우선순위 큐

Priority queue

우선순위 큐

일반적인 큐 vs. 우선순위 큐

- 일반적인 큐: 먼저 들어온 데이터가 무조건 먼저 나가는 자료구조
- 우선순위 큐: 우선순위를 가진 항목들을 저장하여 해당 항목들이 먼저 나가는 큐

우선순위가 있는 대기열

일반적인 대기열

우선순위 큐

우선순위 큐

- 우선순위를 가진 항목들을 저장하는 큐
- 우선 순위가 높은 데이터가 먼저 나가게 됨
- 가장 일반적인 큐로 생각할 수 있음
  * 스택이나 큐를 우선 순위 큐로 구현 가능

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

우선순위 큐

일반화

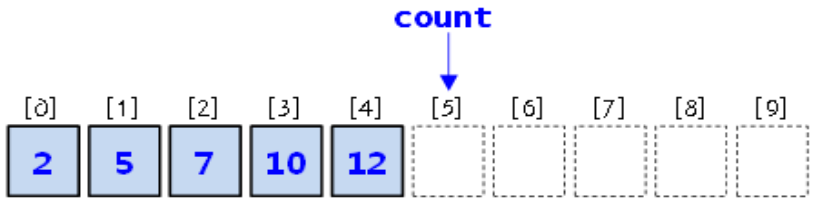
- 스택이나 FIFO 큐를 우선순위 큐로 재 구현 가능

- 연산
  - create(): 우선순위 큐 생성
  - init(): 우선순위 큐 초기화
  - is_empty(): 우선순위 큐가 비어 있는지 검사
  - is_full(): 우선순위 큐가 가득 차 있는지 검사
  - insert(q, x): 우선순위 큐에 요소 추가 (우선순위 포함)
  - delete(q): 우선순위 큐로부터 가장 우선순위가 높은 요소를 삭제하고 이 요소를 반환
  - find(q): 우선 순위가 가장 높은 요소를 반환

- 구분: 최소 / 최대 우선순위 큐

우선순위 큐

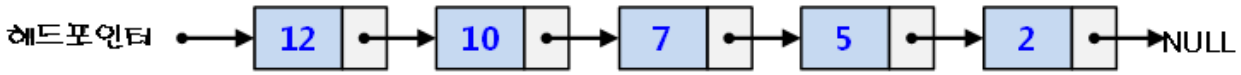
구현 방법 (배열)



- 정렬되지 않은 배열 / 정렬된 배열
- 정렬되지 않은 배열
  - 삽입: 바로 삽입 O(1)
  - 삭제: 가장 높은 요소를 찾아 삭제 O(n)
- 정렬된 배열
  - 삽입: 위치를 찾은 후 삽입 O(n)
  - 삭제: 맨 뒤 위치한 요소 삭제 O(1)

우선순위 큐

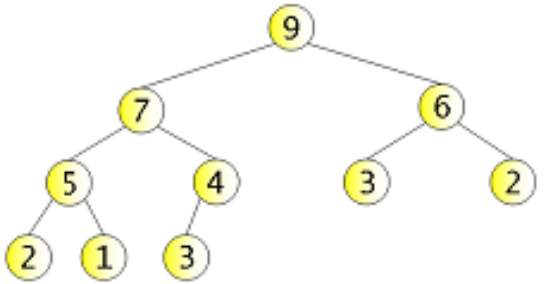
구현 방법(연결 리스트)



- 배열과 거의 동일
- 정렬 고려 하지 않은 연결 리스트 / 정렬된 연결 리스트
- 정렬 고려 하지 않은 연결 리스트
  - 삽입: 첫 번째 노드 삽입 O(1)
  - 삭제: 모든 노드를 검색하여 높은 요소 찾은 후 삭제 O(n)
- 정렬된 연결 리스트
  - 삽입: 우선 순위 값 기준으로 삽입위치 검색 후 삽입 O(n)
  - 삭제: 첫번째 노드 삭제 O(1)

우선순위 큐

구현 방법(힙, heap)



- 완전 이진 트리 일종
- 우선순위 큐를 위해 만들어진 자료 구조
- 느슨한 정렬 상태 유지
- 앞에서 보여준 배열/연결 리스트는 시간 복잡도가 O(n)/O(1) 이 소요 되지만 힙의 경우 O($log_2 n$) 이 걸림
- 최대 (max) / 최소 (min) 힙

시간 복잡도가 n 과 $log_2 n$ 의 차이

n= 1000 라고 한다면

O(n) = 1000초 / O($log_2 n$) = 약 10초

100배

위 일수록 우선순위↑

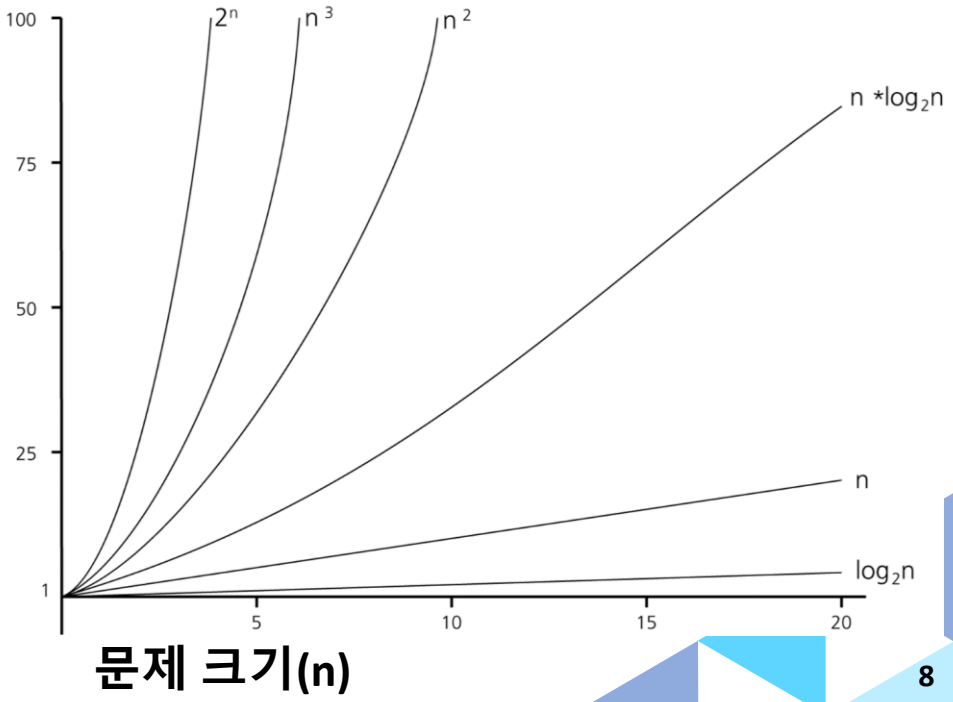
- 최대 힙
  - 부모 노드 키 값이 자식 노드 키 값보다 크거나 같은 완전 이진 트리
- 최소 힙
  - 부모 노드 키 값이 자식 노드 키 값보다 작거나 같은 완전 이진 트리

우선순위 큐

바람직한 알고리즘

- 명확
  - 이해하기 쉽고 가능하면 간명
  - 지나친 기호적 표현은 오히려 명확성을 떨어뜨림
  - 명확성을 해치지 않으면 일반언어 사용 무방

- 효율적
  - 같은 문제를 해결하는 알고리즘들의 수행 시간이 수백만 배 차이 날 수 있음



빠르다고 무조건 좋은 건 아님,
정확성이 떨어질 수도 있음

우선순위 큐

시간 복잡도

- 알고리즘을 풀다 보면 문제의 해답을 찾는 것이 중요한데 이 때, 효율적인 방법을 고민

- 효율적인 방법? = 시간이 적게 걸림 (대표)

- 고로 시간 복잡도를 고민한다는 것

Big-O(빅-오) => 상한 점근

최악 고려

Big-Ω(빅-오메가) => 하한 점근

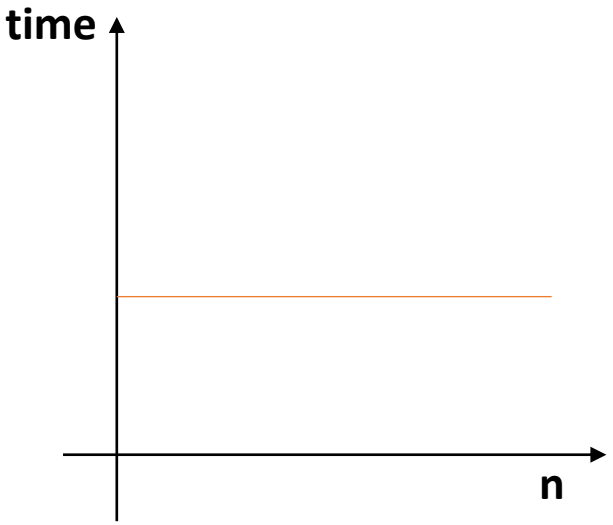
최선 고려

Big-θ(빅-세타) => 상한과 하한의 평균

우선순위 큐

시간 복잡도 O(1) arr[i]

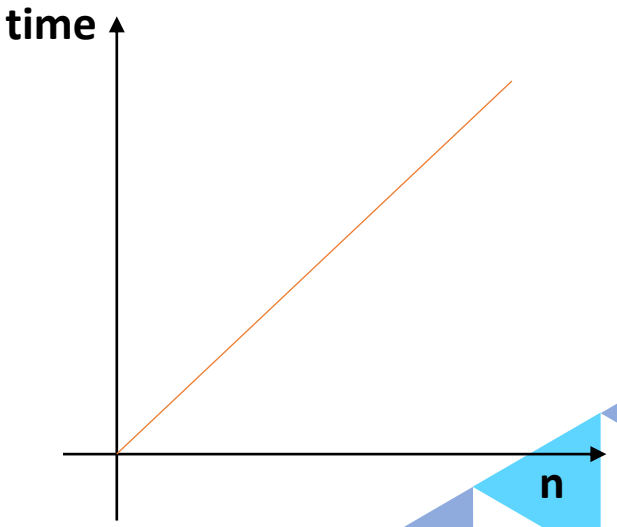
- 일정한 복잡도(constant complexity)
- 입력 값이 증가하더라도 시간이 늘어나지 않음



시간 복잡도 O(n)

- 선형 복잡도(linear complexity)
- 입력 값이 증가함에 따라 시간 또한 같은 비율로 증가

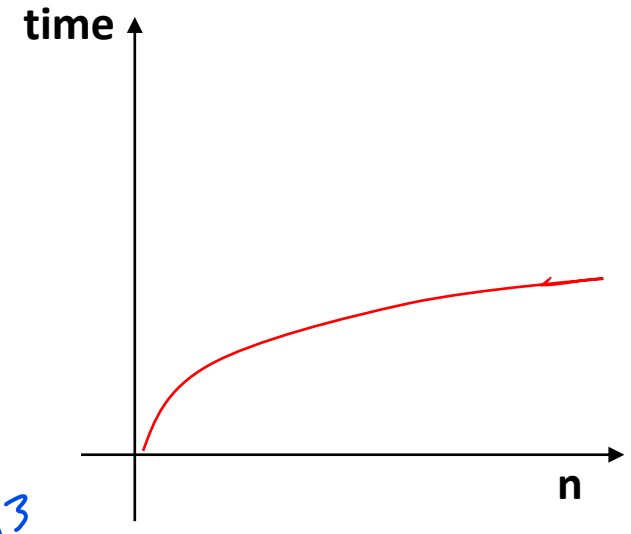
for(int i = 0; i<n ; i++)



우선순위 큐

시간 복잡도 O(log n)

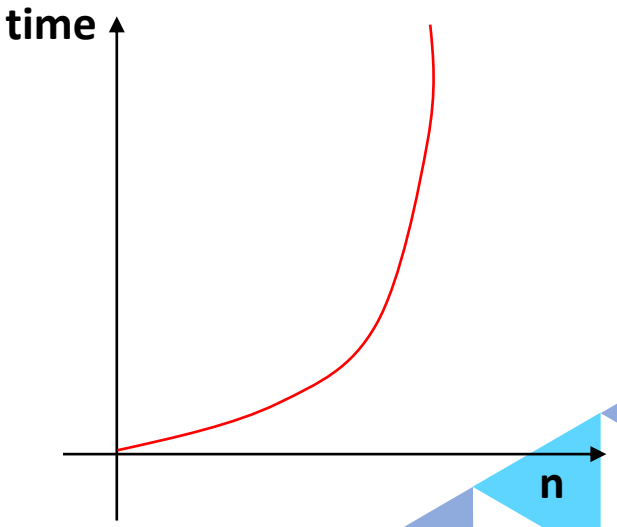
- 로그 복잡도(logarithmic complexity)
- O(1) 다음으로 빠른 시간 복잡도



시간 복잡도 $O(n^2)$

for문이 3개면 n^3

- 2차 복잡도(quadratic complexity)
- 입력 값이 증가함에 따라 시간이 n의 제곱수의 비율로 증가




```
for(int i=0 ; i<n ; i++)
    for(int j=0; j<n ; j++)
```

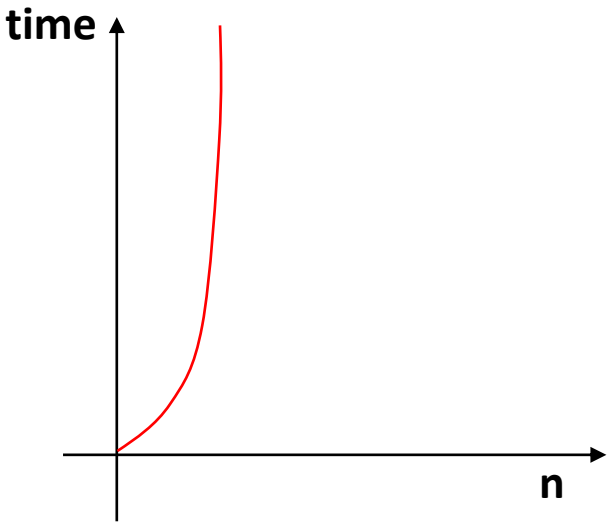
우선순위 큐

시간 복잡도 $O(2^n)$

- 기하급수적 복잡도
  (exponential complexity)

- 가장 느린 시간 복잡도

ex) 재귀함수 피보나치 수열



우선순위 큐

알고리즘 수행 시간

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

- 알고리즘 수행 시간을 좌우하는 기준은 다양함
  - for 루프의 반복횟수, 특정한 행이 수행되는 횟수, 함수 호출 횟수 ...

우선순위 큐

알고리즘 수행 시간

```
sample1(A[ ], n) {
        k =          ;
        return A[k];
}
```

O(1)

```
sample2(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                sum← sum+ A[i] ;
        return sum ;
}
```

O(n)

```
sample3(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                for j ← 1 to n
                        sum← sum+ A[i]*A[j] ;
        return sum ;
}
```

$O(n^2)$

```
sample4(A[ ], n) {
        sum ← 0 ;
        for i ← 1 to n
                for j ← 1 to n {
                        k ← A[1 … n]에서 임의로 [n/2]개를
뽑을 때 이들 중 최댓값 ;
                        sum ← sum + k ;
                }
        return sum ;
}
```

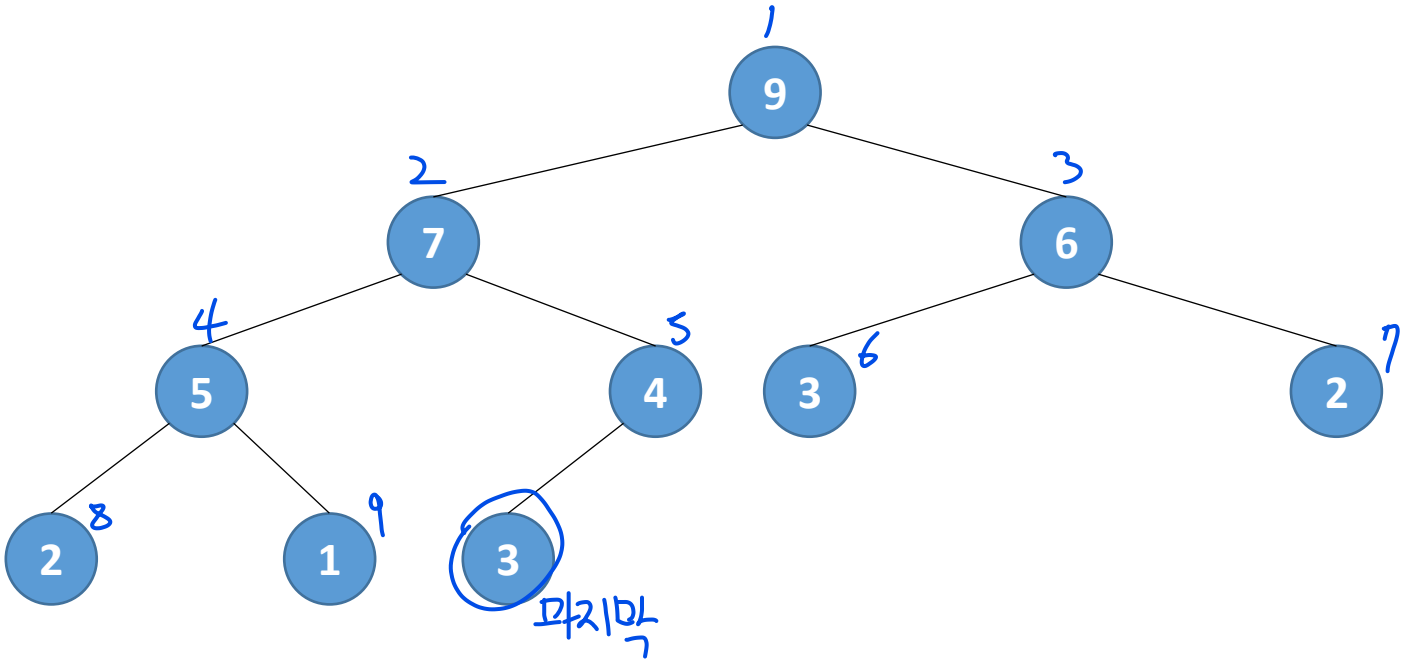
$O(n^3)$

우선순위 큐

완전 이진 트리의 일종

구현 방법(힙, heap)

- 완전 이진 트리 일종
- 마지막 레벨을 제외하고는 각 레벨에 $2^{level-1}$ 노드 존재



1Lv $1=2^0$

2Lv $2=2^1$

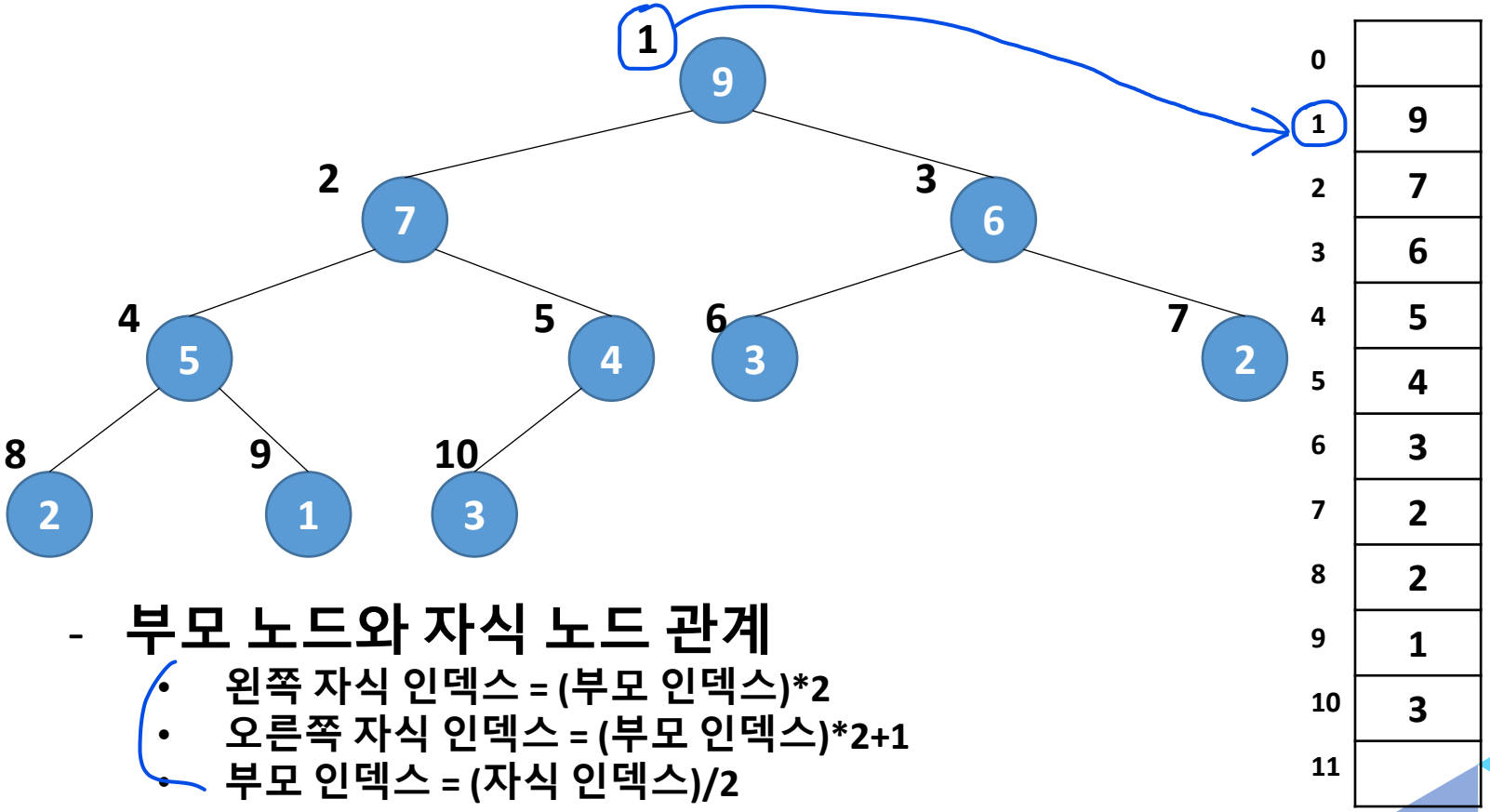
3Lv $4=2^2$

4Lv 3

우선순위 큐

구현 방법(힙, heap)

- 배열 이용
  - 완전 이진 트리 -> 각 노드에 번호 붙임 -> 배열 인덱스



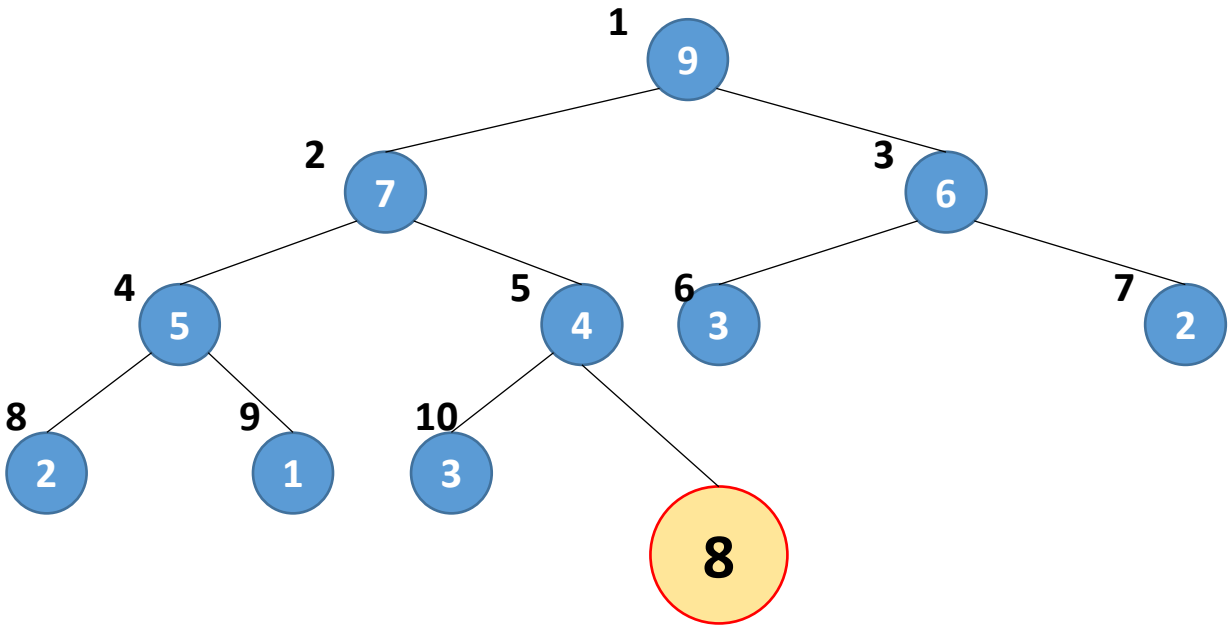
- 부모 노드와 자식 노드 관계
  - 왼쪽 자식 인덱스 = (부모 인덱스)*2
  - 오른쪽 자식 인덱스 = (부모 인덱스)*2+1
  - 부모 인덱스 = (자식 인덱스)/2

우선순위 큐

구현 방법(힙, heap) – 삽입 [최대 힙]

- 삽입의 경우 2 단계로 나눠서 진행

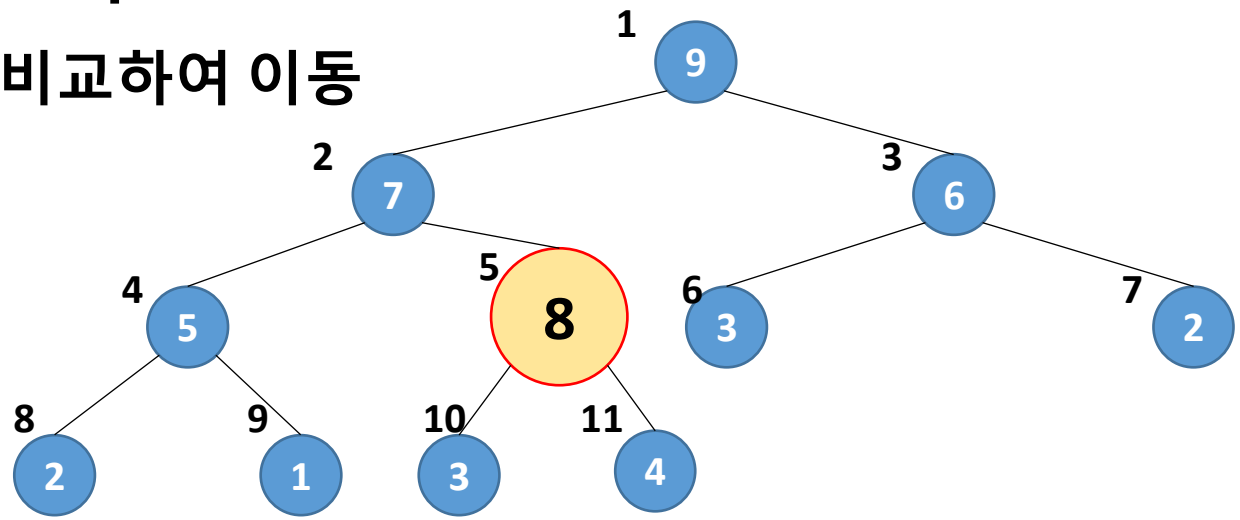
1. 새로운 요소가 들어오면 일단 히프의 마지막 노드에 삽입
2. 부모 노드들과 교환하여 히프 성질 만족



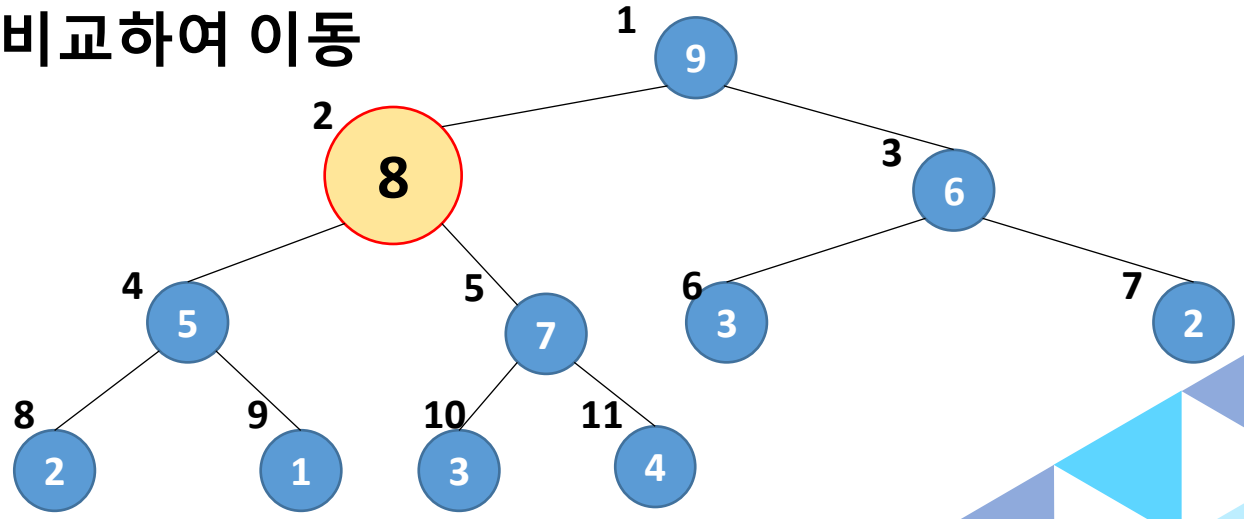
우선순위 큐

구현 방법(힙, heap) – 삽입 [최대 힙]

- 부모 (5) 노드와 비교하여 이동



- 부모 (2) 노드와 비교하여 이동

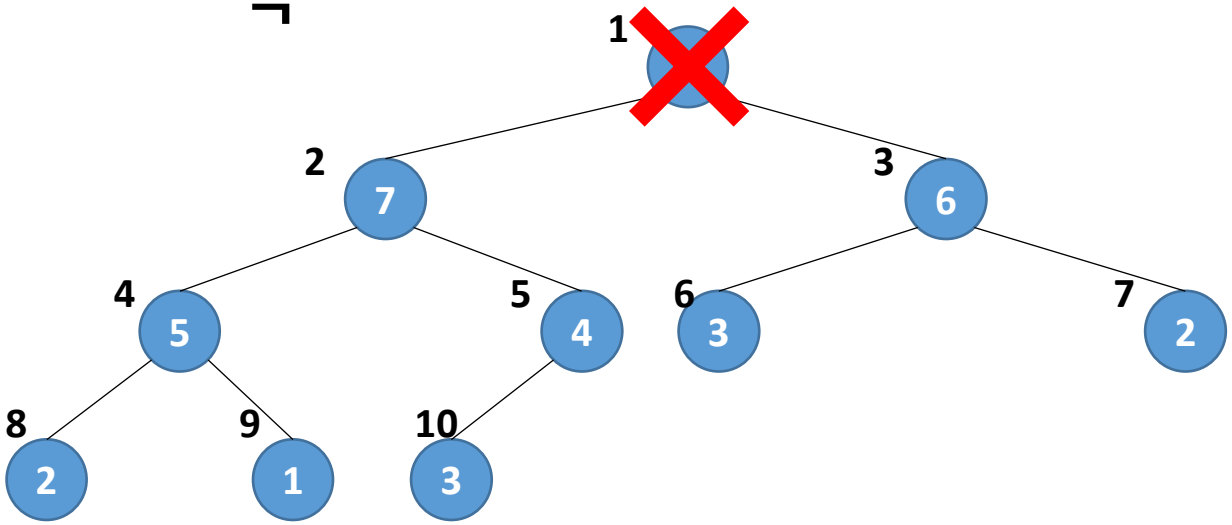


우선순위 큐

구현 방법(힙, heap) – 삭제 [최대 힙]

- 최대 힙의 삭제는 큰 키 값을 가진 노드를 삭제하는 것이므로 루트 노드가 삭제

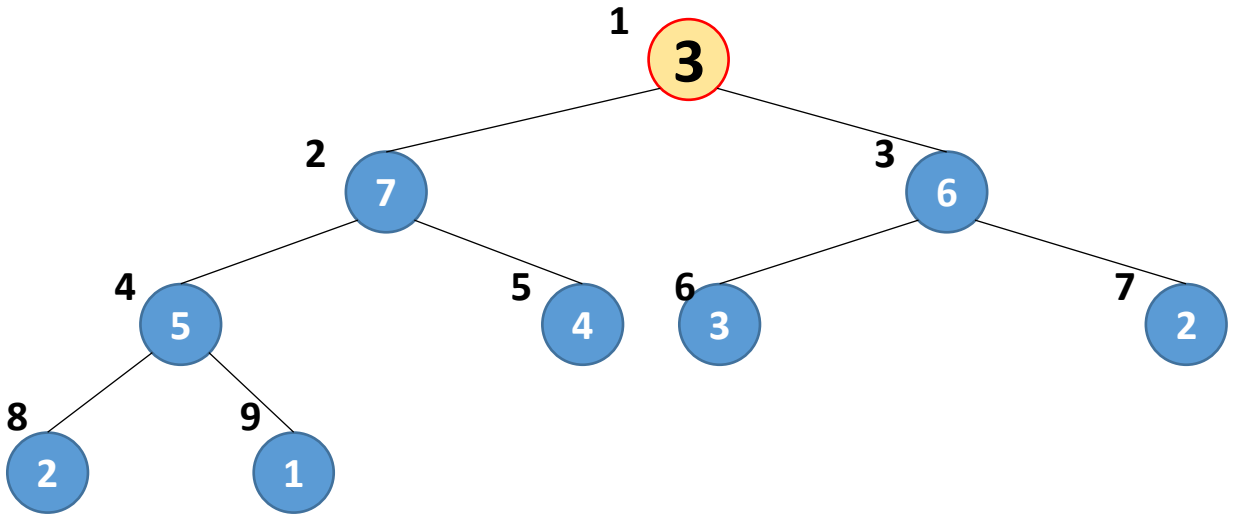
1. 루트 노드 삭제
2. 마지막 노드를 루트 노드로 이동
3. 루트에서부터 단말 노드까지 노드들을 교환하여 힙 성질 만족



우선순위 큐

구현 방법(힙, heap) – 삭제 [최대 힙]

- 마지막 (10) 노드를 루트로 이동

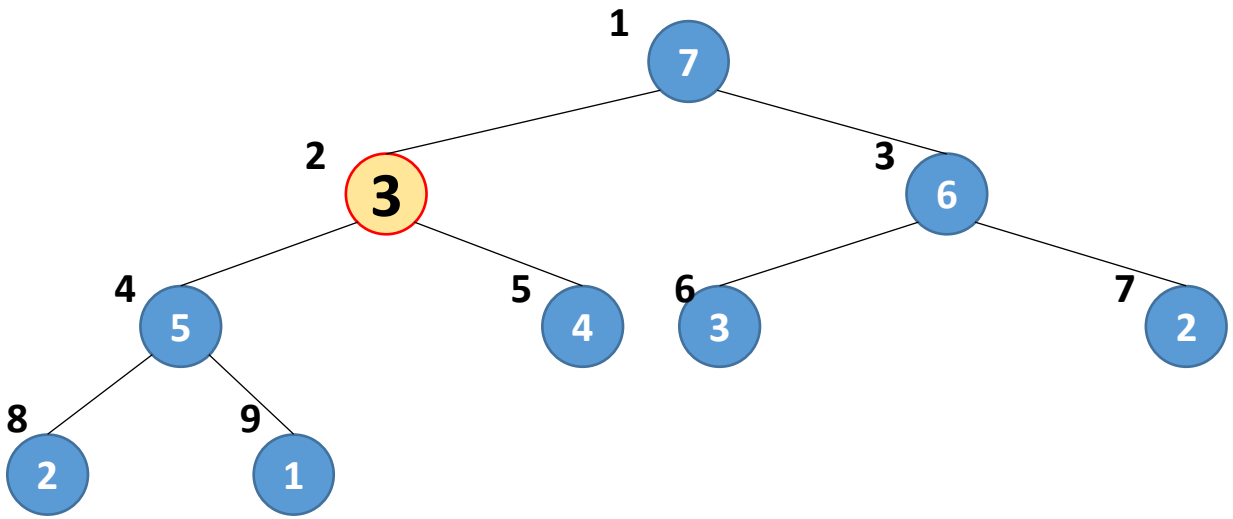


- 최대 힙 조건
  - 부모 노드 값이 자식 노드 값보다 높거나 같다.

우선순위 큐

구현 방법(힙, heap) – 삭제 [최대 힙]

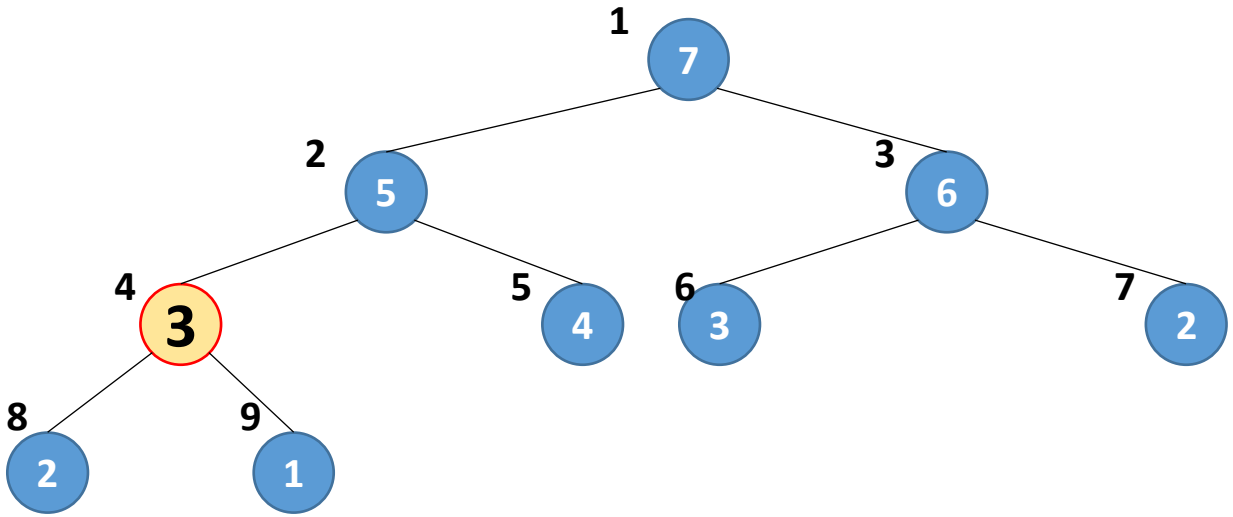
- 루트 노드(1)와 자식 노드(2) 비교 후 이동



우선순위 큐

구현 방법(힙, heap) – 삭제 [최대 힙]

- 부모 노드(2)와 자식 노드(4) 비교 후 이동



우선순위 큐

구현 방법 (힙, heap)

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENT 200
typedef struct {
  int key;
} element;
typedef struct {
  element heap[MAX_ELEMENT];
  int heap_size;
} HeapType;

HeapType* create(){
  return
(HeapType*)malloc(sizeof(HeapType));
}

void init(HeapType* h){
  h->heap_size = 0;
}
```

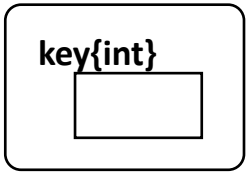
```c
void insert_max_heap(HeapType* h, element item){
  int i;
  i = ++(h->heap_size);
  while ((i != 1) && (item.key > h->heap[i / 2].key)) {
  h->heap[i] = h->heap[i / 2];
  i /= 2;
  }
  h->heap[i] = item;
}
element delete_max_heap(HeapType* h){
  int parent, child;
  element item, temp;
  item = h->heap[1];
  temp = h->heap[(h->heap_size)--];
  parent = 1;
  child = 2;
  while (child <= h->heap_size) {
    if ((child < h->heap_size) &&
      (h->heap[child].key) < h->heap[child + 1].key)
      child++;
    if (temp.key >= h->heap[child].key) break;
    h->heap[parent] = h->heap[child];
    parent = child;
    child *= 2;
  }
  h->heap[parent] = temp;
  return item;
}
```

우선순위 큐

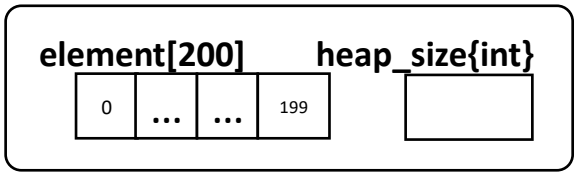
구현 방법 (힙, heap)

```
int main(void){
  element e1 = { 10 }, e2 = { 5 }, e3 = { 30 };
  element e4, e5, e6;
  HeapType* heap;

  heap = create();
  init(heap);

  insert_max_heap(heap, e1);
  insert_max_heap(heap, e2);
  insert_max_heap(heap, e3);

  e4 = delete_max_heap(heap);
  printf("< %d > ", e4.key);
  e5 = delete_max_heap(heap);
  printf("< %d > ", e5.key);
  e6 = delete_max_heap(heap);
  printf("< %d > ₩n", e6.key);

  free(heap);
return 0;
}
```

element



HeapType

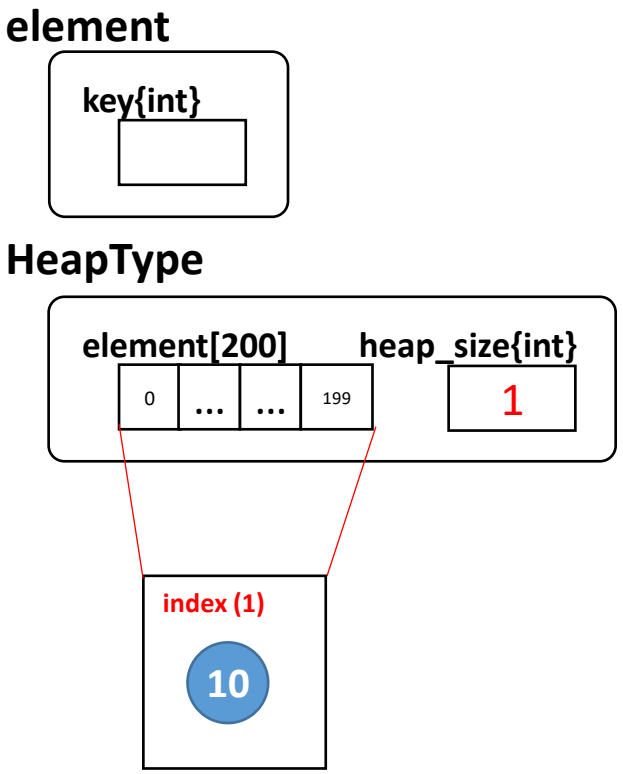


ex1

우선순위 큐

구현 방법 (힙, heap)

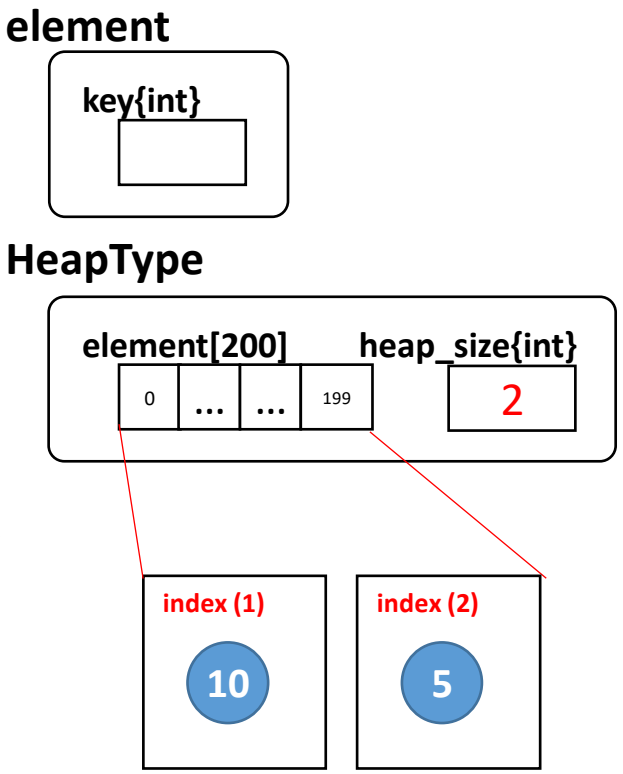
```
int main(void){
  element e1 = { 10 }, e2 = { 5 }, e3 = { 30 };
  element e4, e5, e6;
  HeapType* heap;

  heap = create();
  init(heap);

  insert_max_heap(heap, e1);
  insert_max_heap(heap, e2);
  insert_max_heap(heap, e3);

  e4 = delete_max_heap(heap);
  printf("< %d > ", e4.key);
  e5 = delete_max_heap(heap);
  printf("< %d > ", e5.key);
  e6 = delete_max_heap(heap);
  printf("< %d > ₩n", e6.key);

  free(heap);
return 0;
}
```



우선순위 큐

구현 방법 (힙, heap)

```
int main(void){
  element e1 = { 10 }, e2 = { 5 }, e3 = { 30 };
  element e4, e5, e6;
  HeapType* heap;

  heap = create();
  init(heap);

  insert_max_heap(heap, e1);
  insert_max_heap(heap, e2);
  insert_max_heap(heap, e3);

  e4 = delete_max_heap(heap);
  printf("< %d > ", e4.key);
  e5 = delete_max_heap(heap);
  printf("< %d > ", e5.key);
  e6 = delete_max_heap(heap);
  printf("< %d > ₩n", e6.key);

  free(heap);
return 0;
}
```



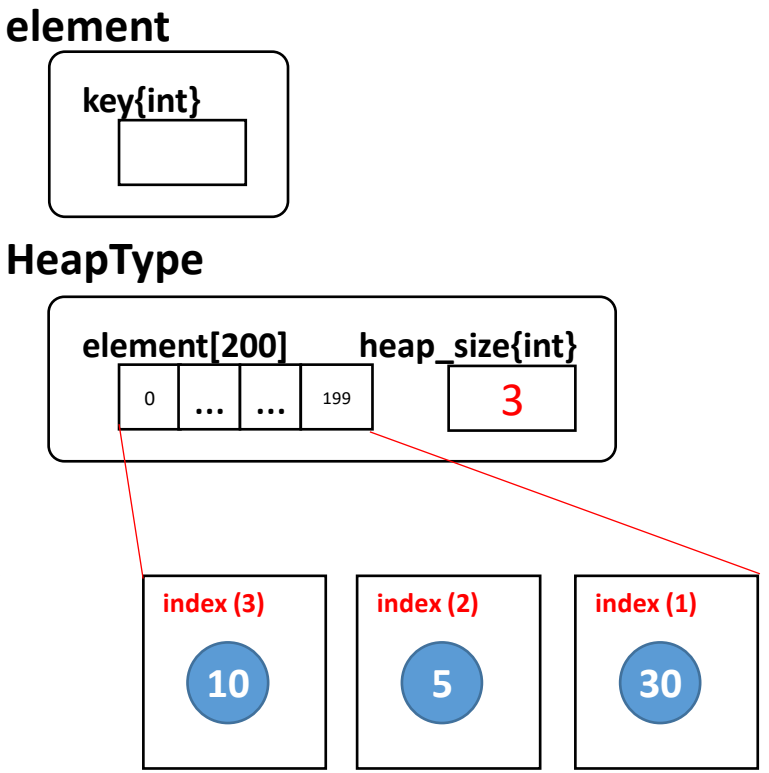

```
while ((i != 1) &&
       (item.key > h->heap[i / 2].key)) {
  h->heap[i] = h->heap[i / 2];
  i /= 2;
}
```

ex1

우선순위 큐

구현 방법 (힙, heap)

```
int main(void){
  element e1 = { 10 }, e2 = { 5 }, e3 = { 30 };
  element e4, e5, e6;
  HeapType* heap;

  heap = create();
  init(heap);

  insert_max_heap(heap, e1);
  insert_max_heap(heap, e2);
  insert_max_heap(heap, e3);

  e4 = delete_max_heap(heap);
  printf("< %d > ", e4.key);
  e5 = delete_max_heap(heap);
  printf("< %d > ", e5.key);
  e6 = delete_max_heap(heap);
  printf("< %d > ₩n", e6.key);

  free(heap);
return 0;
}
```




```
int i;
i = ++(h->heap_size);

while ((i != 1) &&
   (item.key > h->heap[i / 2].key)) {
   h->heap[i] = h->heap[i / 2];
   i /= 2;
}
h->heap[i] = item;
```

ex1

우선순위 큐

힙 정렬

- 정렬해야 할 n개의 요소들을 최대 힙에 삽입
- 한번에 하나씩 요소를 힙에서 삭제하여 저장

```
void heap_sort(element a[], int n){
  int i;
  HeapType* h;
  h = create();
  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);
  }
  free(h);
}
```

```
#define SIZE 8
int main(void){
  element list[SIZE] = { 23, 56, 11, 9,
56, 99, 27, 34 };
  heap_sort(list, SIZE);
  for (int i = 0; i < SIZE; i++) {
    printf("%d ", list[i].key);
  }
  printf("₩n");
  return 0;
}
```

우선순위 큐

허프만 코드

- 각 글자의 빈도가 알려져 있는 메시지의 내용을 압축하는데 사용
- 텍스트 압축을 위해 널리 사용되는 방법
- 출현 빈도가 낮은 문자는 많은 비트의 코드로 변환
- 출현 빈도가 많은 문자는 적은 비트의 코드로 변환

빈도수 분석 →

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

IF) 텍스트가 e, t, n, I, s의 5개의 글자로 만 이루어졌다고 가정하고 각 글자수 빈도수는 다음과 같음

글자	빈도수
e	15
t	12
n	8
i	6
s	4

우선순위 큐

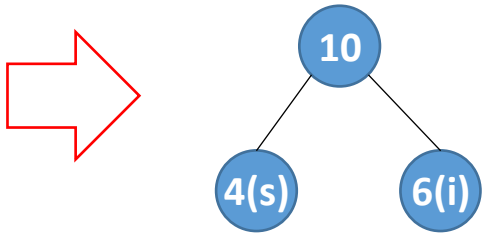
글자	빈도수
e	15
t	12
n	8
i	6
s	4

허프만 코드

내림 차순으로 정렬

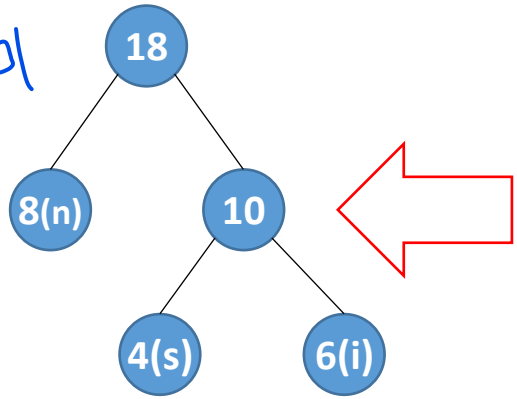
이진 트리로 묶는다 (그런 후 합을 부모 노드로 저장)

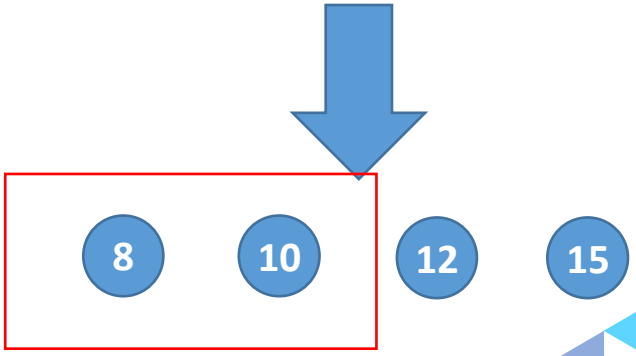
허프만은 최소힙, 꺼냈을 때 내림차순이 정확히는 아님.



**합이 된 10 포함 내림 차순으로 정렬된 다음 값을 이진 트리로 묶고 그 합을 다시 부모 노드로 저장

맨 처음 두 수가 가장 작은 두 수라는 의미

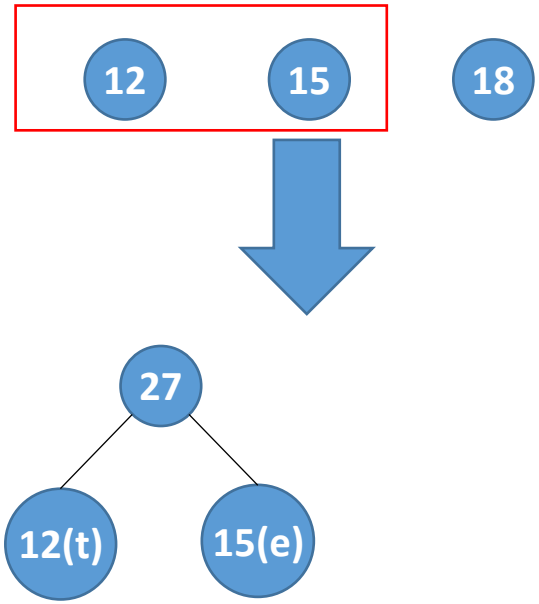




우선순위 큐

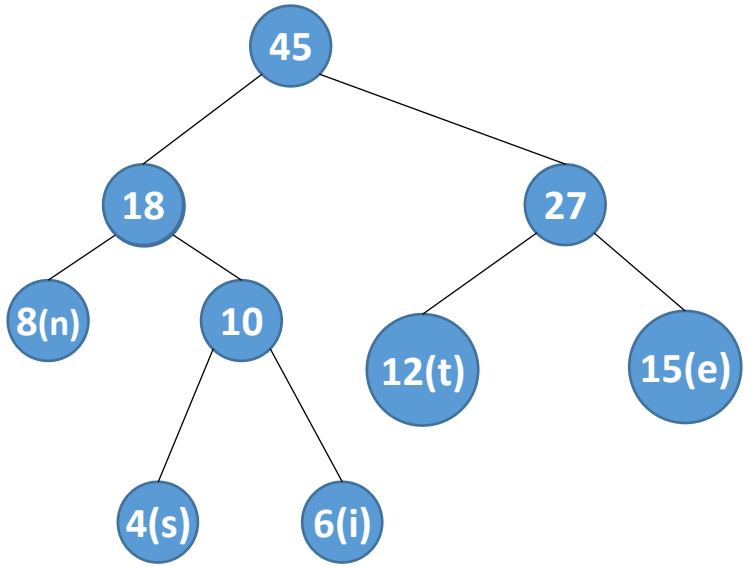
글자	빈도수
e	15
t	12
n	8
i	6
s	4

허프만 코드



내림 차순으로 정렬
* 12, 15를 이진 트리로 묶고
합을 부모 노드로 저장

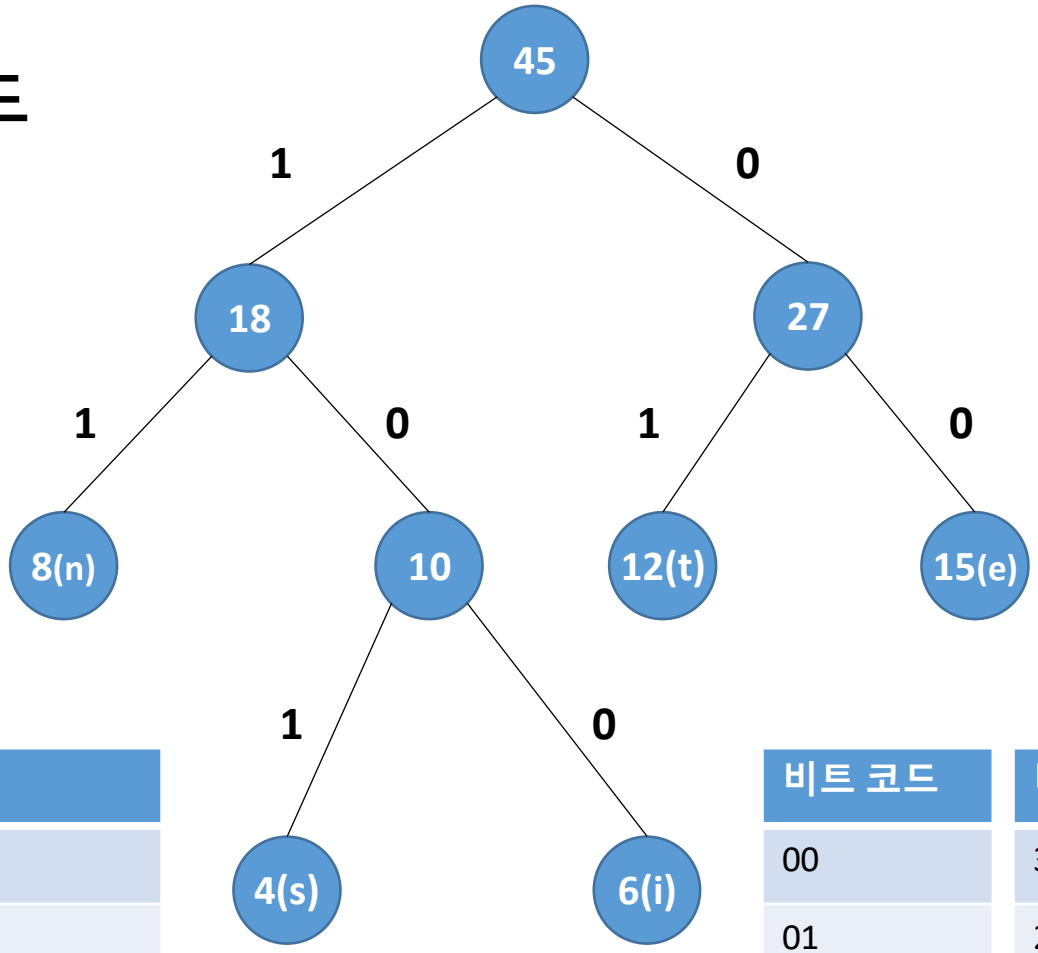
부모 모드
내림 차순으로 정렬 후
합을 부모 노드로 저장



최종 결과

우선순위 큐

허프만 코드



글자	빈도수
e	15
t	12
n	8
i	6
s	4

비트 코드	비트 수
00	30
01	24
11	16
100	18
101	12