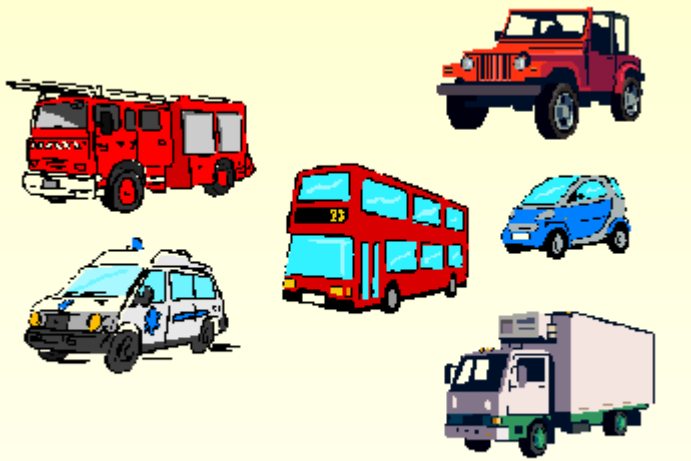
CHAP 8

우선순위 큐(Priority Queue)

우선순위 큐

- 의미 : 우선순위를 가진 데이터를 저장하는 큐
- 동작 : FIFO 순서가 아니라 우선 순위가 높은 데이터가 먼저 나가는 구조
- 비교 : 스택, 큐(입력된 순서) ➔ 특정 순서를 기준으로 우선순위 큐로 전환

자료구조	출력되는 순서
스택	*가장 **나중**에 들어온 데이터*
큐	*가장 **먼저** 들어온 데이터*
우선순위큐	*가장 **우선순위가 높은** 데이터*



- 응용분야
  - 시뮬레이션 시스템(사건의 시간 순)
  - 네트워크 트래픽 제어
  - 운영 체제에서의 작업 스케쥴링

우선순위 큐 ADT

·객체: n개의 element형의 우선 순위를 가진 요소들의 모임

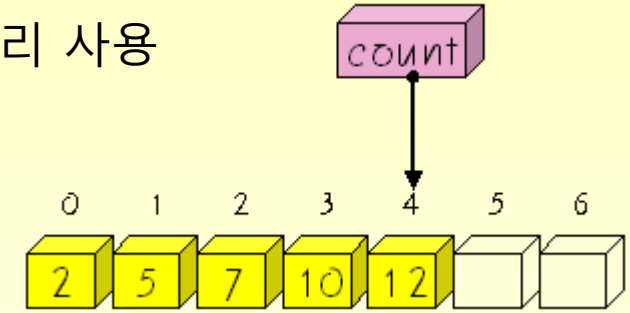
·연산:

- **create() : 우선순위 큐를 생성한다.**
- **init(q) : 우선순위 큐를 초기화한다.**
- **is_empty(q) : 우선순위 큐가 비어있는지를 검사한다.**
- **is_full(q) : 우선순위 큐가 가득 찼는가를 검사한다.**
- **insert(q, x) : 우선순위 큐에 요소 x를 추가한다.**
- **delete(q) : 우선순위 큐에서 우선순위가 높은 요소를 삭제/반환한다.**
- **find(q) : 우선순위 큐에서 우선순위가 가장 높은 요소를 반환한다.**

- 주요 연산 : insert(삽입), delete(삭제)
- 우선순위 큐의 종류
  - 최소 우선순위 큐　　최대 우선순위 큐

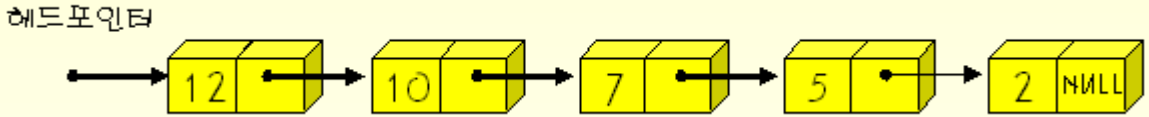
우선순위 큐 구현방법

- 배열로 구성한 우선순위 큐 : 크기 순으로 저장
- 연결리스트로 구성한 우선순위 큐 : 크기 순으로 저장
- 히프(heap)트리를 이용한 우선순위 큐 : 이진트리 사용

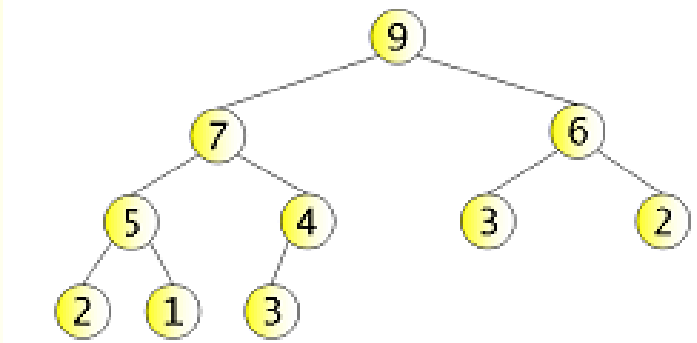


• (예)배열로 구성한 최소 우선순위 큐

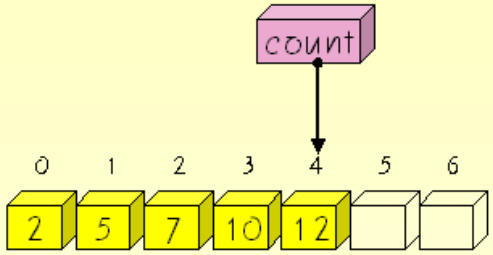
• (예)연결리스트로 구성한 최대 우선순위 큐



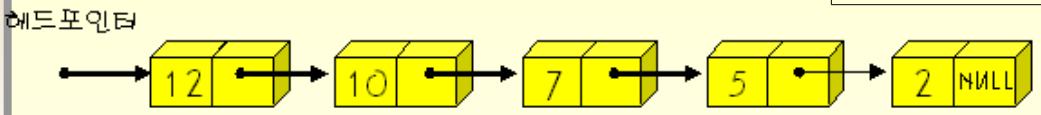
• (예)Heap 이진트리로 구성한 최대 우선순위 큐

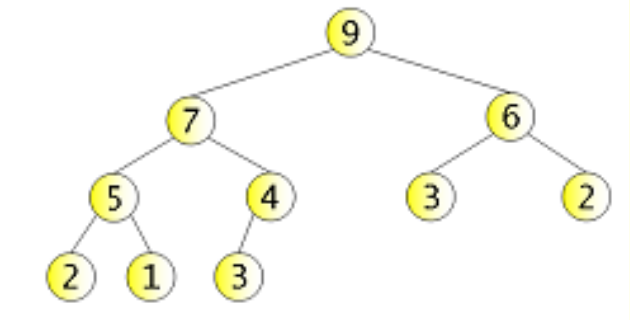


구현방법에 따른 우선순위 큐의 성능



표현 방법	삽입 성능	삭제 성능
순서없는 배열	O(1)	O(n)
순서없는 연결리스트	O(1)	O(n)
정렬된 배열	O(n)	O(1)
정렬된 연결리스트	O(n)	O(1)
히프	$O(\log_2 n)$	$O(\log_2 n)$





<우선순위 큐에서의 동작과정>

- 삽입 : 우선순위 큐에 삽입하여 자기자리에 맞게 재구성
- 삭제 : 맨앞의 데이터를 출력하고 제거한 후, 재구성

<예 1> 각 구조에 8 삽입

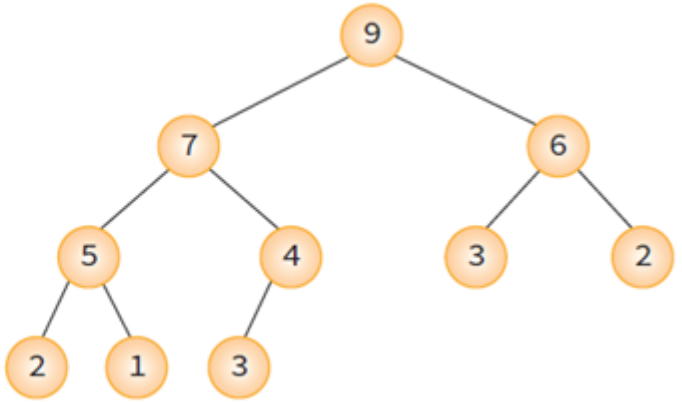
<예 2> 각 구조에 7 삭제

heap tree ?

Heap 트리의 구조 : 노드 키는 다음 조건을 만족하는 완전 이진트리

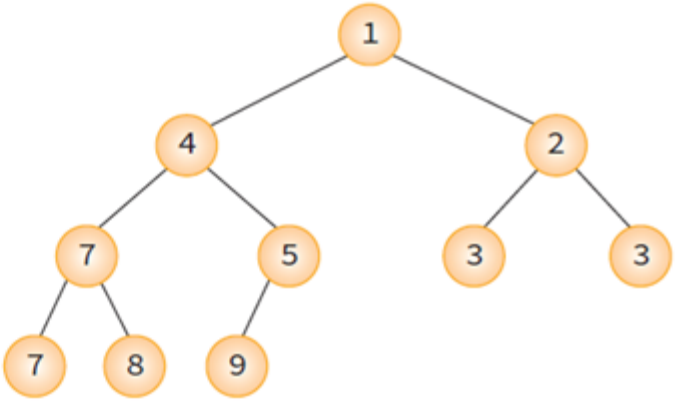
최대 히프(max heap):
부모 노드의 키값이 자식 노드의 키값보다 크거나 같은 완전 이진 트리

key(부모 노드) $\geq$ key(자식 노드)



최소 히프(min heap):
부모 노드의 키값이 자식 노드의 키값보다 작거나 같은 완전 이진 트리

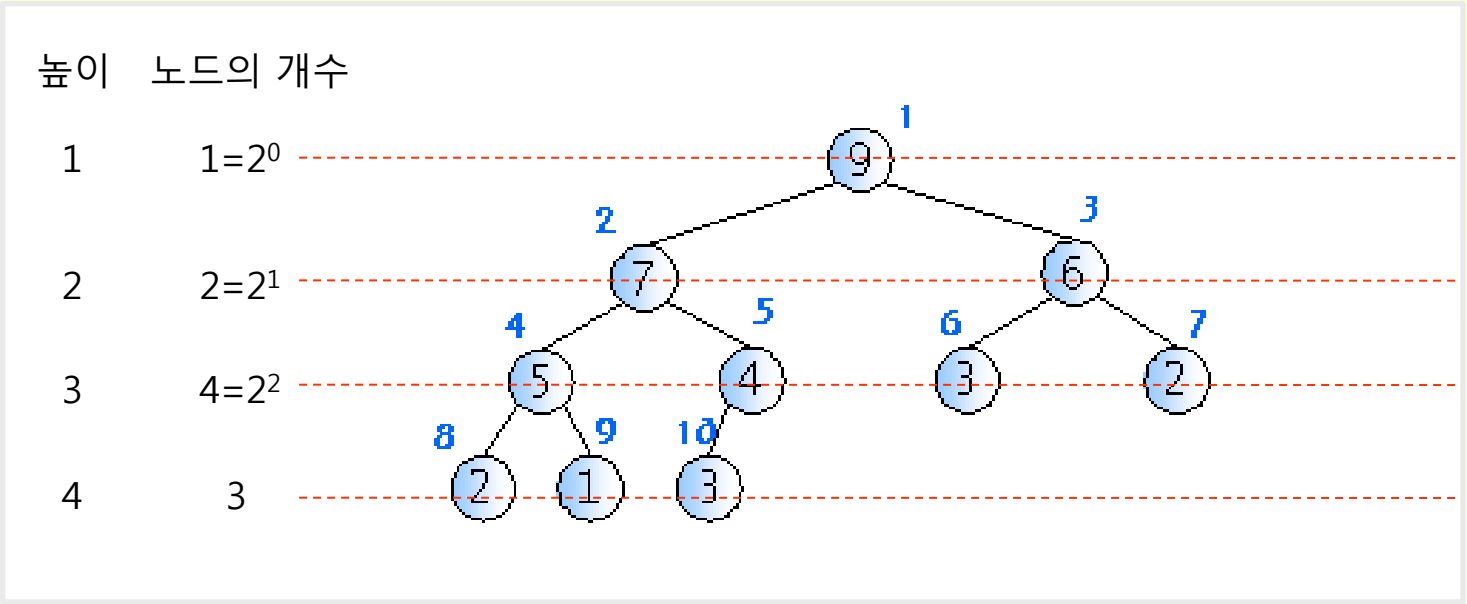
key(부모 노드) $\leq$ key(자식 노드)



히프 높이 : 성능과 비례

- n개의 노드를 가지고 있는 히프 트리의 높이 = $O(\log_2 n)$
  - 완전 이진트리 구조 ➔ 마지막 레벨을 제외한 모든 레벨 i은 2^{i-1}개의 노드가 존재
  - 아래 예에서, 히프 트리의 노드가 10개 ➔ 높이 = $O(\log_2 10)$ = 3.XXX = 4

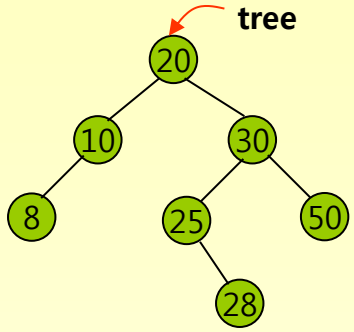
    히프 트리의 높이가 4 ➔ 8개 ≤ 노드 개수 ≤ 15개



<참고>이진트리 구현에 배열을 사용하면?

- 모든 노드는 자기 자리가 있음 : 루트는 첫번째(첨자 1) 위치

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	20	10	30	8	-	25	50	-	-	-	-	-	28	-	-

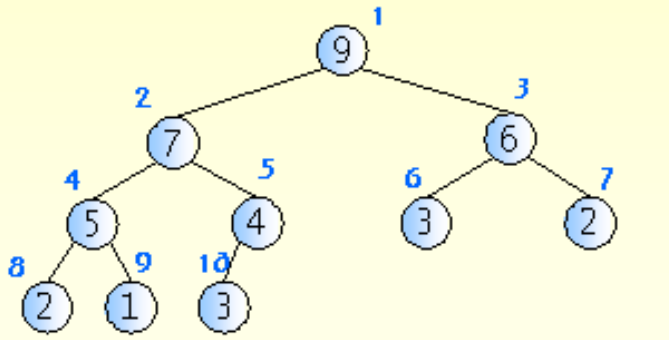


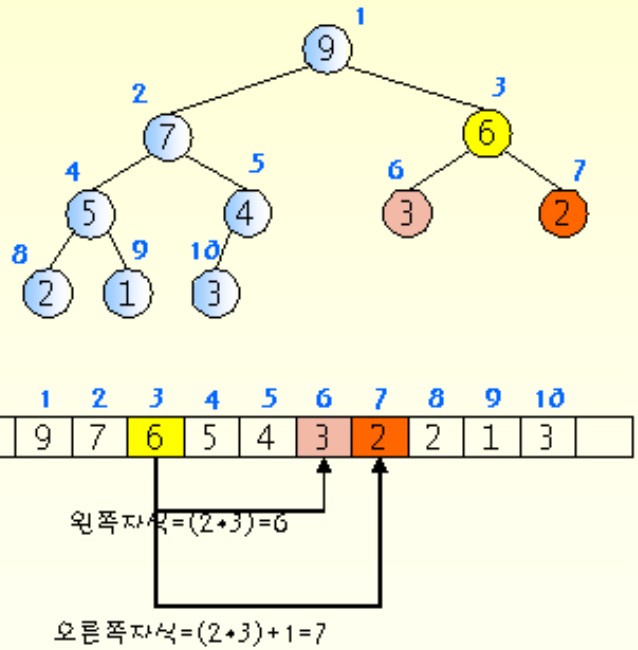
- 부모노드와 자식노드의 첨자 관계
  - 완전 이진트리 구조를 가정하면, 각 노드의 순서 번호를 배열의 첨자로 적용
    - **왼쪽 자식의 첨자 = (부모노드의 첨자) * 2**
    - **오른쪽 자식의 첨자 = (부모노드의 첨자) * 2 + 1**
    - **부모노드의 첨자 = (자식노드의 첨자) / 2**

- 트리로 구성한 이진 검색트리의 삽입, 삭제, 순회, 검색 동작 : 첨자 기준
  - **삽입** : (탐색 후) 삽입 값이 T[1]보다 작으면 T[2], 크면 T[3]➔ 반복 검색 ➔ T[i] 저장
  - **삭제** : (탐색 후) 삭제 값의 자식 유형이 무자식/외자식/쌍자식 경우에 따라 삭제
  - **순회** : (중위, 전위, 후위) 노드 포인터가 아닌 첨자를 이용하여 함수를 순환 호출

배열로 구현하는 방법

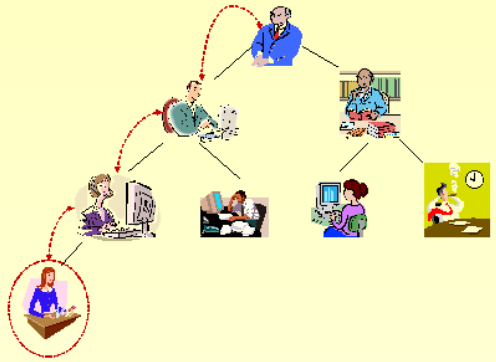
- 히프 트리를 배열로 구현한다면?
  - 완전 이진트리 구조를 가지므로 각 노드에 번호를 적용 ➔ 배열의 첨자로 활용
- 부모노드와 자식노드의 첨자를 계산하기 쉽다.
  - 왼쪽 자식의 첨자 = (부모노드의 첨자)*2　　오른쪽 자식의 첨자 = (부모노드의 첨자)*2 + 1
  - 부모노드의 첨자 = (자식노드의 첨자)/2





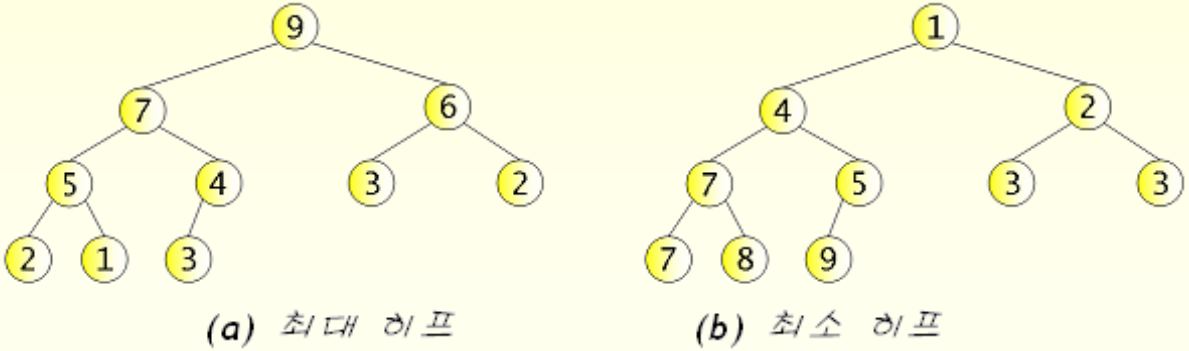
히프에서의 삽입

- 히프 트리의 삽입을 비유하자면...
  회사에 신입사원이 들어오면 일단 맨 아래자리에 앉힌 후, 신입사원의 업무수행능력을 평가해서 잘하면 승진, 못하면 그 자리에 그냥 유지시키는 동작



- 히프 트리의 삽입 동작

<단계 1> 히프에 새로운 데이터가 입력되면, 그 노드를 히프의 마지막 노드 위치에 삽입
<단계 2> 삽입 후, 노드 값을 부모 노드 값과 비교하여 우선순위가 더 높으면 부모 노드와 교환하는 과정을 반복 ➔ 히프 성질은 그대로 유지됨

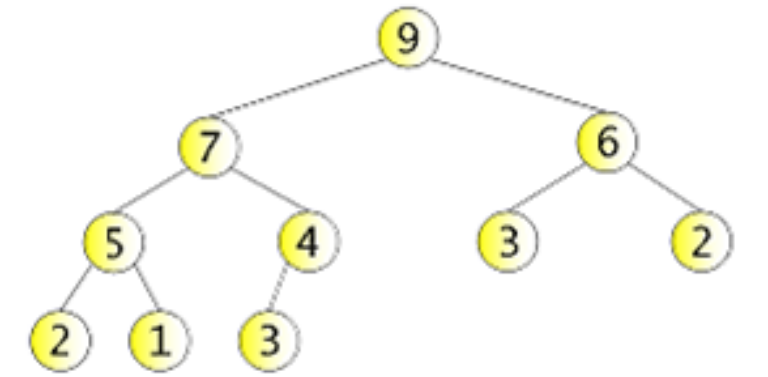


삽입 과정 : Up-heap 연산

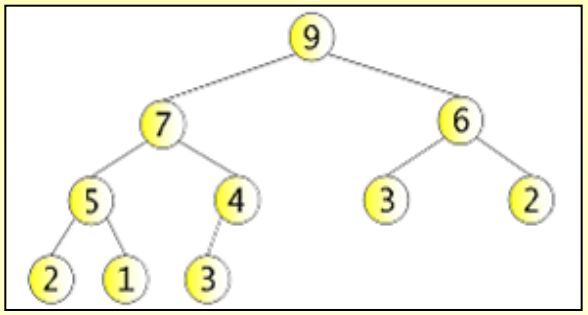
- 새로운 값 k 노드를 히프의 맨 마지막 노드로 저장 후, 히프의 고유 성질이 유지되도록 갱신 작업이 수행되어야 함
- 갱신 작업 : up-heap 과정 ➔ 새로 저장된 노드 위치 ~ 루트까지 경로에 있는 모든 노드들의 우선순위를 새로 저장된 노드의 우선순위와 비교

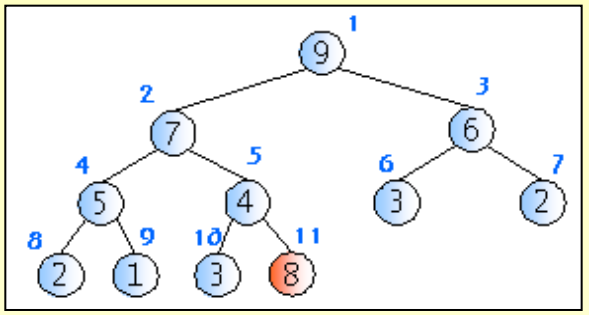
  if (새 노드의 우선순위 > 부모 노드의 우선순위) 부모-자식 노드의 교체;
  else 히프트리 리턴;

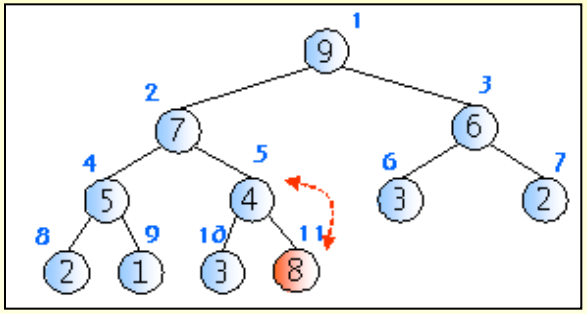
- 히프 트리의 높이 = $O(\log_2 n)$
  ➔ up-heap 연산 시간도 $O(\log_2 n)$

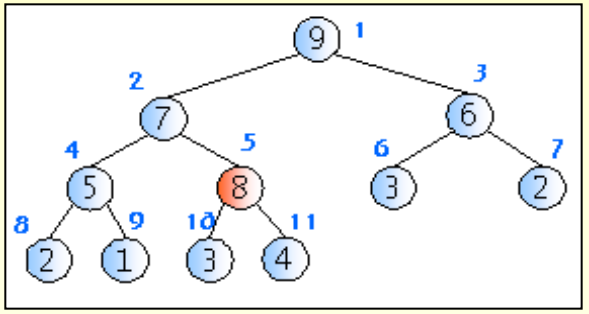


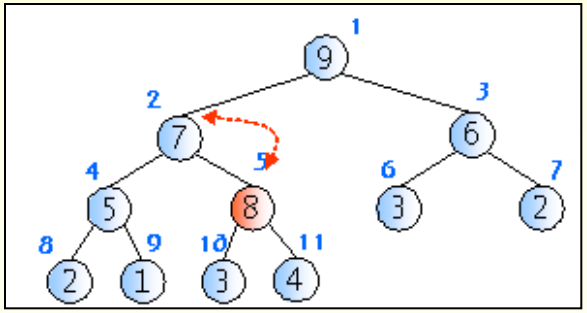
Up-heap 연산 : 8 삽입

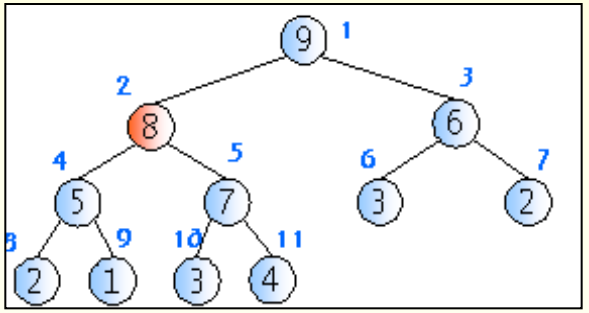












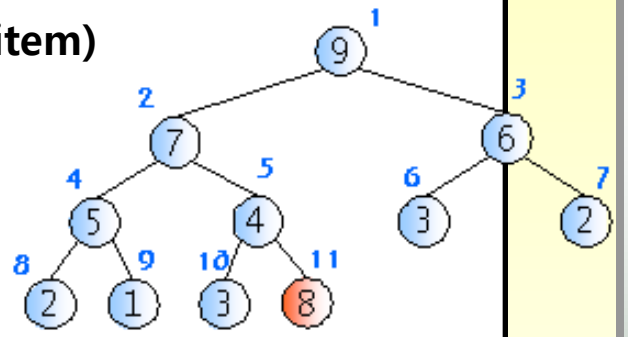
Up-heap 알고리즘

```
#define MAX_ELEMENT 200  //배열로 최대 200개 노드의 히프 트리를 구현
typedef   struct {
            int   heap[MAX_ELEMENT];
            int heap_size;     //실제 히프트리에 저장된 노드 개수
} HeapType;
HeapType   *heap = create(); // 선언 및 초기화 : heap_size = 0
```

Up-heap 알고리즘

```
//히프 트리를 배열로 구현하면...
// shem 개수가 heap_size인 히프트리 H에 item 삽입
HeapType * insert_max_heap(HeapType * H, int  item)
{
   int  i = (H->heap_size) + 1;

   while ((i != 1) && (item > H->heap[i/2]) )
   {
            H->heap[i] = H->heap[i/2]; //부모노드 값을 자식노드 값으로 저장
            i = i/2; //부모노드 자리로 up
   }
   H->heap[i] = item;   //더 이상 up되지 못하면 그 자리에 저장
}
```

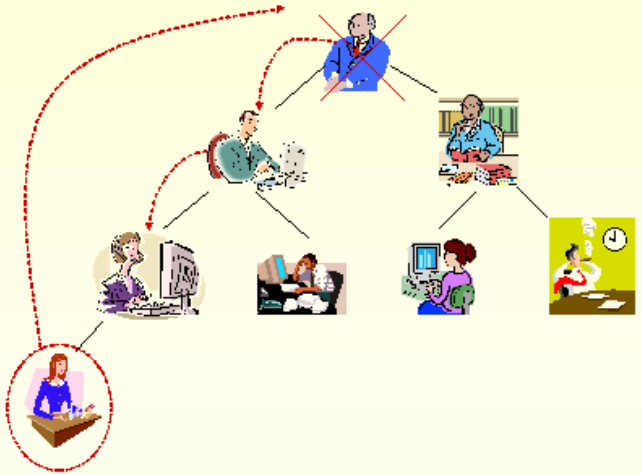


힙프에서의 삭제

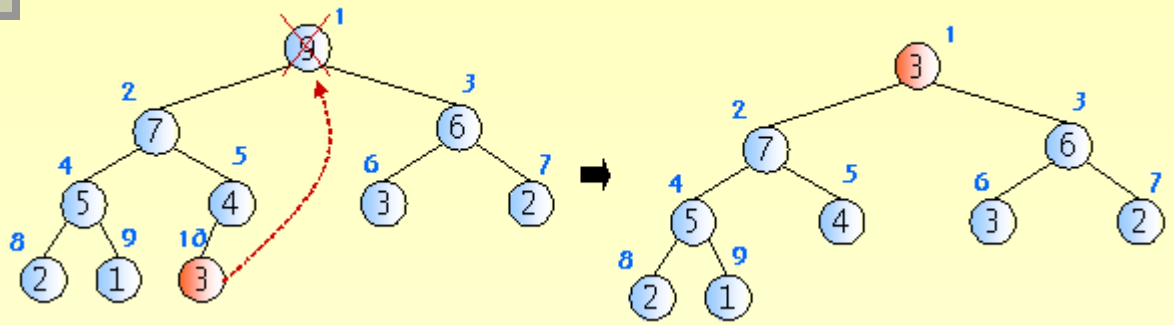
- 최대 힙프의 삭제 : 가장 큰 키 값을 가진 노드를 삭제하는 것을 의미
  ➔ 당연히 루트를 출력하고 삭제해야 함

- 비유를 하자면... 삭제해서 회사 사장의 자리가 비게 되면, 먼저 가장 말단에 있는 사원을 사장 자리로 올린 후에 사원의 능력이 밑에 있는 이사급보다 못하면 강등, 또 부장급보다 못하면 강등 ... 반복하는 과정

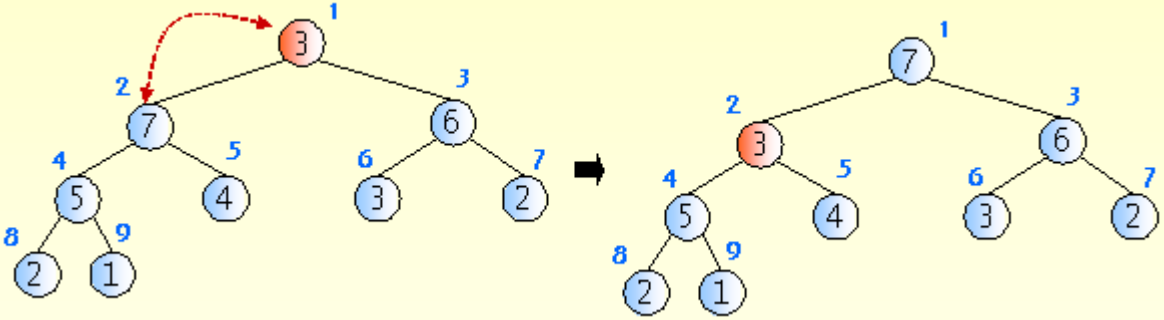
<단계 1> 루트노드를 출력(삭제)한다.
<단계 2> 맨 끝의 노드를 루트 노드로 저장한다.
<단계 3> 루트에서부터 단말노드까지의 경로 상에 있는 노드들을 비교하여 낮으면 내리고, 높으면 올리는 재구성과정을 반복한다.

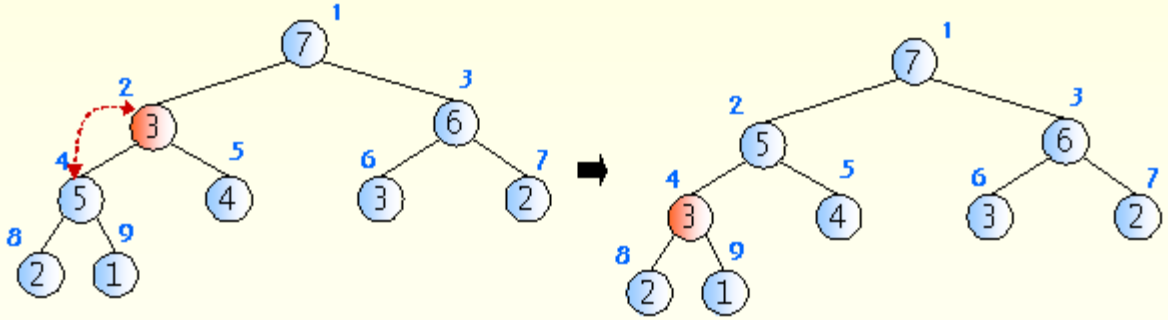


삭제 과정 : down heap



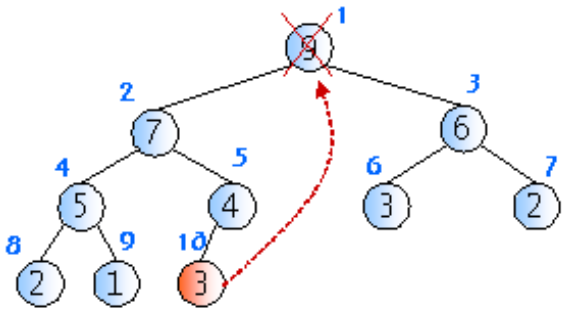
히프 높이 = $O(\log_2 n)$ ➔
down heap 시간도 $O(\log_2 n)$





Down-heap 알고리즘

```
int  delete_max_heap(HeapType *H)
{
    int  parent, child, item, temp;
    item = H->heap[1];  temp = H->heap[(H->heap_size)--];
    parent = 1;  child = 2;
    while( child <= H->heap_size )
    {  // 자식 노드들 중에서 우선순위가 제일 높은 자식노드 찾는다.
       if( ( child < H->heap_size ) && (H->heap[child]) < H->heap[child+1]) )
              child++;
       if( temp >= H->heap[child] ) break;
       // 한단계 아래로 이동
       H->heap[parent] = H->heap[child];
       parent = child;  child *= 2;
    }
    H->heap[parent] = temp;
    return item;
 }
```



Heap Tree 전체 프로그램

```
#include <stdio.h>
#include <stdlib.h>
typedef struct { int  heap[200];  int heap_size;} HeapType;
HeapType* create() { return (HeapType*)malloc(sizeof(HeapType)); }
void          init(HeapType* h) {  h->heap_size = 0;  }
HeapType* insert_max_heap(HeapType* h, element item) { ..... }
int           delete_max_heap(HeapType* h                    { ..... }
void main()
{
   HeapType * Heap = create();
   init(Heap);
   insert_max_heap(Heap, 10); insert_max_heap(Heap, 50); insert_max_heap(Heap, 80);
   insert_max_heap(Heap, 30); insert_max_heap(Heap, 90); insert_max_heap(Heap, 60);
   insert_max_heap(Heap, 40); insert_max_heap(Heap, 70);

   while (Heap->heap_size > 0)
       printf(" %d ", delete_max_heap(Heap));
}
```

히프를 이용한 응용 : 히프 정렬

- 히프 트리의 삽입/삭제 기능을 이용하면 정렬(sorting) 연산이 가능
  1) 먼저 정렬해야 할 n개 데이터를 최소 히프에 삽입
  2) 히프의 모든 데이터를 삭제하면 출력되므로 오름차순 정렬 결과로 나타남
- 시간 성능
  - 히프 트리에 데이터를 삽입하거나 삭제할 때의 시간 = $O(\log_2 n)$
  - 정렬할 데이터 개수가 n개 ➔ n개 * $O(\log_2 n)$ = $O(n\log_2 n)$: **so fast**
- 히프에 데이터를 저장하여 출력하는 정렬 알고리즘을 히프 정렬이라고 함

입력 예 : 5, 3, 7, 1, 8, 9, 4, 2, 6

히프정렬 프로그램

```
// 최소히프를 이용한 정렬함수 : 인수(데이터배열, 개수)
void heap_sort(int  data[], int  n)
{
    int i;
    HeapType  *h;   init(h); // 히프 초기화
    for(i=0; i<n; i++)
        insert_min_heap(h, data[i]); //히프에 배열요소를 차례대로 삽입
    for(i=(n-1); i>=0; i--)
        data[i] = delete_min_heap(&h); // 히프 삭제(출력)
}
```

Heap 트리를 이용한 우선순위큐 실습

```
#include <stdio.h>
void main()
{
    // 변수 선언
    // 메뉴방식으로 1)삽입  2)삭제  3)정렬  4)종료 선택

    insert_min_heap(H, item); //삽입

    printf("삭제된 값 = %d", delete_min_heap(H)); //삭제

    for (i=H->heap_size; i>=0; i--) //정렬
        printf(" %d ", delete_min_heap(H)); //힙트리의 노드갯수만큼 반복
}
```

허프만 코드

- 저장할 내용은 EBCD 또는 ASCII 코드로 저장 : 문자 당 8비트 크기 고정
  - 1024개 문자를 가진 문서 파일 = 1024byte(1KB)
  - 2000*1800 픽셀을 가진 이미지 파일 = 2000*1800*3B = 10MB
- 저장 용량을 줄이려면? **데이터마다 가변길이의 비트를 적용(일종의 압축)**
  - 자주 나타나는 데이터의 비트 수는 적게 배정
  - 가끔 나타나는 데이터의 비트 수는 그대로 배정
- 우선순위 큐를 응용하면 데이터의 비트 수를 가변적으로 적용 가능 ➔ 압축
  - 가변길이 비트 방식을 생성하는 이진트리 : **허프만 코딩 트리**

허프만 코드

- **허프만 코딩**
  - 데이터의 빈도 수를 이용하여 데이터의 비트 수를 차등화 배정하는 방식
  - 빈도 수에 의해 최소 히프 구성 ➔ 빈도 수가 적은 순서대로 우선순위 큐를 적용

빈도수 분석 →

A	80
B	16
C	32
D	36
E	123
F	22
G	26
H	51
I	71
…	
Z	1

허프만 코딩 : 데이터의 빈도 수 이용

- [예] 텍스트가 e, t, n, i, s의 5개의 글자로만 이루어졌다고 가정할 때, 글자의 빈도수가 다음과 같다고 가정

글자	비트 코드	빈도수	비트 수
e	000	15	3*15 = 45
t	110	12	3*12 = 36
n	010	8	3* 8 = 24
i	011	6	3* 6 = 18
s	100	4	3* 4 = 12
합계			133

허프만 코드 생성 절차

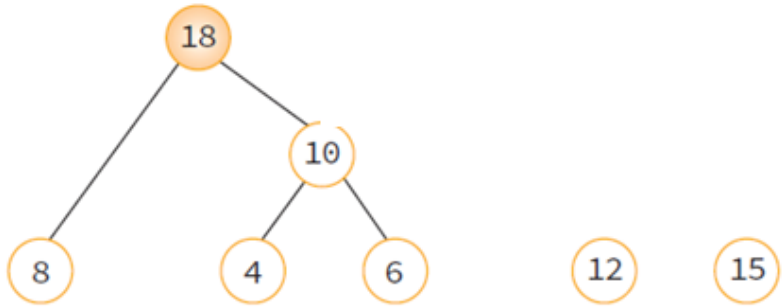
글자	비트 코드	빈도수	비트 수
e	000	15	3*15 = 45
t	110	12	3*12 = 36
n	010	8	3* 8 = 24
i	011	6	3* 6 = 18
s	100	4	3* 4 = 12
합계			133

~~4~~ ~~6~~ 8 12 15

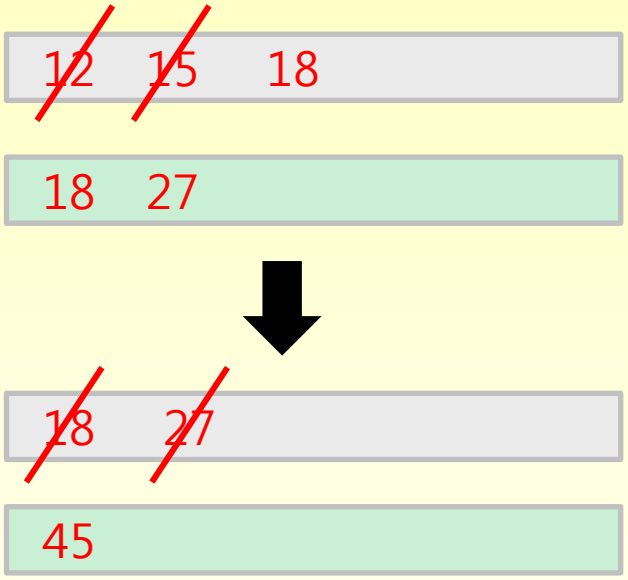
8 10 12 15

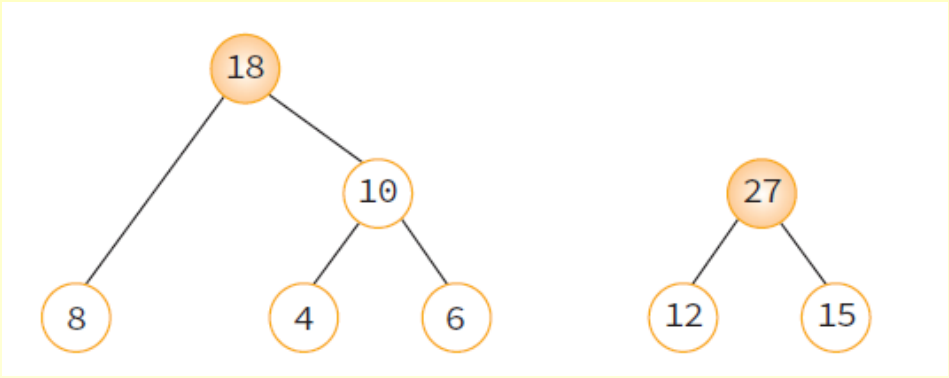
~~8~~ ~~10~~ 12 15

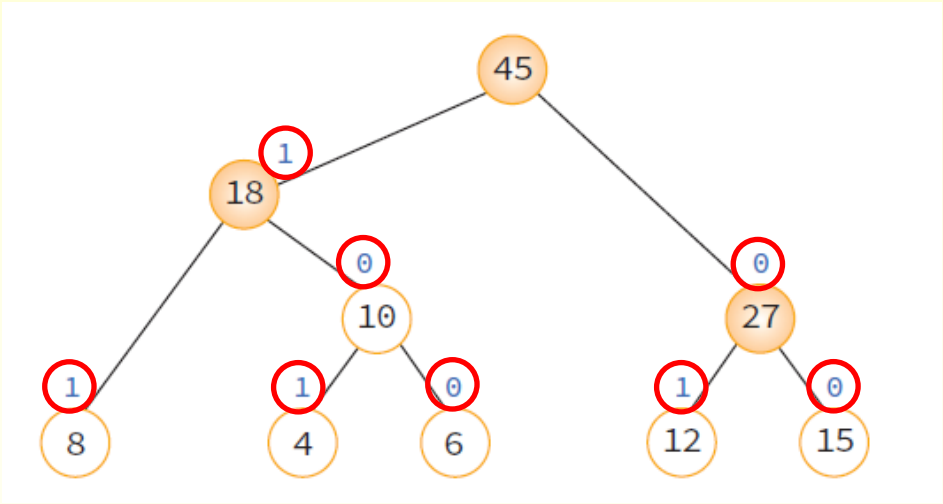
12 15 18



허프만 코드 생성 절차

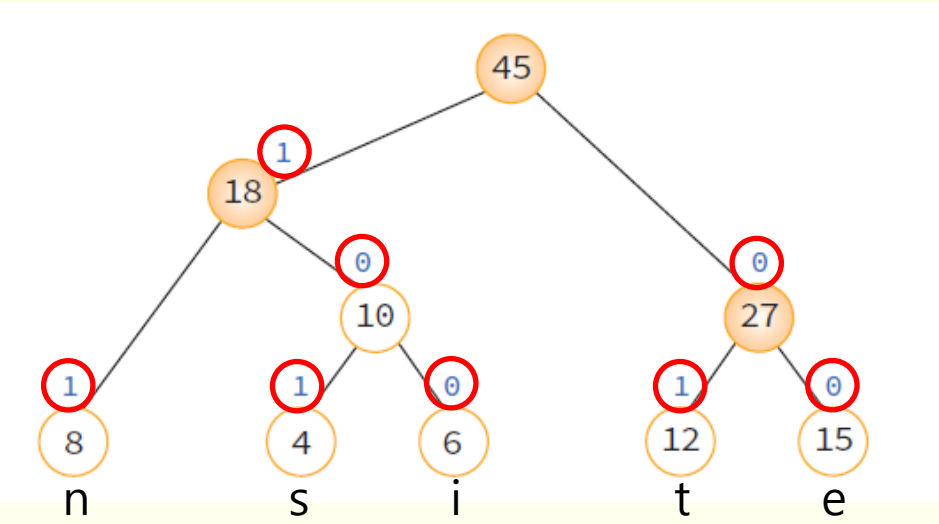

12 15 18
18 27
18 27
45


18
10
8
4
6
27
12
15


45
1
18
0
10
0
27
1
8
1
4
0
6
1
12
0
15

허프만 코드 생성 절차

글자	비트 코드	빈도수	비트 수
e	000	15	3*15 = 45
t	110	12	3*12 = 36
n	010	8	3* 8 = 24
i	011	6	3* 6 = 18
s	100	4	3* 4 = 12
합계			133



글자	비트 코드
e	00
t	01
n	10
i	110
s	111

2*15 = 30
2*12 = 24
2* 8 = 16
3* 6 = 18
3* 4 = 12
100

허프만 코드 프로그램

```
#include <stdio.h>
#include <stdlib.h>
typedef struct  { // 허프만 트리의 노드 구조
    int weight;   char ch;   struct TreeNode *left;   struct TreeNode *right;
} TreeNode;

typedef struct { // 우선순위 큐에 저장될 데이터 구조
    TreeNode* ptree;   char ch;   int key;
} element;

typedef struct { // 우선순위 큐 구조
   element  heap[MAX_ELEMENT];  int  heap_size;
} HeapType;
```

```
HeapType* create() {  return (HeapType*)malloc(sizeof(HeapType)); }
void init(HeapType* h) {  h->heap_size = 0; }
void insert_min_heap(HeapType* h, element item) {  ……..  }
int   delete_min_heap(HeapType* h) {  ……..  }

// 허프만 트리 생성 함수
TreeNode* make_tree(TreeNode* left, TreeNode* right) {
    TreeNode* node = TreeNode*)malloc(sizeof(TreeNode));
    node->left = left;   node->right = right;
    return node;
}
// 허프만 트리 제거 함수
void destroy_tree(TreeNode* root) {
    if (root == NULL) return;
    destroy_tree(root->left);
    destroy_tree(root->right);
    free(root);
}
```

```
int  is_leaf(TreeNode* root) {
    return !(root->left) && !(root->right);
}
void  print_array(int codes[], int n)  {
    for (int i = 0; i < n; i++)   printf("%d", codes[i]);
}
void  print_codes(TreeNode* root, int codes[], int top) {
    if (root->left) {// 왼쪽 링크를 따라가면 허프만 코드1 저장하고 순환 호출
        codes[top] = 1;
        print_codes(root->left, codes, top + 1);
    }
   if (root->right) {// 오른쪽 링크를 따라가면 허프만 코드0 저장하고 순환 호출
        codes[top] = 0;
        print_codes(root->right, codes, top + 1);
   }
   if (is_leaf(root)) { // 링크 모두 NULL(단말노드)이면 코드 출력
        printf("%c: ", root->ch);  print_array(codes, top);
   }
}
```

```
void huffman_tree(int freq[], char ch_list[], int n) // 허프만 코드 생성 함수
{
    int i;    TreeNode *node, *x;    HeapType* heap;    element e, e1, e2;
    int codes[100];     int top = 0;
    heap = create();   init(heap); // 히프 생성 및 초기화
    for (i = 0; i<n; i++) {
        node = make_tree(NULL, NULL);   e.ch = node->ch = ch_list[i];
        e.key = node->weight = freq[i];    e.ptree = node;
        insert_min_heap(heap, e);
    }
    for (i = 1; i<n; i++) {
        e1 = delete_min_heap(heap); // 우선순위 큐에서 최소값2개 노드를 삭제
        e2 = delete_min_heap(heap);
        x = make_tree(e1.ptree, e2.ptree); // 허프만트리에서 2개 노드를 결합
        e.key = x->weight = e1.key + e2.key;
        e.ptree = x;
        printf("%d+%d->%d \n", e1.key, e2.key, e.key);
        insert_min_heap(heap, e);
     }
     e = delete_min_heap(heap); // 허프만 트리 완성
     print_codes(e.ptree, codes, top);
     destroy_tree(e.ptree);
}
```

```
void   main(void)
{
      char ch_list[] = { 's', 'i', 'n', 't', 'e' };  //입력 데이터
       int   freq[]     = { 4, 6, 8, 12, 15 };    //데이터 빈도 수
       huffman_tree(freq, ch_list, 5);
}
```

<우선순위 큐에서 결합된 결과>

4+6 → 10
10+12 → 22
8+15 → 23
22+23 → 45

<허프만 코드 : 문자별 이진코드>

s 111
i 110
t 10
n 01
e 00