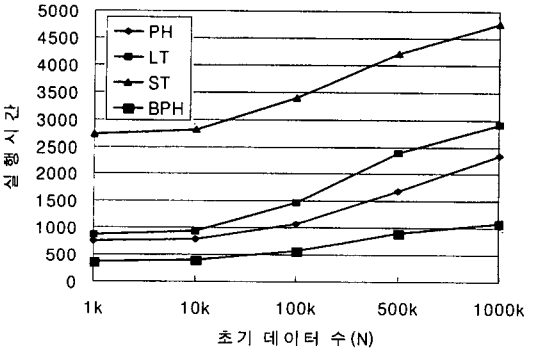


그림 13 삽입과 삭제 확률이 0.75과 0.5인 경우 실행시간 비교

각각의 자료구조에서 필요한 메모리의 양을 보면 PH, LT, ST는 이진 트리 구조를 가지기 때문에 데이타형이 4 바이트 정수이고 포인터가 4 바이트라고 가정하면 데이터의 개수가 N이면 12N 바이트가 필요하고, 블록 노드의 크기가 64 바이트인 BPH는 64 x ⌈N/14⌉ 바이트가 필요하게 된다. 예를 들어 N=100만이면 이진 트리 구조는 12MB가 필요하지만 BPH는 약 4.6MB가 필요하게 되어 60%의 메모리를 절약할 수 있게 된다.

5. 결 론

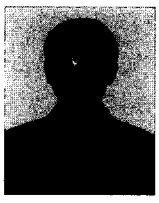
대부분의 시스템에서는 프로세서와 메모리의 속도 차이를 줄이기 위하여 캐쉬 메모리를 사용한다. 캐쉬를 얼마나 효율적으로 사용하느냐의 문제는 시스템 성능뿐만 아니라 알고리즘의 성능에 있어서도 결정적인 영향을 준다.

본 논문에서는 블록과 버퍼를 이용한 pairing heap 알고리즘을 제안하였고 실험을 통해 우수함을 보였다. 블록을 이용하여 다수의 데이터를 저장하면 포인터에 필요한 공간을 줄일 수 있는 데. 데이터가 정수형 일 때는 약 60%의 공간을 줄일 수 있다. 또 블록을 사용하면 트리를 재구성할 때 원소의 비교횟수를 줄일 수 있고 캐쉬의 효율도 증가되어 실행 시간도 약 30% 감소하게 된다.

참 고 문 헌

[1] M. S. Lam and E. E. Rothberg. The Cache Performance and Optimizations of Blocked Algorithms. Conf on Architectural Support for Program-

ming Languages and Operating Systems (ASPLOS Ⅳ), pages 63-74, 1991.
[2] A. LaMarca and R. E. Ladner. The Influence of Caches on the Performance of Heaps. Journal of Experimental Algorithmics, Vol 1, Article 4, 1996.
[3] A. Lamarca and R. E. Ladner. The Influence of Caches on the Performance of Sorting . In Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 370-379, 1997.
[4] R. E. Ladner, J. D. Fix and A. LaMarca. Cache Performance Analysis of Traversals and Random Accesses. In Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 613-622, 1999.
[5] A. LaMarca. Caches and Algorithms. Ph. D. Dissertation, University of Washington, May 1996.
[6] Trishul M. Chilimbi, Mark D. Hill and James R. Larus. Making Pointer-Based Data Structures Cache Conscious. IEEE Computer pp. 67-74, Dec. 2000.
[7] Eliss Horowitz, Sartaj Sahni, and Susan Anderson-Freed, Fundamental of Data Structure in C, Computer Science Press, 1993.
[8] Fredman, M.L., and Sdegewick, R., Sleator, D.D., and Tarjan, R.E. The pairing heap: A new form of self-adjusting heap. Algorithmica 1 (Mar, 1986.) 111-129.
[9] John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: Experiments and Analysis. ACM (Mar 1987.) Volume 30 Number 3.
[10] S. Cho and S. Sahni, "Weighted Biased Leftist Trees and Modified Skip List", ACM Journal on Experimental Algorithmics, 1998.



정 균 락

1978년 서울대학교 계산통계학과 학사. 1980년 한국과학기술원 전산학과 석사. 1991년 미네소타대학 박사. 1980~1984년 한국과학기술원 연구원. 1998년 플로리다대학 방문교수. 1991년~현재 홍익대학교 컴퓨터공학과 교수. 관심 분야는 알고리즘 설계 및 분석, VLSI 알고리즘, 병렬 알고리즘

김 경 훈

1999년 홍익대학교 컴퓨터공학과 학사. 2001년 홍익대학교 전자계산학과 석사. 현재 포스데이타 SW개발부 연구원. 관심분야는 알고리즘 설계, 인터넷 컴퓨팅, 멀티미디어