4-딥* : 캐쉬를 이용한 빠른 4-원 딥

정 해 재†

요 약

스케줄링이나 정렬과 같은 응용에 이용될 수 있는 양단 우선순위 큐는 포인터를 사용하는 것과 포인터를 이용하지 않고 묵시적으로 표현하는 두 가지가 있다. 묵시 자료 구조는 메모리 이용에 있어서 포인터를 사용하는 것보다 효율적이다. 본 논문에서는 캐쉬 메모리를 효율적으로 이용하는 새로운 묵시 양단 우선순위 큐인 4-딥*를 제안한다. 실험을 통하여, 제안된 4-딥*가 이진 트리에 근거한 딥뿐만 아니라 대칭 최소-최대 힙보다 빠름을 보인다.

4-Deap* : A Fast 4-ary Deap using Cache

Haejae Jung†

ABSTRACT

Double-ended Proirity queues(DEPQ) can be used in applications such as scheduling or sorting. The data structures for DEPQ can be constructed with or without pointers. The implicit representation without pointers uses less memory space than pointer-based representation. This paper presents a novel fast implicit heap called 4-deap* which utilizes cache memory efficiently. Experimental results show that the 4-deap* is faster than symmetric min-max heap as well as deap.

키워드 : 자료 구조(Data Structures), 양단 우선순위 큐(Double-ended Priority Queue), 묵시(Implicit), 딥(Deap), 최소 힙(Min Heap), 최대 힙(Max Heap)

1. 서 론

양단 수선순위 큐(DEPQ)는 운영 체제 스케줄링, 사건 시뮬레이션, 또는 정렬과 같은 응용에 사용될 수 있는 자료 구조로서, 집합 S에 대해 적어도 다음의 세 가지 연산을 지원한다.

- Insert(S, x) : 임의의 키를 가지는 새로운 데이터 x를 집합 S에 삽입
- Delmin(S) : 집합 S의 최소 키 값을 가지는 데이터를 삭제
- Delmax(S) : 집합 S의 최대 키 값을 가지는 데이터를 삭제

지금까지 발표된 묵시 양단 우선순위 큐에 관한 대표적인 자료 구조로는 완전 이진 트리에 근거한 딥(deap), 최소-최대 힙(min-max heap), 및 대칭 최소-최대 힙(smmh : symmetric min-max heap)이 있다[1-4]. 그러나, 이들 자료 구조는 캐쉬 메모리를 효과적으로 이용하지 못하고 있다. 예를 들면, Delmin()이나 Delmax() 연산을 실행하는 동안, 각 레벨에서 두 개의 자식 노드를 접근한다. 이들 두 자식 노드가 어떤 캐쉬 블록의 일부만 점유할 경우, 그 캐쉬 블록의 나머지 부분은 이용되지 않는다. 또한, 캐쉬 메모리의 효율적인 사용은 자료 구조 성능을 개선하는 것으로 나타났다[5, 6].

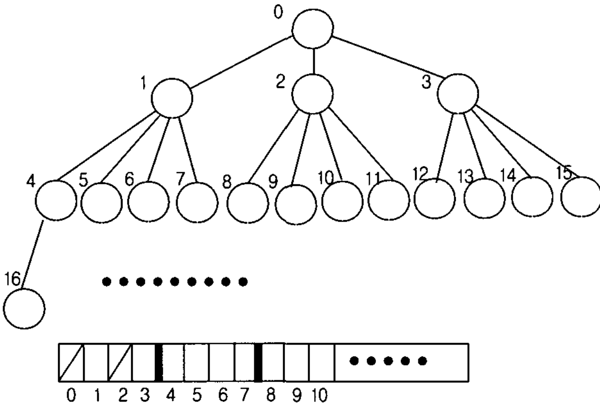
현대의 일반적인 컴퓨터 시스템의 한 캐쉬 블록의 크기는 2^i(i는 양의 정수)으로 되어 있는데, 일반적으로 32, 64, 또는 128바이트로 되어 있다. 데이터가 4바이트 크기의 두 필드 <키, 값>으로 구성되고 캐쉬 블록의 크기가 32바이트일 경우, 각 캐쉬 블록은 4개의 데이터를 수용할 수 있다. 따라서, 4-원 딥을 구성하여 모든 형제 노드들을 동일한 캐쉬 블록에 적재하는 경우, 딥 연산 동안 각 트리 레벨에서의 캐쉬 미쓰는 많아야 한번 발생한다. 또한, Delmin() 또는 Delmax() 연산 동안 캐쉬 블록의 모든 데이터를 접근하게 되므로 캐쉬 이용율은 증가한다.

본 논문에서는 양단 우선순위 큐를 구현하기 위한 4원 딥인 4-딥*를 제안한다. 4-딥*는 완전 트리 형태를 가지고, 일차원 배열에 묵시적으로 표현된다. 다음 절에서는 4-딥* 구조에 대해 살펴보고, 3장에서는 4-딥*의 연산에 대해 기술한다. 성능 실험 결과를 4장에서 보인 후, 5장에서 결론을 맺도록 한다.

† 종신회원 : 성신여자대학교 컴퓨터정보학부 교수
논문접수 : 2004년 3월 10일, 심사완료 : 2004년 11월 8일

2. 4-딥* 구조

4-딥*는 양단 우선순위 큐를 위한 자료 구조로서, 딥에서처럼 분리된 형태의 최소 및 최대 힙을 가진다. 4-딥*의 최소 및 최대 힙의 루트 노드는 각각 6개의 자식 노드를 가지고, 그 외의 모든 노드들은 4개의 자식 노드를 가진다.



(그림 1) 4-딥*를 위한 트리 구조 및 배열 표현

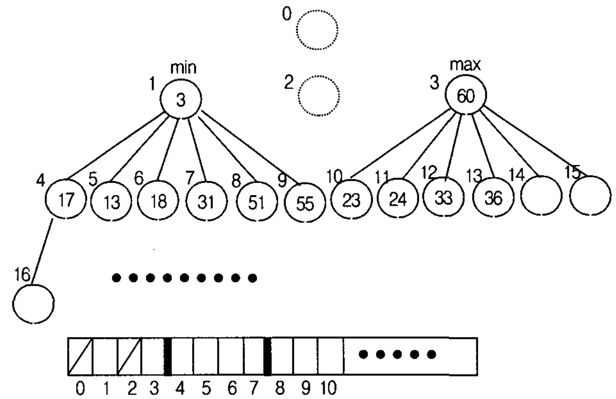
4-딥*는 (그림 1)에 나타난 루트 노드가 3개의 자식 노드를 가지고 나머지 모든 노드는 4개의 자식 노트를 가지는 트리로부터 구축된다. 이렇게 구성된 트리는 각 노드의 바깥에 표현된 노드 인덱스와 배열 인덱스에 나타난 바와 같이 일차원 배열에 묵시적으로 표현된다. 이러한 트리 구성에서 모든 형제 노드들은 하나의 캐쉬 블록에 적재될 수 있다. 즉, 노드 0~3를 하나의 캐쉬 블록에 적재하면, 그 다음의 형제 노드 4~7이 다른 하나의 캐쉬 블록에 적재되고, 계속해서 모든 형제 노드들은 캐쉬 블록 간에 걸치지 않고 하나의 캐쉬 블록에 적재 된다.

이렇게 표현된 트리의 부모-자식 관계는 노드 인덱스 i에 대해 다음의 수식으로 표현된다.

- $parent(i) = \lfloor i/4 \rfloor,\ i \geq 1$
- $children(i) = 4i, 4i+1, 4i+2, 4i+3,\ i \geq 0$(i가 0일 경우 $4i$는 무시)

(그림 1)의 트리를 양단 우선순위 큐로 만들기 위해 노드 0과 2를 무시하고, 노드 1을 최소 힙의 루트로 노드 3을 최대 힙의 루트로 한다. 또한 노드 2의 자식 노드 중 왼쪽 2개의 노드를 노드 1의 자식 노드로 연결하고, 오른쪽 2개의 노드를 노드 3의 자식 노드로 한다. 이렇게 구성된 4-딥*는 (그림 2)에 나타나 있다.

(그림 2)의 모든 형제 노드들이 동일한 캐쉬 블록에 저장될 경우, 딥 연산 동안 각 레벨에서 최대 한번의 캐쉬 미쓰가 발생한다. 단, 최소 및 최대 힙의 루트 노드는 6개의 자식 노드를 가지므로 두 번의 캐쉬 미쓰가 발생될 수 있다.



(그림 2) 4-딥* 및 배열 표현

제안된 4-딥*는 다음의 성질을 가진다.

- 노드 0과 2는 사용되지 않는다.
- 노드 1과 3은 각각 최소 힙과 최대 힙의 루트 노드이다.
- 최소 힙의 노드는 최대 힙에 대응하는 노드를 가지고, 최대 힙의 노드 또한 최소 힙에 대응하는 노드를 가진다. 노드 i를 최소 힙의 어떤 노드라 하고, 노드 j를 최대 힙에 속한 대응 노드(corresponding node)라 할 때, 노드 i의 키는 노드 j의 키보다 작거나 같다.
- 4-딥*는 규칙적인 구조를 가진다. 즉, 트리의 각 레벨의 가장 왼쪽 노드는 4의 멱승의 인덱스 값을 가진다.

4-딥*의 규칙성에 의해 최소 힙과 최대 힙의 대응 관계는 다음과 같이 계산된다. 여기서, n은 4-딥*에 있는 데이터의 수이고, 함수 $f(i) = 4^{\lfloor \log_4 i \rfloor}$이다. 단, i는 2가 아닌 1이상의 정수이다.

- i의 최대 대응 노드 j 계산

  $j = i + \lfloor 3^* f(i)/2 \rfloor$;

  if $(j \equiv 2)$ $j = j+1$;

  if $(j > n)$ $j = \lfloor j/4 \rfloor$;

- j의 최소 대응 노드 i 계산

  $i = j - \lfloor 3^* f(j)/2 \rfloor$;

  if $(i \equiv 2)$ $i = i-1$;

3. 4-딥* 연산

양단 우선순위 큐는 서론에서 기술한 세 가지 연산을 지원한다. 4-딥*에서의 연산 알고리즘을 설명하기 위해 4-딥*는 최대 m개의 데이터를 저장하기 위한 배열 A[m+2]에 묵시적으로 표현되고, 변수 last는 그 배열에 저장된 마지막 데이터의 배열 인덱스를 나타낸다고 가정한다. 배열의 마지막 인덱스 m+1을 MaxIndex라 한다.

삽입 및 삭제 연산을 위한 다음의 유틸리티 함수들을 가

정한다.

- MinPartner(j) : 최대 힙 노드 인덱스 j에 대한 최소 대응 노드 인덱스를 계산하여 리턴한다. 위의 최소 대응 노드 계산식 참조.
- MaxPartner(i) : 최소 힙 노드 인덱스 i에 대한 최대 대응 노드 인덱스를 계산하여 리턴한다. 위의 최대 대응 노드 계산식 참조.
- MaxHeap(i) : 인덱스 i가 최대 힙에 속하면 참을 아니면 거짓을 리턴한다. 즉 i가 3이거나, $5f(i)/2 \le i \le 4f(i)-1$를 만족하면 참을 리턴한다. 여기서, $f(i) = 4^{\lfloor \log_4 i \rfloor}$이다.
- MinInsert(i, x) : 데이터 x를 최소 힙에 노드 i로부터 시작하여 상향식으로 삽입한다. 이는 전통적인 최소 힙의 삽입 방법과 동일하다.
- MaxInsert(i, x) : 데이터 x를 최대 힙에 노드 i로부터 시작하여 상향식으로 삽입한다. 이는 정통적인 최대 힙의 삽입 방법과 동일하다.

3.1 삽입 연산

(알고리즘 1)에 나타난 것처럼 임의의 데이터 삽입은 리프 노드로부터 상향식으로 이루어진다.

삽입 알고리즘에서는 우선 last와 MaxIndex를 비교하여 삽입 가능한 지를 조사하고, 가능하면 last를 증가시킨다. 4-딥*가 비어 있었을 경우 입력 데이터 x를 A[1]에 삽입하고 리턴한다. 그렇지 않고, 증가된 last가 2이면 노드 2는 사용하지 않기 때문에 1을 더 증가시킨다. 그 후, last가 최대힙에 속하면, 최소힙의 대응 노드 i에 있는 데이터 키와 x의 키 값을 비교하여, x를 삽입할 위치를 결정한다. 즉, x의 키가 작으면, 대응 노드 테이터를 last 위치로 옮기고, 빈 자리에 x를 MinInsert() 함수를 통하여 최소힙에 삽입한다. 그렇지 않으면, x를 MaxInsert() 함수를 이용하여 최대힙에 삽입한다. 노드 last가 최소 힙에 속할 경우, last의 최대 힙 대응 노드 j를 계산힌다. 여기서도, x값을 최대 대응 노드 A[j]와 비교하여 x가 최소 또는 최대 힙에 삽입되어야 할지 결정한 후, MinInsert() 또는 MaxInsert()를 이용하여 삽입한다.

```
bool Insert( Element x )
{
  if(last == MaxIndex) return FALSE ; // full
  last++ ;
  if(last == 1) { A[1] = x ; return TRUE ; }
  if(last == 2) last++ ;
  switch(MaxHeap(last)) {
    case TRUE : // last is in max heap
      i = MinPartner(last) ;
      if(A[i].key > x.key) {A[last] = A[i] ; MinInsert(i, x) ; }
      else MaxInsert(last, x) ;
      break ;
    case FALSE :
      j = MaxPartner(last) ;
      if(j == 2) j = 3 ;
      if(x.key > A[j].key) {A[last] = A[j] ; MaxInsert(j, x) ; }
      else MinInsert(last, x) ;
  } // switch
  return TRUE ;
}
```

(알고리즘 1) 삽입 알고리즘

알고리즘 올바름(correctness) : 삽입 알고리즘은 리프 노드에서 대응 노드를 계산하여 작은 값을 최소 힙에 큰 값을 최대 힙에 삽입한다. 삽입은 상향식으로 최대 또는 최소 힙 성질을 만족시키도록 이루어진다. 따라서, (알고리즘 1)에 기술된 알고리즘은 4-딥*의 성질을 만족한다.

[정리 1] 4-딥*에서 임의의 데이터 삽입은 $O(\log_4 n)$ 시간 복잡도를 가진다. 여기서, n은 데이터 수이다.

[증명] 삽입 알고리즘은 대응 관계에 있는 노드를 비교하여 새로운 데이터의 삽입 위치를 결정하고, 그에 따라 MinInsert()또는 MaxInsert()를 이용하여 최소 또는 최대 힙에 삽입하는데, 이 두 함수는 $O(\log_4 n)$ 시간 복잡도를 가진다. 따라서, 4-딥*에서의 삽입은 $O(1+\log_4 n) = O(\log_4 n)$ 시간 복잡도를 가진다. ◇

3.2 최소 데이터 삭제 연산

4-딥*로부터 가장 작은 키를 가진 데이터를 삭제하는 최소키 데이터 삭제 알고리즘은 (알고리즘 2)에 나타나 있다. 4-딥*에 하나의 데이터만이 있을 경우 그 데이터를 삭제하여 리턴한다. 데이터가 두개 있을 경우, 최소 데이터 A[1]을 삭제하고, 최대 데이터 A[3]를 삭제된 위치로 이동시킨다. 세 개 이상의 데이터가 있을 경우, 최소키를 가진 데이터를 x에 저장하고, last가 가리키는 데이터를 tmp에 임시 저장한 후, last를 감소시킨다. tmp를 4-딥*에 재삽입하기 위해 최소 힙의 루트로부터 시작하여 리프 노드 쪽으로 내려가면서 삽입 위치를 찾게 된다. 내려가는 도중 위치를 찾게 되면 그 위치에 tmp를 저장하고 x를 리턴한다. 그렇지 않으면, 변수 i는 최소 힙의 리프 노드에 도달하게 된다. 리프 노드에서 i에 대한 최대 대응 노드 j를 MaxPartner() 함수를 이용하여 찾아, A[j]를 tmp의 값과 비교하여 tmp의 값이 작을 경우 tmp를 A[i]에 저장하고 리턴한다. tmp가 클 경우에는 A[j]의 값을 A[i]로 이동시키고, tmp를 최대 힙에 MaxInsert() 함수를 이용하여 삽입한 후, 저장된 x를 리턴하고 종료한다.

```
Element Delmin( void )
{
 if(last == 0) return NULL ; // empty
 if(last == 1) { x = A[1] ; last = 0 ; return x ; }
 if(last == 3) { x = A[1] ; A[1] = A[3] ; last = 1 ; return x ; }
 x = A[1] ; tmp = A[last] ;
 last-- ;
 i = 1 ; j = 4 ; // j : i의 가장 왼쪽 자식 노드 인덱스
```

```
  while( j <= last ) {
    j = i의 자식 노드 중 가장 작은 키를 가진 노드 인덱스 ;
    if(tmp.key <= A[j].key) { A[i] = tmp ; return x ; }
    i = j ; j = 4*j ;
  } // while
  // i : 최소 힙의 리프 노드.
  j = MaxPartner(i) ;
  if( tmp.key <= A[j].key ) { A[i] = tmp ; return x ; }
  A[i] = A[j] ;
  MaxInsert(j, tmp) ;
  return x ;
}
```

(알고리즘 2) Delmin() 알고리즘

알고리즘 올바름 : 위의 알고리즘에서 본 것처럼, 최소키 데이터를 삭제한 후, 제일 마지막 데이터 A[last]를 재삽입하기 위해 최소 힙의 루트로부터 하향하면서 삽입 위치를 찾기 시작한다. 이때, 자식 노드 중 가장 작은 데이터를 위로 올림으로서 최소 힙의 성질을 만족시킨다. 리프 노드에 도달하는 경우, 대응 노드를 찾아 재삽입 데이터 tmp와 비교한다. 재삽입 데이터가 작으면 최소 힙에 저장되고, 그렇지 않으면 대응 노드 데이터가 최소 힙으로 이동되고 재삽입 데이터가 최대 힙에 삽입된다. 따라서, 대응 관계 및 최대 힙 성질이 충족되어 4-딥*의 성질을 만족한다.

[정리 2] 4-딥*에서의 최소키를 가진 데이터 삭제는 $O(\log_4 n)$ 시간 복잡도를 가진다. 여기서 n은 데이터 수이다.

[증명] A[last]의 재삽입을 위해 최소 힙의 루트로부터 시작하여 리프 노드까지 $O(\log_4 n)$ 시간, 대응 노드를 찾는데 상수 시간, 및 대응 노드로부터 상향식으로 최대 힙의 루트까지 $O(\log_4 n)$ 시간이 걸린다. 따라서, 최소 데이터 삭제 알고리즘은 $O(\log_4 n)$ 시간 복잡도를 가진다. ◆

3.3 최대 데이터 삭제 연산

최대 데이터 삭제의 경우, last가 최대 힙에 속할 때 노드 last와 노드 last의 대응 노드의 오른쪽 노드 (RS 노드) 중 최대 키를 가진 데이터가 재삽입되어야 한다. 이것은 RS 노드의 최대 힙에 있는 대응 노드가 k개의 자식 노드를 가지기 때문이다. 여기서, k는 최대 힙에 있는 대응 노드가 노드 3이면 $1 < k < 6$이고, 그렇지 않으면 $1 < k < 4$이다. 예를 들면, (그림 2)에서 last가 13일 경우, 그 대응 노드는 7이 된다. 노드 7의 오른쪽 노드 중 가장 큰 키를 가진 노드는 노드 9가 되고 노드 9의 키 55는 노드 last의 키보다 크므로 노드 9의 데이터가 재삽입되어야 한다.

(알고리즘 3)는 Delmax() 연산에 대한 알고리즘을 나타낸다. 4-딥*에 데이터가 한 개 있을 경우, 그 데이터 A[1]이 삭제된다. 그렇지 않으면, 최대 데이터 A[3]를 삭제한다. 그 후, A[last]가 재삽입을 위해 tmp에 임시 저장된다. 그러나, 노드 last가 최대 힙에 속할 경우, RS 노드 중 하나가 최대 키를 가질 수 있으므로, last의 대응 노드가 계산되고, 노드 last와 RS 노드 중 최대 키를 갖는 데이터가 추출되어 재삽입을 위해 tmp에 저장된다. 재삽입 데이터 선택 후, last를 감소시키고, 최대 힙의 루트로부터 내려가면서 tmp의 삽입 위치를 결정한다. 내려가는 도중 tmp 값이 자식 노드보다 클 경우 그 위치에 삽입을 하고, 그렇지 않으면 i는 리프 노드에 도달하게 된다. 리프 노드에서 i의 대응 노드 인덱스 j가 계산되고, A[j]보다 tmp가 클 경우 tmp는 i가 가리키는 리프 노드에 저장된다. 그렇지 않으면, A[j]를 A[i]로 이동시키고, tmp를 최소 힙에 MinInsert()를 호출하여 삽입한다.

```
Element Delmax(void)
{
  if( last == 0 ) return NULL ; // empty
  if( last == 1 ) { x = A[1] ; n=0 ; return x ; }
  if( last == 3 ) { x = A[3] ; n=1 ; return x ; }
  x = A[3] ; // delete the max element in node 3
  tmp = A[last] ;
  if( MaxHeap(last) ) {
    i = MinPartner( last ) ;
    m = 노드 i의 오른쪽 형제 노드 중 가장 큰 키를 갖는 노드 ;
    if( tmp.key < A[m].key ) swap( tmp, A[m] ) ;
  } // end if
  last-- ;
  i = 3 ; j = 10 ; // j is the left-most child of i
  while( j <= last ) {
      j = i의 자식 노드 중 가장 큰 키를 가진 노드 인덱스 ;
      if( tmp.key >= A[j].key ) { A[i] = tmp; return x ; } ;
      A[i] = A[j] ; i = j ; j=4*j ;
  } // while
  // i : max heap의 리프 노드를 가리킴.
  j = i의 대응 노드 중 가장 큰 키를 가지는 노드 인덱스 ;
  if( A[j].key <= tmp.key ) { A[i] = tmp ; return x ; }
  A[i] = A[j] ; MinInsert(j, tmp) ; // 최소 힙 삽입
  return x ;
}
```

(알고리즘 3) Delmax() 알고리즘

알고리즘 올바름 : (알고리즘 3)에서 재삽입 데이터는 A[last]와 그의 RS 노드들 중 최대 값이 선정되므로, 4-딥*의 대응 관계를 만족시킨다. 또한, 재삽입시 최대 힙의 루트로부터 내려가면서 최대 힙의 성질이 만족되도록 삽입이 이루어지고, 리프 노드에 도달하는 경우 대응 노드와 비교하여 리프 노드 대응 성질이 성립하도록 한다. 재삽입 데이터가 최소 힙에 삽입될 경우에서 최소 힙 성질을 만족시킴으로서, 4-딥*의 성질을 만족한다.

[정리 3] 4-딥*에서 최대 데이터 삭제는 $O(\log_4 n)$ 시 복잡도를 가진다. 여기서 n은 데이터 수 이다.

[증명] Delmax() 알고리즘에서 재삽입을 위해 최대 힙에서의 하향 단계 및 최소 힙에서의 상향 단계 각각이 $O(\log_4 n)$ 시간 걸리고, 대응 노드 비교는 상수 시간이 걸리므로, 전체 시간 복잡도는 $O(2\log_4 n + 1) = O(\log_4 n)$이 된다. ◆

3.4 상세 성능 비교 분석

지금까지 4-딥*의 3가지 연산에 대한 알고리즘을 살펴보았다. 본 절에서는 성능에 중요한 영향을 주는 캐쉬 메모리 이용 효율성 및 키 비교 회수에 관한 세부 분석을 하여, 딥과 비교한다.

[정리 4 : 캐쉬 이용률] 4-딥*는 최악의 경우 딥보다 1/2의 캐쉬 미쓰를 발생시킨다.

[증명] 연산 동안 각 자료 구조는 트리 높이 h개의 캐쉬 블록을 접근한다. 따라서, 딥은 최악의 경우 $\log_2 n$개의 캐쉬 블록을 접근하고, 4-딥*는 $\log_4 n$개의 캐쉬 블록을 접근한다. 그러므로, 최악의 경우 캐쉬 미쓰 비율은 $\log_4 n / \log_2 n = 1/2$이 된다. ◆

[정리 5 : 키 비교 회수] 삽입 연산 동안 4-딥*는 딥보다 약 1/2만큼 비교한다. 최대 및 최소 데이터 삭제 연산 동안 4-딥*는 딥보다 약 5/6만큼 비교한다.

[증명] 삽입은 대응 노드와 입력 x의 비교 후, 최대 또는 최소 힙에 삽입을 한다. 상향식 삽입 시, 트리 각 레벨마다 한번의 비교를 하므로, 높이가 h인 트리에서 총 비교 회수는 h+1이 된다. 따라서, 4-딥*는 ($\log_4 n+1$)번, 딥은 ($\log_2 n+1$)번 비교를 하므로, 4-딥*는 딥보다 약 $(\log_4 n+1)/(\log_2 n+1) \simeq 1/2$을 비교한다.

다음은 삭제 연산에 대해 살펴보자. 최소 데이터 삭제 동안 비교 회수는 하향 단계에서 각 레벨당 4번의 비교를 필요로 하고, 대응 노드와의 1번 비교, 상향 단계에서의 각 레벨당 1번의 비교가 필요하므로, 약 $(5^* \log_4 n+1)$번의 비교를 하게 된다. 딥의 경우에는 하향 단계에서 각 레벨 당 2번의 비교를 하므로 $(3\log_2 n+1)$번의 비교를 한다. 따라서, 4-딥*는 딥보다 $(5\log_4 n+1)/(3\log_2 n+1) \simeq$ 5/6만큼 비교한다. 최대 데이터 삭제 동안 데이터 비교 회수를 보면, 4-딥*의 경우 재삽입 데이터 선정 시 5번, 최대 힙에서의 하향 중 각 레벨에서 4회, 대응 노드와의 1회, 최소 힙에서의 상향 중 각 레벨에서 1회 비교가 요구되므로, 총 비교 회수는 $5^*\log_4 n+6$이 된다. 딥의 경우 $(3\log_2 n+1)$번 비교가 요구되므로, 4-딥*는 딥보다 $(5\log_4 n+1)/(3\log_2 n+1) \simeq$ 5/6만큼 비교하게 된다. ◆

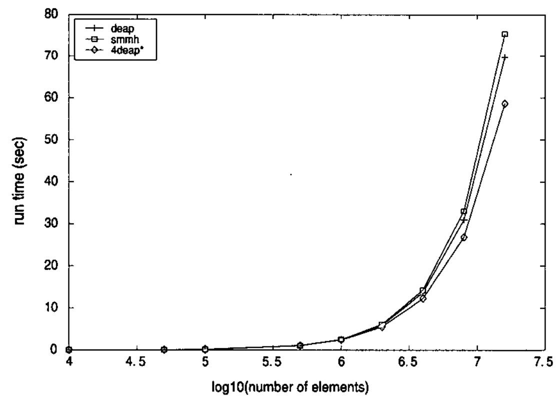
위의 분석 결과 4-딥*는 딥보다 상수 배 만큼 빠른 것으로 나타났으며, 이를 증명하기 위해 다음 절에서 실험 결과를 보인다.

4. 실험 결과

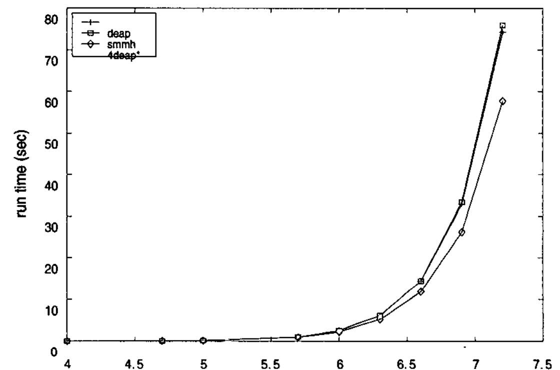
딥은 기존의 다른 양단 우선순위 큐들과 유사한 성능을 보인 것으로 나타났다[7]. 따라서, 본 논문에서는 딥, [7]에서 실험에 포함되지 않았던 smmh, 및 4-딥*를 구현하여 실험을 하였다. 이들 자료 구조를 구현함에 있어서, 4-딥*의 최소 및 최대 힙의 자식 노드를 제외한 모든 형제 노드들은 동일한 캐쉬 블록에 적재되도록 하였다. 각 노드의 크기는 8바이트로 하였고, 1기가 바이트의 주 기억 장치와 32/64바이트의 L1/L2 캐쉬 블록 크기를 가진 Ultrasparc 워크스테이션에서 실험이 이루어졌다.

실험은 n = 10,000(10K), 50K, 100K, 500K, 1M(백만), 2M, 4M, 8M, 16M개의 데이터에 대하여 다음의 시험 모델에 대해 이루어졌다.

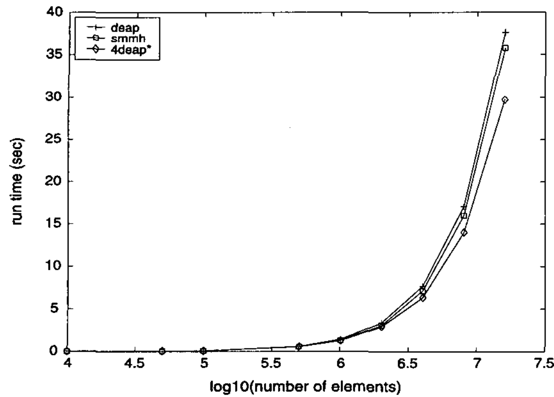
- 홀드 시험 모델[8] : n/2 무작위 수로 각 자료 구조를 초기화한 후, n개의 혼합된 50% Insert(), 25% Delmin(), 25% Delmax() 연산이 실행되었다. n개의 혼합 연산에 대한 시간이 측정되었다.
- 정렬-D 시험 모델 : n개의 무작위 수를 각 자료 구조에 삽입한 후, Delmax() 연산이 n회 실행되었다. n번의 삽입 및 n번의 삭제 연산에 걸린 시간이 측정되었다.
- 정렬-I 시험 모델 : n개의 무작위 수를 각 자료 구조에 삽입한 후, n번의 Delmin() 연산을 반복하였다. 시간 측정은 2n번의 삽입 및 삭제 연산에 대해 이루어졌다.



(그림 3) 정렬-D 시험 모델에서의 실행 시간



(그림 4) 정렬-I 시험 모델에서의 실행 시간



(그림 5) 홀드 시험 모델에서의 실행 시간

(그림 3)~(그림 5)에 나타난 측정·시간은 15회 실행의 평균이다. 모든 시험 모델에 대해 4-딥*(4-deap*로 표기)가 가장 빠르고, 딥과 smmh는 유사한 성능을 보이고 있다. 그림에서 보는 바와 같이, 4-딥*는 기존의 딥보다 20~30% 실행 속도가 빠른 것으로 나타났다.

5. 결 론

본 논문에서는 양단 우선순위 큐인 4-딥*를 제안하였다. 제안된 자료 구조는 최대 및 최소 힙의 루트 노드를 제외한 모든 노드는 4개의 자식을 가지고, 두 루트 노드는 6개의 자식 노드를 가진다.

4-딥*는 양단 우선순위 큐 연산 동안 캐쉬 미쓰 수를 최악의 경우 기존의 이진 트리에 근거한 딥보다 반으로 줄이고, 형제 노드들을 캐쉬 블록에 정합하여 캐쉬 메모리를 효율적으로 이용한다. 실험 결과 기존의 자료 구조 보다 약 20~30% 빠른 것으로 나타났다.

참 고 문 헌

[1] S. Carlsson, "The deap - a double-ended heap to implement double-ended priority queues," Information Processing Letters, 26, pp.33-36, 1987.

[2] M. D. Atkinson, J.-R. Sack, T. Strothette, "Min-Max heaps and generalized priority queues," Comm. ACM, 29, pp.996-1000, 1986.

[3] A. Arvind, C. P. Rangan, "Symmetric Min-Max heap : A Simpler data structure for double-ended priority queue," Information Processing Letters, 69, pp.197-199, 1999.

[4] D. Mehta, S. Sahni (editors), Handbook of data structures and applications, Chapman-Hall/CRC press, 2004.

[5] H. Jung, S. Sahni, "Supernode binary search trees," International Journal of Foundations of Computer Science, Vol. 14, No.3, pp.465-490, 2003.

[6] 정해재, "8-힢* : 빠른 8-원 묵시 우선순위 큐", 정보처리학회논문지A, Vol.11-A, No.3, pp.213-216, Jun., 2004.

[7] S. Cho, S. Sahni, "Mergeable double-ended priority queues," International Journal of Foundations of Computer Science, Vol.10, No.1, pp.1-17, 1999.

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

정 해 재

e-mail : hjjung@cs.sungshin.ac.kr

1984년 경북대학교 전자공학과(전산전공)(공학사)

1987년 서울대학교 컴퓨터공학과(공학석사)

2000년 University of Florida 컴퓨터정보학과(공학박사)

1988년 1995년 한국전자통신연구원 선임연구원

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

2002년~2003년 서울대학교 컴퓨터신기술연구소 객원연구원

2003년~현재 성신여자대학교 컴퓨터정보학부 초빙교수

관심분야 : 컴퓨터 알고리즘 및 자료 구조, 계산 기하, 컴퓨터 비젼