8장: 우선순위큐

- 목차
  - 1. 우선순위큐
  - 2. 우선순위큐 구현 방법
  - 3. 히프

1. 우선 순위 큐

- 우선순위큐(priority queue): 우선순위를 가진 항목들을 저장하는 큐
- FIFO 순서가 아니라 우선 순위가 높은 데이터가 먼저 나가게 된다.
- 가장 일반적인 큐: 스택이나 FIFO 큐를 우선 순위 큐로 구현할 수 있다.

자료구조	*삭제되는 요소*
스택	*가장 최근에 들어온 데이터*
큐	*가장 먼저 들어온 데이터*
우선순위큐	*가장 우선순위가 높은 데이터*

- 응용분야:
  - 시뮬레이션 시스템(여기서의 우선 순위는 대개 사건의 시각이다.)
  - 네트워크 트래픽 제어
  - 운영 체제에서의 작업 스케쥴링

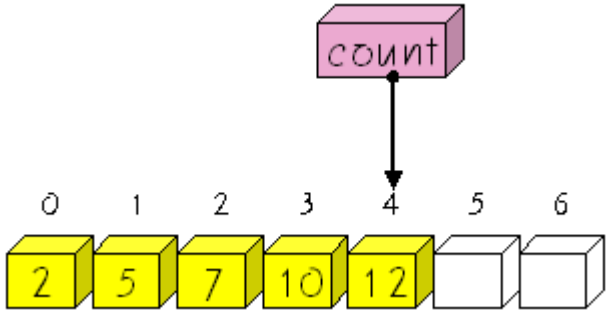
우선 순위 큐

```
•객체: n개의 element형의 우선 순위를 가진 요소들의 모임
•연산:
 ▪ create() ::=      우선 순위큐를 생성한다.
 ▪ init(q) ::= 우선 순위큐 q를 초기화한다.
 ▪ is_empty(q) ::= 우선 순위큐 q가 비어있는지를 검사한다.
 ▪ is_full(q) ::= 우선 순위큐 q가 가득 찼는가를 검사한다.
 ▪ insert(q, x) ::= 우선 순위큐 q에 요소 x를 추가한다.
 ▪ delete(q) ::= 우선 순위큐로부터 가장 우선순위가 높은 요소를 삭제하고 이 요소를 반환한
다.
 ▪ find(q) ::= 우선 순위가 가장 높은 요소를 반환한다.
```

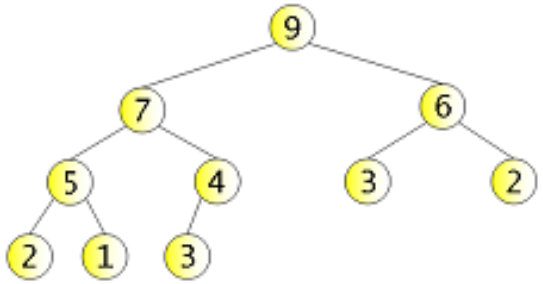
- 가장 중요한 연산은 insert 연산(요소 삽입), delete 연산(요소 삭제)이다.
- 우선 순위 큐는 2가지로 구분
  - 최소 우선 순위 큐: 가장 우선 순위가 낮은 요소 먼저 삭제
  - 최대 우선 순위 큐: 가장 우선 순위가 높은 요소 먼저 삭제

2. 우선순위큐 구현방법

- 배열을 이용한 우선 순위 큐
- 연결리스트를 이용한 우선 순위 큐
- 히프(heap)를 이용한 우선 순위 큐

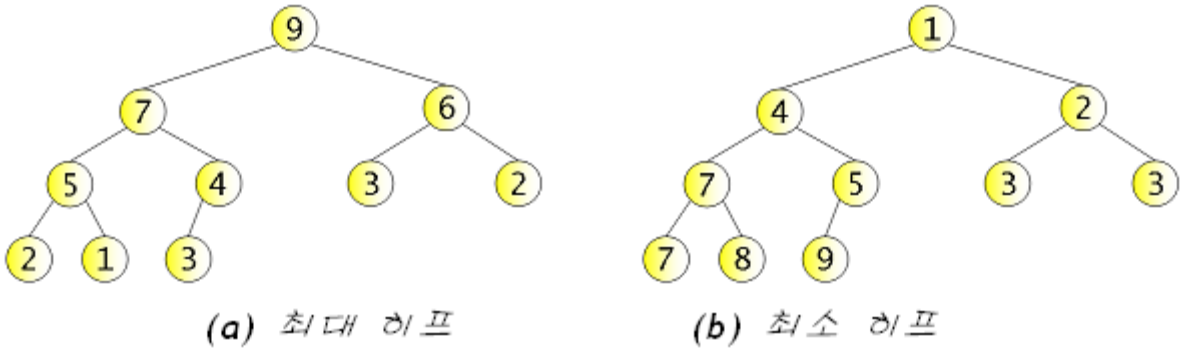


표현 방법	삽 입	삭 제
순서없는 배열	O(1)	O(n)
순서없는 연결 리스트	O(1)	O(n)
정렬된 배열	O(n)	O(1)
정렬된 연결 리스트	O(n)	O(1)
히프	O(logn)	O(logn)



3. 히프(heap)란?

- 히프란 노드들이 저장하고 있는 키들이 다음과 같은 식을 만족하는 완전 이진 트리
- 최대 히프(max heap)
  - 부모 노드의 키값이 자식 노드의 키값보다 크거나 같은 완전 이진 트리
  - key(부모노드) ≥ key(자식노드)
- 최소 히프(min heap)
  - 부모 노드의 키값이 자식 노드의 키값보다 작거나 같은 완전 이진 트리
  - key(부모노드) ≤ key(자식노드)

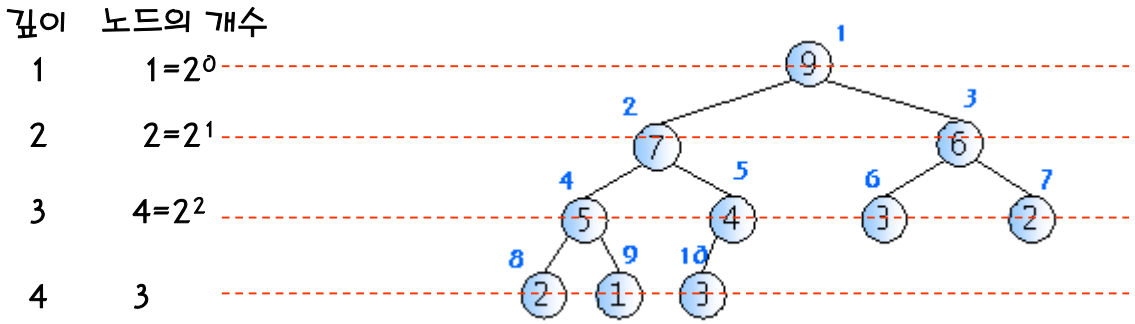


히프의 성질

- 히프는 여러 개의 값들 중에서 가장 큰 값이나 작은 값을 빠르게 찾아내도록 만들어진 자료 구조임
- 히프 트리에서는 중복 값을 허용함
- 히프 안의 데이터들은 부모 노드의 키 값이 자식 노드의 키 값보다 항상 크거나 같은(최대히프) 느슨한 정렬 상태를 유지
- 히프의 목적은 전체를 정렬할 필요 없이 삭제 연산이 수행될때 마다 가장 큰 값 또는 작은값을 찾아내는 것임

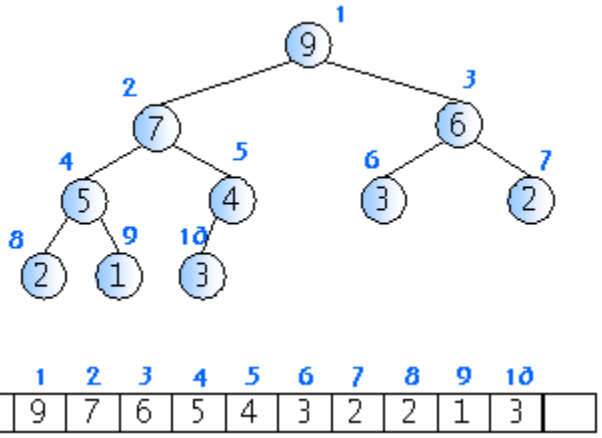
히프의 높이

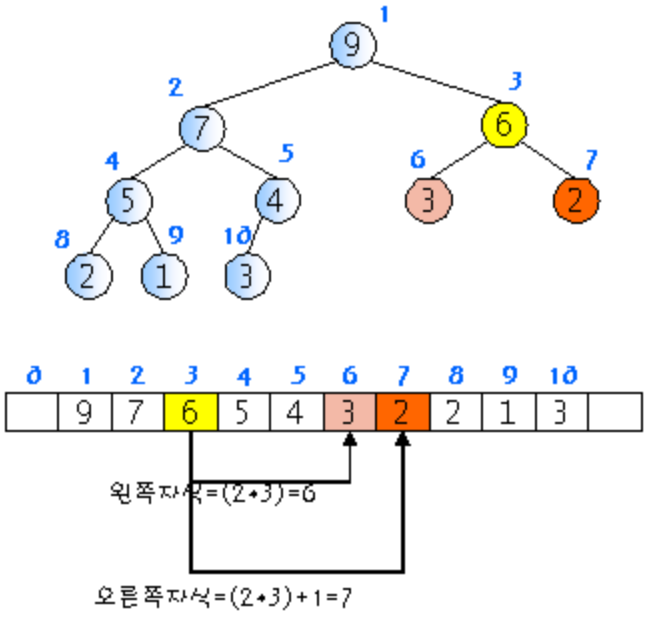
- n개의 노드를 가지고 있는 히프의 높이는 $\log_2 n$ 따라서 시간 복잡도는 $O(\log_2 n)$
- 히프는 완전이진트리
- 마지막 레벨 h을 제외하고는 각 레벨 i에 2^{i-1}개의 노드 존재



히프의 구현방법

- 히프는 배열을 이용하여 구현
  - 완전 이진 트리이므로 각 노드에 번호를 붙일 수 있다
  - 이 번호를 배열의 인덱스라고 생각
- 부모노드와 자식노드를 찾기가 쉽다.
  - 왼쪽 자식의 인덱스 = (부모의 인덱스)*2
  - 오른쪽 자식의 인덱스 = (부모의 인덱스)*2 + 1
  - 부모의 인덱스 = (자식의 인덱스)/2

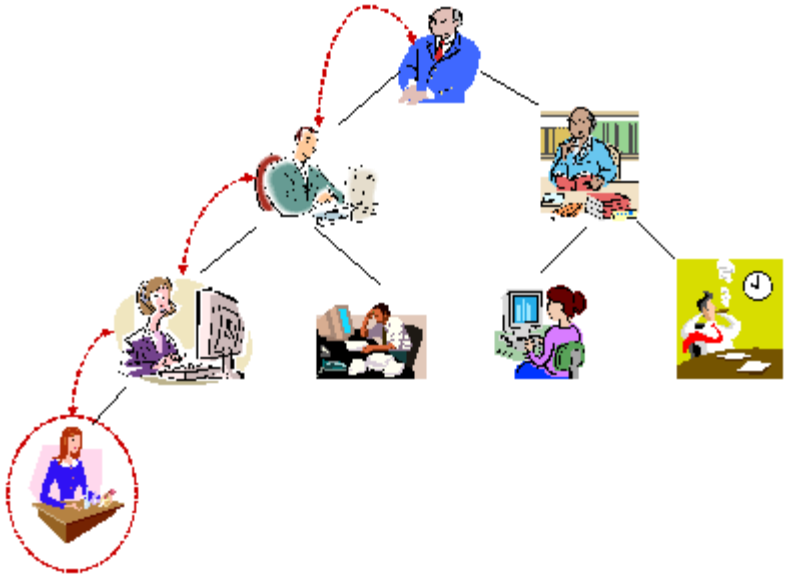




히프에서의 삽입

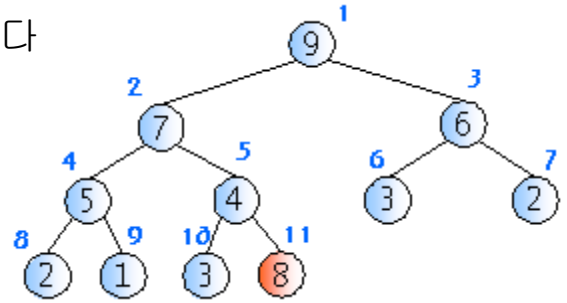
- 히프에 있어서 삽입 연산은 회사에서 신입 사원이 들어오면 일단 말단 위치에 앉힌 다음에, 신입 사원의 능력을 봐서 위로 승진시키는 것과 비슷

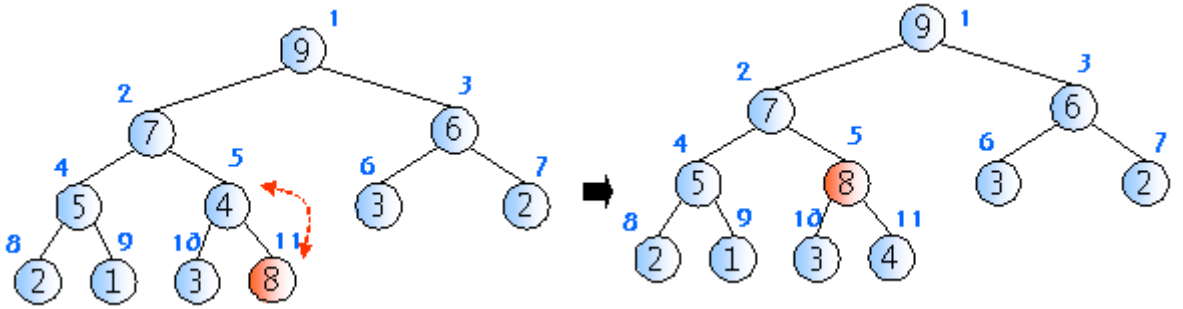
1) 히프에 새로운 요소가 들어오면, 일단 새로운 노드를 히프의 마지막 노드에 이어서 삽입
2) 삽입 후에 새로운 노드를 부모 노드들과 교환해서 히프의 성질을 만족

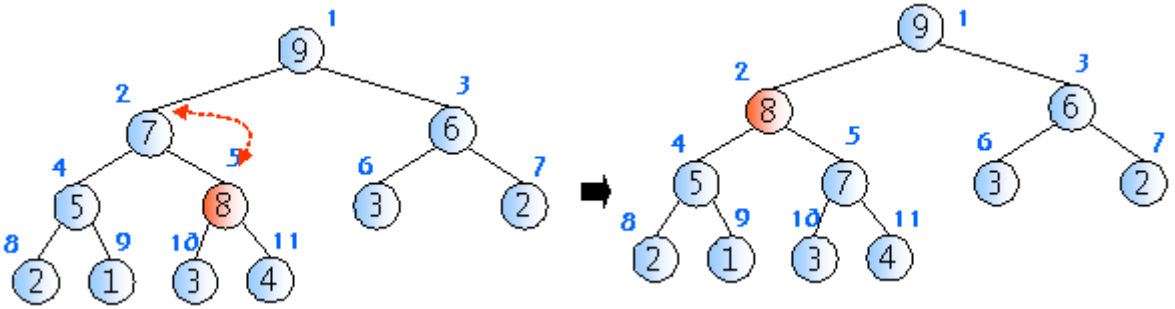


upheap 연산

- 새로운 키 k의 삽입연산후 히프의 성질이 만족되지 않을 수 있다
- upheap는 삽입된 노드로부터 루트까지의 경로에 있는 노드들을 k와 비교, 교환함으로써 히프의 성질을 복원한다.
- 키 k가 부모노드보다 작거나 같으면 upheap는 종료한다
- 히프의 높이가 O(logn)이므로 upheap연산은 O(logn)이다.







upheap 알고리즘

```
insert_max_heap(A, key)


 heap_size ← heap_size + 1; // 히프의 크기를 하나 증가
 i ← heap_size;  // 히프에 저장할 위치: 마지막 인덱스
 A[i] ← key;  // 히프에 추가
 while i ≠ 1 and A[i] > A[PARENT(i)] do // 루트노드가 아니고 부모노드보다 크면
                A[i] ↔ A[PARENT]; // i번째 노드와 부모 노드 교환
                i ← PARENT(i);   // 한단계 위로 이동
```

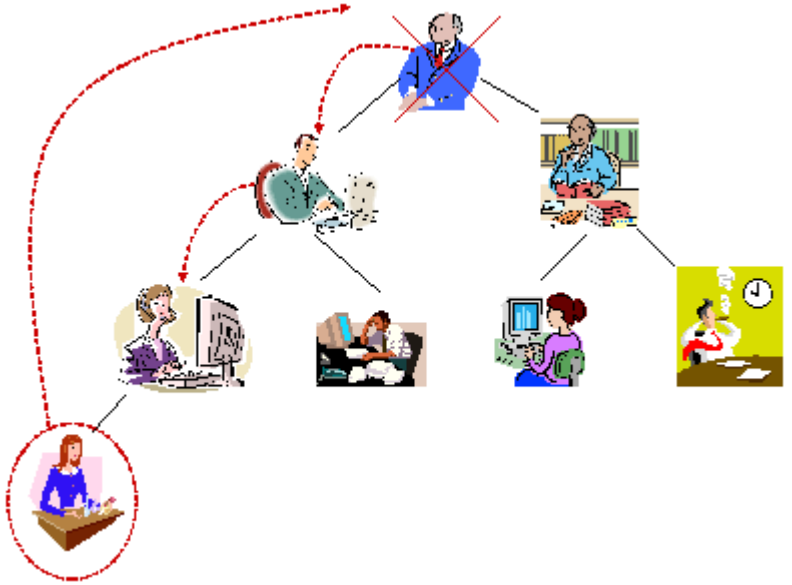
히프에서의 삭제

- 최대히프에서의 삭제는 가장 큰 키값을 가진 노드를 삭제하는 것을 의미

  -> 따라서 루트노드가 삭제된다.

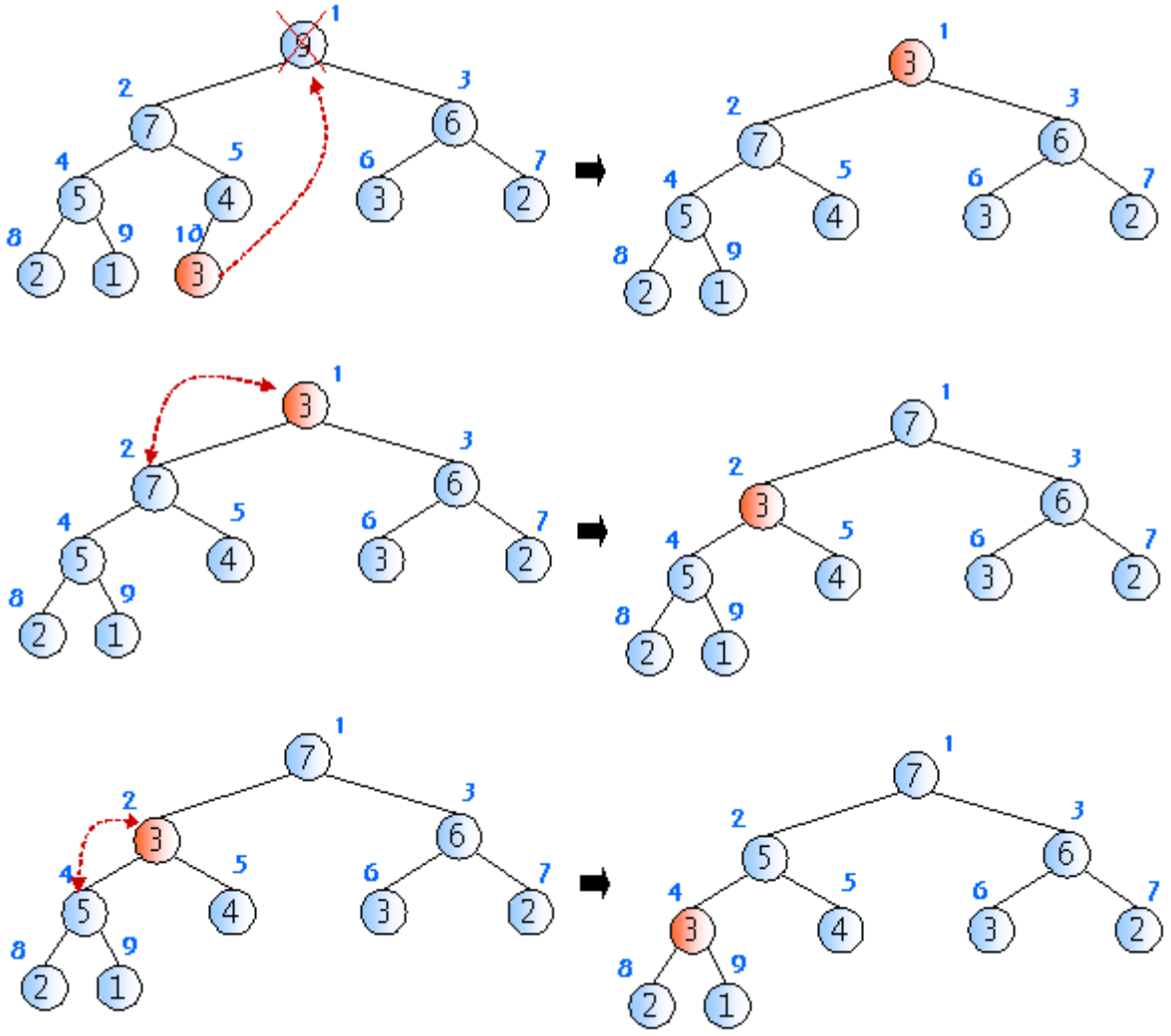
- 삭제 연산은 회사에서 사장의 자리가 비게 되면 먼저 제일 말단 사원을 사장 자리로 올린 다음에, 능력에 따라 강등시키는 것과 비슷하다.

1) 루트노드를 삭제한다
2) 마지막노드를 루트노드로 이동한다.
3) 루트에서부터 단말노드까지의 경로에 있는 노드들을 교환하여 히프 성질을 만족시킨다.



downheap 알고리즘

- 히프의 높이가 O(logn)이므로 downheap연산은 O(logn)이다.



downheap 알고리즘

```
delete_max_heap(A)

item ← A[1];  // 루트값 반환을 위해 저장
A[1] ← A[heap_size]; // 마지막 노드를 루트로 이동
heap_size←heap_size-1; // 히프의 크기를 하나 줄임
i ← 2;  //루트의 왼쪽 자식부터 비교
while i ≤ heap_size do // 히프의 크기보다 작으면 비교
        if i < heap_size and A[LEFT(i)] > A[RIGHT(i)] // 왼쪽자식이 더 클 경우
                then largest ← LEFT(i);
                else largest ← RIGHT(i);
        if A[PARENT(largest)] > A[largest] // 부모 노드가 더 클 경우 중지
                then break;
        A[PARENT(largest)] ↔ A[largest]; // 노드 교환
        i ← CHILD(largest);  // 한레벨 아래로 이동

return item;
```

히프정렬

- 히프를 이용하면 정렬 가능
- 먼저 정렬해야 할 n개의 요소들을 최대 히프에 삽입
- 한번에 하나씩 요소를 히프에서 삭제하여 저장하면 된다.
- 삭제되는 요소들은 값이 증가되는 순서(최소히프의 경우)
- 하나의 요소를 히프에 삽입하거나 삭제할 때 시간이 O(logn) 만큼 소요되고 요소의 개수가 n개이므로 전체적으로 O(nlogn)시간이 걸린다. (빠른편)
- 히프 정렬이 최대로 유용한 경우는 전체 자료를 정렬하는 것이 아니라 가장 큰 값 몇 개만 필요할 때이다.
- 이렇게 히프를 사용하는 정렬 알고리즘을 히프 정렬이라고 한다.

히프정렬 프로그램

```
// 우선 순위 큐인 히프를 이용한 정렬
void heap_sort(element a[], int n)
{
        int i;
        HeapType h;

        init(&h);
        for(i=0;i<n;i++){
                insert_max_heap(&h, a[i]);
        }
        for(i=(n-1);i>=0;i--){
                a[i] = delete_max_heap(&h);
        }
}
```

이산 이벤트 시뮬레이션

- 컴퓨터 시뮬레이션: 실재하거나 이론적으로 존재하는 물리적인 시스템의 모델을 디자인하고 그 모델을 디지털 컴퓨터에서 실행하고 그 실행 결과를 분석하는 학문
- 이산 이벤트 시뮬레이션: 모든 시간의 진행은 이벤트의 발생에 의해 이루어짐
  - 이벤트가 발생하면 시간이 진행
  - 많은 이벤트들이 발생하고 이러한 이벤트들은 시간적으로 먼저 발생된 이벤트가 먼저 처리 되어야 함
  - 따라서 우선 순위 큐를 이용하여 이벤트를 저장하고 이벤트의 발생시각을 우선순위로 이벤트를 처리

아이스크림 가게 시뮬레이션

- 아이스크림 가게에 손님들이 들어오고 나가는 과정을 시뮬레이션
- 아이스크림 가게 시뮬레이션의 고려사항은 가게 의자의 개수: 몇 개의 의자를 비치해야만 최대 이익을 발생 시킬 수 있는지를 테스트
- 아이스크림 가게의 이벤트
  - 손님이 도착하는 이벤트 (ARRIVAL)
  - 손님이 주문하는 이벤트 (ORDER)
  - 손님이 가게를 떠나는 이벤트 (LEAVE)

아이스크림 가게 시뮬레이션

- 각각의 이벤트들은 이벤트가 발생된 시각과 손님의 숫자를 가지고 있음
- 도착한 손님은 랜덤한 시간후에 주문 이벤트를 발생
- 손님은 주문을 하고 나서 랜덤한 시간 후에는 가게를 떠나는 이벤트가 발생
- 알고리즘
  - 손님 도착 이벤트: 손님의 숫자와 현재 남아 있는 의자를 비교하여 남아 있는 의자의 수가 더 많으면 손님을 받고 남아 있는 의자의 수는 손님의 수만큼 감소. 의자가 적은 경우는 손님은 주문을 하지 않고 나감
  - 주문 이벤트: 손님의 숫자대로 주문을 받고 잠시후 손님은 나감
  - 손님이 떠나는 이벤트: 떠나는 손님들의 숫자만큼 남아 있는 의자를 증가 시킴

허프만 코드(Huffman Code)

- 어떠한 문서의 각 글자의 빈도가 알려져 있는 메시지의 내용을 압축하는데 사용
- 빈도수: 영문 텍스트에서 해당 글자가 나타나는 회수
- 만약 45글자의 텍스트가 있고 아래와 같은 빈도수와 같고 한 글자를 표시하기 위해서 3비트가 필요하면,

  45 * 3 = 135 비트가 필요

글자	빈도 수
e	15
t	12
n	8
i	6
s	4

- 기본 개념: 많이 나오는 글자에 짧은 비트를 할당하고 잘 나오지 않는 글자에 긴 비트를 할당 (전체 크기 감소)

허프만 코드(Huffman Code)

- 아래의 표와 같이 비트를 할당하면

  15 * 2 + 12 * 2 + 8 * 2 + 6 * 3 + 4 * 3 = 88비트로 표현 가능

글자	비트 코드	빈도 수	비트 수
e	00	15	30
t	01	12	24
n	10	8	16
i	110	6	18
s	111	4	12
합계			88

- 위의 비트를 이용하여 텍스트를 효과적으로 압축이 가능

허프만 코드(Huffman Code)

- 압축 해독: 만약 3비트로 한글자를 표현 하면 그냥 3비트씩 끊어서 읽으면 되지만, 가변길이의 경우 어디서 끊어 읽어야 할지가 정확하지 않음
- teen의 경우
  - 호프만 코드를 이용하면 01000010
  - 첫번째 글자는 0, 01, 010 이 가능
  - 그러나 코드 테이블에 0, 010 은 없음
  - 따라서 01이 첫글자임 (‘t’)
  - 두번째 글자는 0, 00, 000 이 가능
  - 코드 테이블에 00 만 존재함
  - 따라서 00 이 두번째 글자임 (‘e’)
  - 계속 반복함

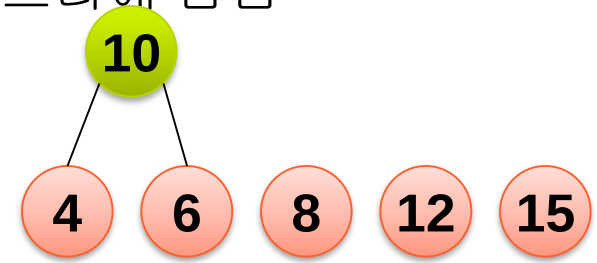
글자	비트 코드	빈도 수	비트 수
e	00	15	30
t	01	12	24
n	10	8	16
i	110	6	18
s	111	4	12
합계			88

허프만 코드(Huffman Code)

- 압축 해독이 가능한 원인: 모든 코드가 다른 코드의 첫부분이 아님. 따라서 비트열의 왼쪽에서 오른쪽으로 조사하여 보면 정확히 하나의 코드만 일치함. 이런 종류의 코드를 허프만 코드 라고 함

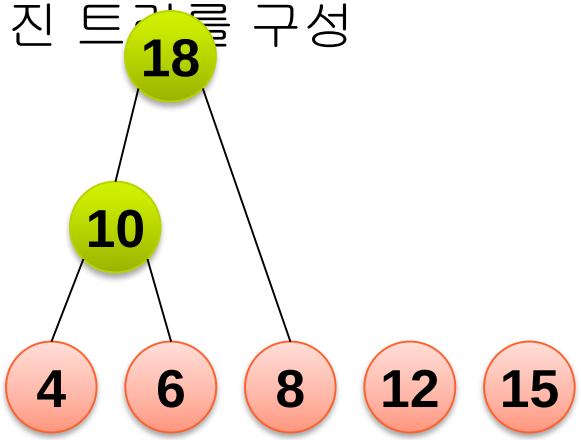
글자	빈도 수
e	15
t	12
n	8
i	6
s	4

- 허프만 코드 생성 절차
  - 먼저 빈도수에 따라 글자 나열
    - s(4), i(6), n(8), t(12), e(15)
  - 가장 작은 빈도수의 글자 두개로 이진 트리 구성하고 두 노드의 값을 합쳐서 이진 트리에 삽입

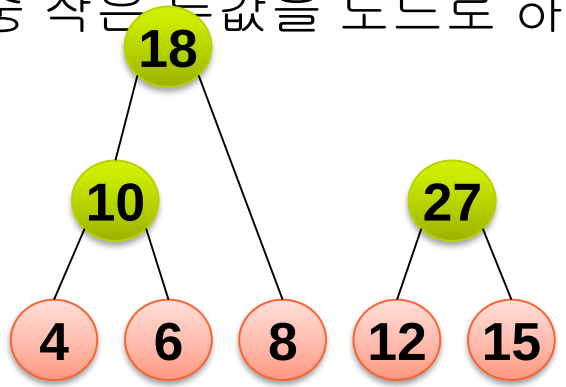


허프만 코드(Huffman Code)

- 이 빈도수를 정렬하여 (8, 10,12, 15)를 얻고, 이들중 다시 작은 두값을 이용하여 이진 트리를 구성

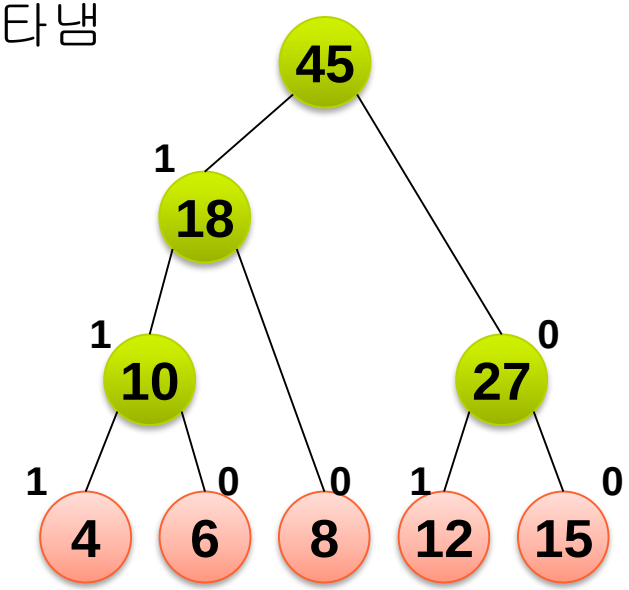


- 다시 두 노드의 값을 합해서 노드를 만들고 트리에 삽입하여 (12, 15, 18)을 얻고, 이중 작은 두값을 노드로 하여 이진 트리 구성



허프만 코드(Huffman Code)

- 같은 방법으로 (18, 27)을 이용하여 이진 트리를 구성하면 허프만 트리가 완성됨. 이 허프만 트리에서 왼쪽 간선은 비트 1을 오른쪽 간선은 비트 0을 나타냄



- 각 글자에 대한 호프만 코드는 루트로 부터 단말 노드까지의 경로에 있는 비트를 할당하면 된다