해달 자료구조

5차시 - 우선순위 큐

목차

Part 1

우선순위 큐

개념

우선순위 큐(Priority Queue)

본래 큐는 FIFO(선입선출)
그러나 우선순위 큐에서는 각 데이터가 우선순위를 가지고,
우선순위가 높은 데이터가 먼저 나감

우선순위 큐는
최대 우선순위 큐와 **최소 우선순위 큐**로 구분할 수 있음

최대 우선순위 큐 – 우선순위가 가장 높은 요소가 먼저 삭제됨
최소 우선순위 큐 – 우선순위가 가장 낮은 요소가 먼저 삭제됨

ADT

우선순위 큐(Priority Queue) 추상자료형

- 객체: n개의 element형의 우선 순위를 가진 요소들의 모임
- 연산:
  - create() ::= 우선순위 큐를 생성한다.
  - init(q) ::= 우선순위 큐 q를 초기화한다.
  - is_empty(q) ::= 우선순위 큐 q가 비어있는지를 검사한다.
  - is_full(q) ::= 우선순위 큐 q가 가득 찼는가를 검사한다.
  - insert(q, x) ::= 우선순위 큐 q에 요소 x를 추가한다.
  - delete(q) ::= 우선순위 큐로부터 가장 우선순위가 높은 요소를 삭제하고 이 요소를 반환한다.
  - find(q) ::= 우선순위가 가장 높은 요소를 반환한다.

구현 방법

우선순위 큐를 구현하는 방법에는 배열, 연결 리스트, 히프를 이용하는 방법이 있음

이 중 가장 효율적인 방법은 히프(heap)

구현 방법 - 배열

1) 정렬 되어 있지 않은 배열
   삽입 - 배열의 맨 끝에 새로운 요소를 추가하면 됨
   삭제 - 우선순위가 높은 요소를 찾아야하므로 처음부터 끝까지 탐색 후 삭제
   또한 삭제 후 요소들의 위치를 앞으로 이동시킬 필요가 있음

2) 정렬 되어 있는 배열
   삽입 - 다른 요소와 값을 비교하여 삽입 위치 결정.
   그 후 삽입 위치 뒤에 있는 요소들을 이동시켜 빈 자리를 만들고 삽입
   삭제 - 맨 뒤에 위치한 요소를 삭제하면 됨

구현 방법 - 연결 리스트

1) 정렬 되어 있지 않은 연결 리스트

삽입 - 첫 번째 요소로 노드를 삽입

그리고 배열과 달리 맨 앞에 삽입하기 위해 다른 노드를 이동시킬 필요가 없음.

삭제 - 우선순위가 높은 요소를 찾아야하므로 처음부터 끝까지 탐색 후 삭제

2) 정렬 되어 있는 연결 리스트

삽입 - 우선순위가 높은 요소가 앞에 위치하도록 하는 것이 유리

따라서 우선순위 값을 기준으로 삽입 위치를 찾아야 함

삭제 - 첫 번째 노드를 삭제하면 됨

Part 2

힙(heap)

개념

히프는 최대값이나 최소값을 빠르게 찾아내도록 만들어진 자료구조

간단히 말하면, 부모 노드의 키 값이 항상 자식 노드의 키 값보다 큰 이진 트리

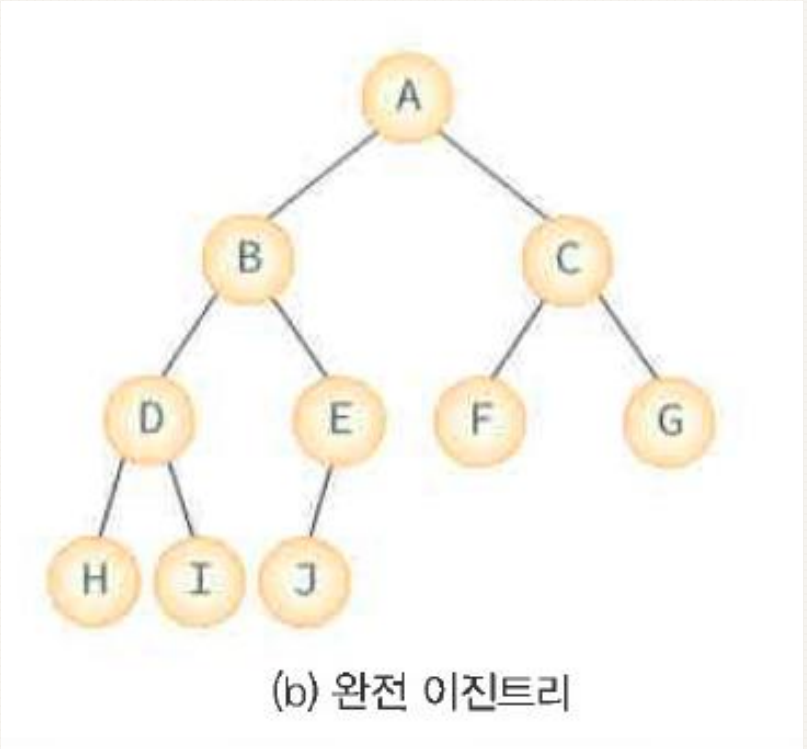
REMIND - 이진 트리

이진 트리: 모든 노드가 2개의 서브 트리를 가지는 트리

완전 이진 트리(complete binary tree)

높이가 k일 때,
레벨 1부터 k-1까지는 노드가 모두 채워져 있고
레벨 k에서는 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진 트리

레벨 k에서는 노드가 모두 채워져 있지 않아도 되지만 중간에 빈 곳이 있으면 안됨



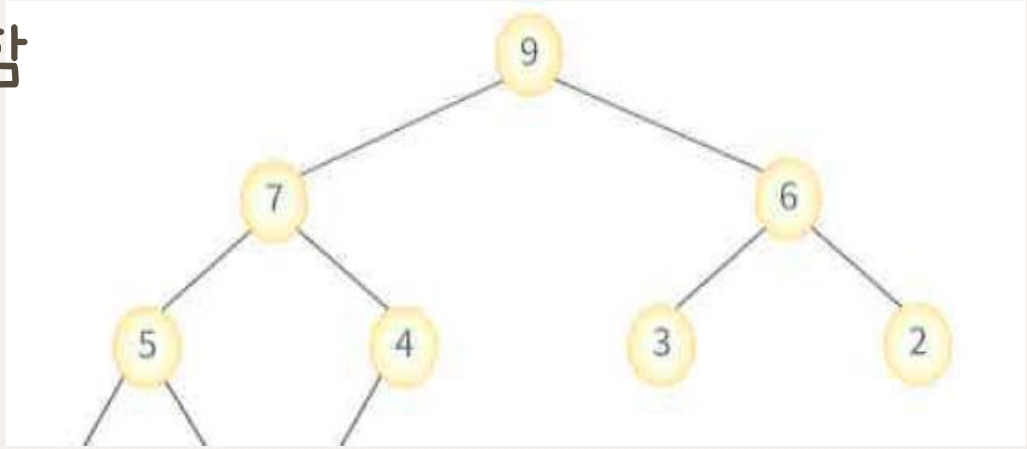
(b) 완전 이진트리

개념

히프는 최대값이나 최소값을 빠르게 찾아내도록 만들어진 자료구조

간단히 말하면, 부모 노드의 키 값이 항상 자식 노드의 키 값보다 큰 이진 트리

히프는 완전 이진 트리. 또한 중복된 값을 허용함



히프의 데이터들은 느슨한 정렬 상태를 유지함

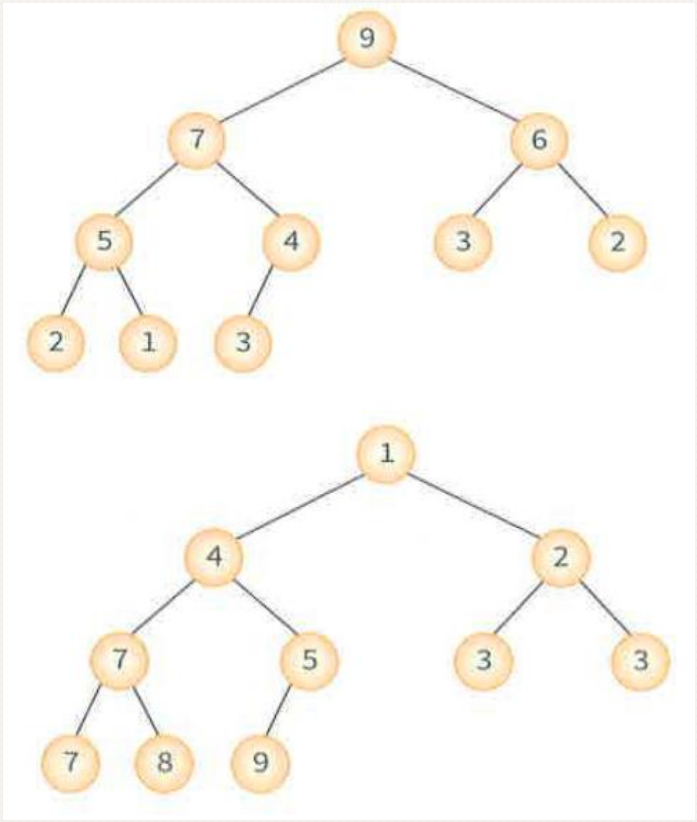
전체가 정렬되어 있지 않고
큰 값이 상위 레벨에, 작은 값이 하위 레벨에 있음

종류

우선순위 큐가 최대 우선순위 큐와 최소 우선순위 큐로 구분되는 것처럼
히프도 최대 히프와 최소 히프로 구분됨

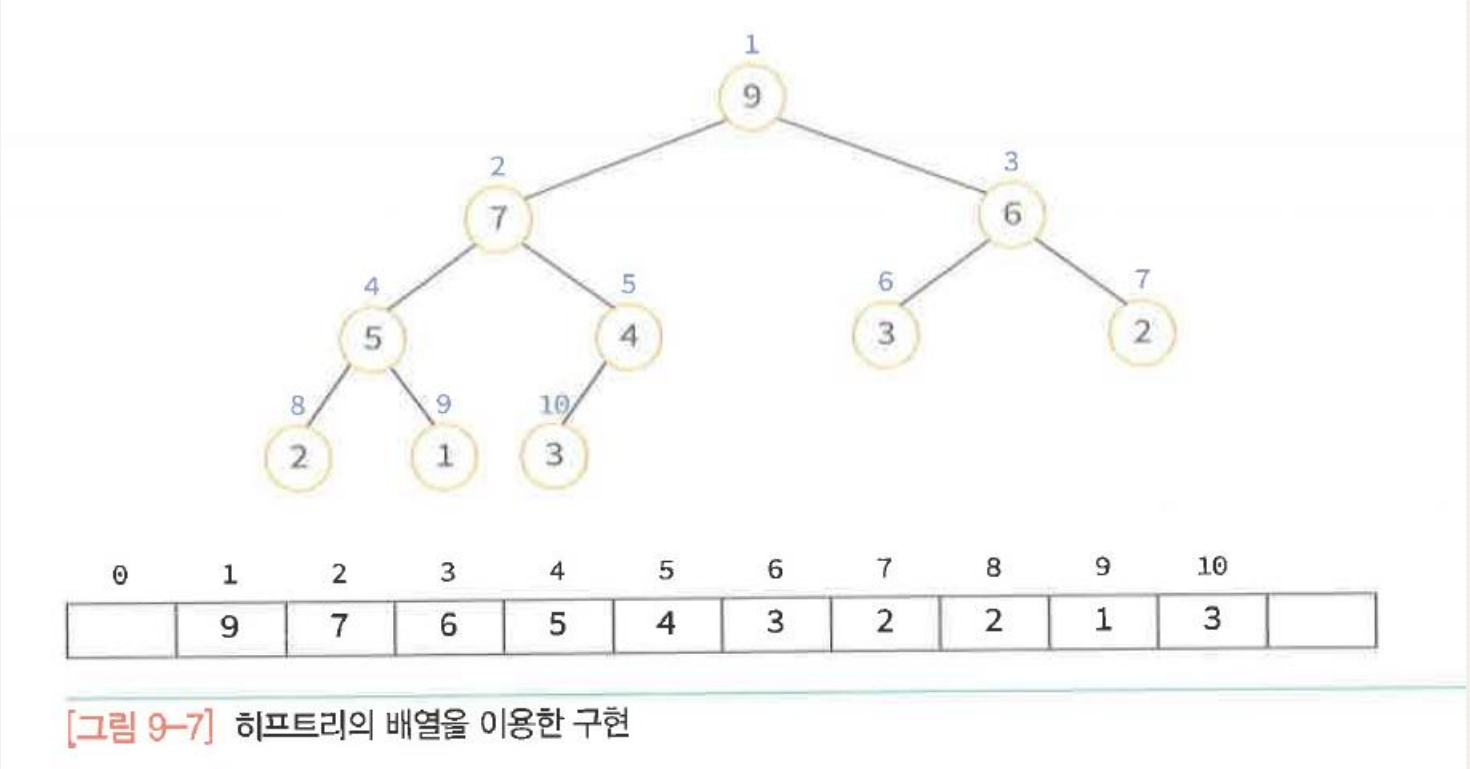
최대 히프(max heap):
부모 노드의 키값이 자식 노드의 키값보다 크거나 같음

최소 히프(min heap):
부모 노드의 키값이 자식 노드의 키값보다 작거나 같음



구현

히프는 완전 이진 트리이기 때문에 각각의 노드에 차례대로 번호를 붙일 수 있음
이 번호를 배열의 인덱스로 생각하면 배열에 히프의 노드들을 저장하는 것이 가능
따라서 히프를 저장하는 표준적인 자료구조는 배열



[그림 9-7] 히프트리의 배열을 이용한 구현

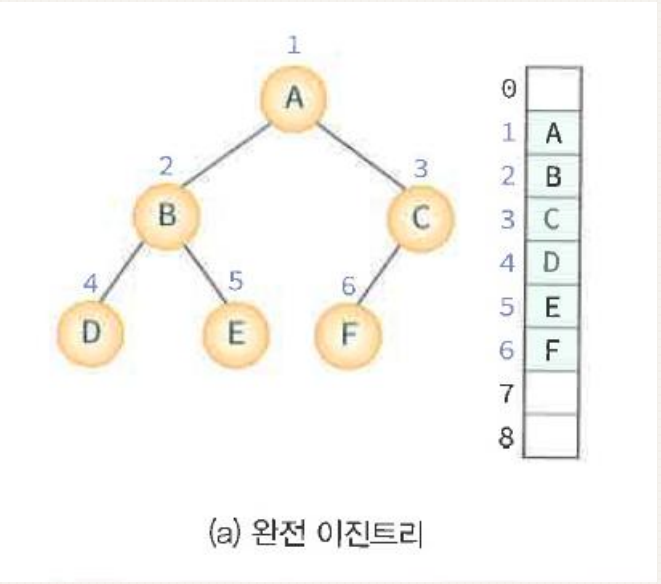
REMIND – 이진 트리의 표현(배열 이용)

배열을 이용하여 이진 트리를 표현하는 방법은 주로 포화 이진 트리나 완전 이진 트리의 경우 많이 쓰이는 방법

높이가 k면 $2^k - 1$개의 공간을 할당, 번호대로 노드들을 저장

부모와 자식의 인덱스 사이에는 다음과 같은 공식이 성립함

- 부모 노드 인덱스 = (자식의 인덱스) / 2
- 왼쪽 자식 노드 인덱스 = (부모의 인덱스) * 2
- 오른쪽 자식 노드 인덱스 = (부모의 인덱스) * 2 + 1



(a) 완전 이진트리

구현 - 정의

```
#define MAX_ELEMENT 200
typedef struct {
    int key;
} element;
typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;
```

히프의 각 요소를 구조체 element로 정의

그리고 element의 1차원 배열을 만들어
히프를 구현함

여기서 heap_size는
현재 히프 안에 저장된 요소의 개수

구현 - 삽입

히프에 새로운 요소가 들어오면

1) 새로운 노드를 히프의 마지막 노드로 삽입하고

2) 부모 노드의 값과 비교

3) 부모 노드의 값보다 새로운 노드의 값이 더 크면 교환

```
// 현재 요소의 개수가 heap_size인 히프 h에 item을 삽입한다.
// 삽입 함수
void insert_max_heap(HeapType* h, element item)
{
    int i;
    i = ++(h->heap_size);

    // 트리를 거슬러 올라가면서 부모 노드와 비교하는 과정
    while ((i != 1) && (item.key > h->heap[i / 2].key)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item;  // 새로운 노드를 삽입
}
```

구현 - 삭제

히프의 삭제는 다음과 같은 과정으로 이루어짐

1) 루트 노드를 삭제하고

2) 루트 노드의 자리에 마지막 노드를 가져옴

3) 자식 노드의 값을 비교해서 자식 노드의 값이 더 크면 교환
   자식 노드 중에서 더 큰 값과 교환함

```
// 삭제 함수
element delete_max_heap(HeapType* h)
{
    int parent, child;
    element item, temp;

    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
    while (child <= h->heap_size) {
        // 현재 노드의 자식노드 중 더 큰 자식노드를 찾는다.
        if ((child < h->heap_size) &&
            (h->heap[child].key) < h->heap[child + 1].key)
            child++;
        if (temp.key >= h->heap[child].key) break;
        // 한 단계 아래로 이동
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2;
    }
    h->heap[parent] = temp;
    return item;
}
```

Part 3

히프 정렬

히프 정렬

히프는 우선순위가 높은 데이터가 먼저 나감

이 우선순위를 값의 크기라고 생각하면, 정렬이 가능함

최대 히프를 사용하면 값이 가장 큰 것부터 나올 것이므로 내림차순 정렬이 되고
최소 히프를 사용하면 값이 가장 작은 것부터 나올 것이므로 오름차순 정렬이 됨

이렇게 히프를 사용하는 정렬 알고리즘을 히프 정렬(heap sort)이라고 함