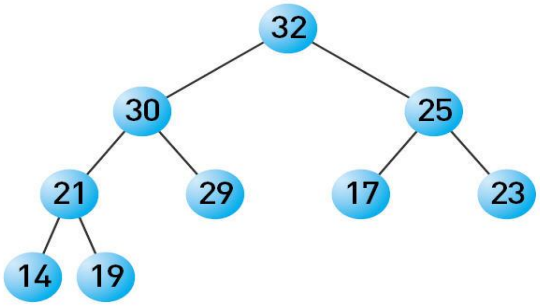
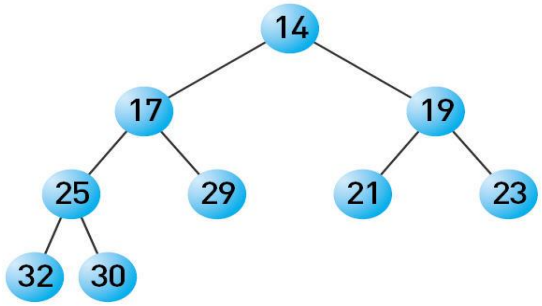
Min/Max Tree(Heap)

What is a Heap ?

- 완전 이진 트리(complete binary tree)의 특별한 형태
- 2가지 유형
  - max heap (cf. max tree) : 최대 heap
    - Key_value(부모노드) >= key_value(자식노드)
    - Key_value가 가장 큰 노드를 찾기 위한 완전 이진 트리
  - min heap (cf. min tree) : 최소 heap
    - Key_value(부모느드) <= key_value(자식노드)
    - Key_value가 가장 작은 노드를 찾기 위한 완전 이진 트리



(a) 최대 히프



(b) 최소 히프

- 우선순위 큐(priority queue)를 구성하는 유용한 방법
  - 노드의 우선 순위를 기반으로 큐를 운용
  - 수행시간을 우선순위로 한다면 가장 작은 노드부터 제거
  - 값을 우선순위로 한다면 큰 노드부터 제거 : max heap 사용
  - 구현 : 배열, linked list, heap
- 우선순위 큐의 구성방법에 따른 복잡도

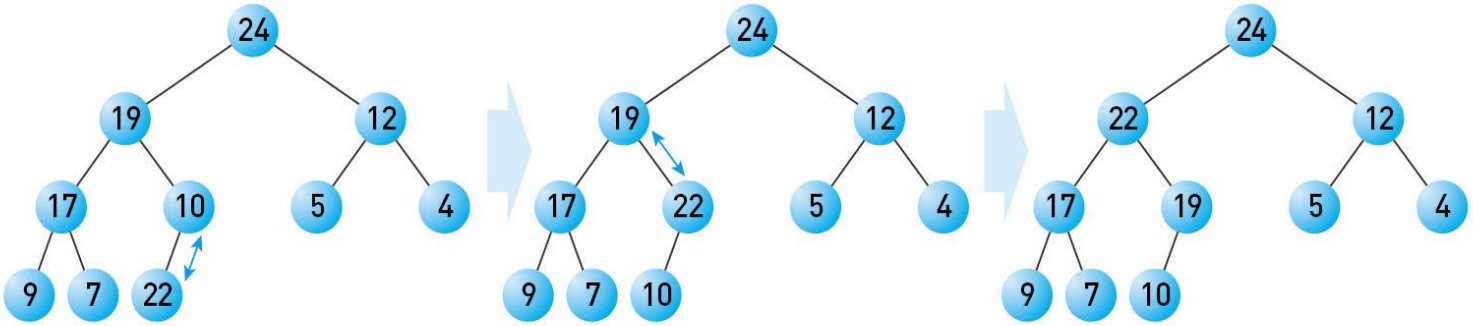
	삽입(Add)	삭제(Remove)	탐색(Retrieve)
정렬된 배열	O(n)	O(1)	O(1)
정렬되지 않은 배열	O(1)	O(n)	O(n)
정렬된 리스트	O(n)	O(1)	O(1)
정렬되지 않은 리스트	O(1)	O(n)	O(n)
히프	O(logn)	O(logn)	O(logn)

- Heap의 Abstract Data Type(ADT)
  - heap의 생성
  - insertion : 새로운 요소를 heap에 삽입
  - deletion : 가장 큰(작은) 값의 요소를 heap으로부터 삭제
  - heapFull : heap이 full이면 TRUE 리턴
  - heapEmpty : heap이 empty이면 TRUE 리턴

Heap ADT의 구현

- **노드의 삽입(insertion)**
  - 진급(Up Heap, Promotion)시키는 것과 유사
    - heap가 full이 아니면
    - 일단, 새로운 노드를 히프의 마지막 노드에 이어서 삽입
    - 새로운 노드를 부모 노드의 key 값과 비교하여
    - if (key_value(new_node) > key_value(parent_node) → 부모노드와 위치 교환의 과정을 더 이상 교환이 없을 때까지 반복

22를 삽입하는 경우의 예

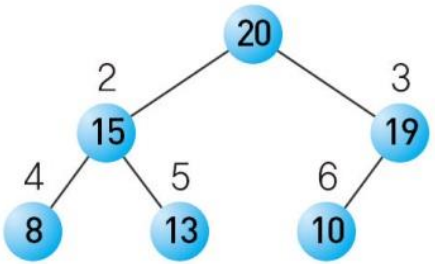


- **heap ADT를 위한 데이터 구성(배열을 사용)**

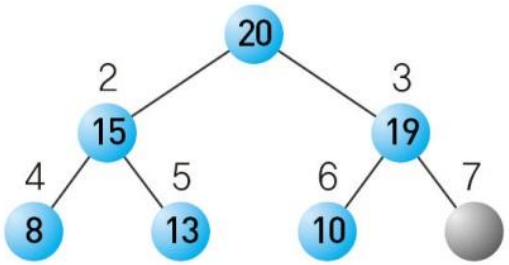
```
#define MAX_ELEMENT 100      //maximum heap size
#define heapFull(n)        (n == MAX_ELEMENT – 1)
#define heapEmpty(n)    (!n)
typedef struct {
            int   key;
            // other members if necessary
} element ;
element   heap[MAX_ELEMENT];            // 힙을 구조체 배열로 정의
int   heapSize = 0;                     // 힙을 구성하는 element의 수
```

- **void insert_maxHeap(element item) {**

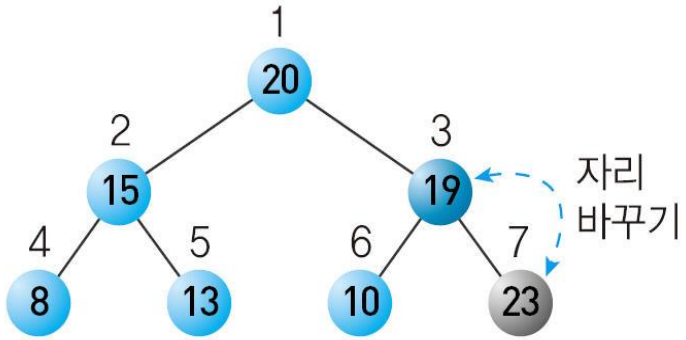
```
    int h;
    if (heapFull(heapSize)) {          // 힙이 full인가를 check
        printf("The heap is full₩n);
        exit(1);
    }
    h = ++heapSize;        // 힙 크기를 하나 증가시킨다
    while((h != 1) && (item.key > heap[h/2].key))  //부모 값보다 크면
        heap[h] = heap[h/2];
        h = h/2;
    }
    heap[h] = item;
}
```

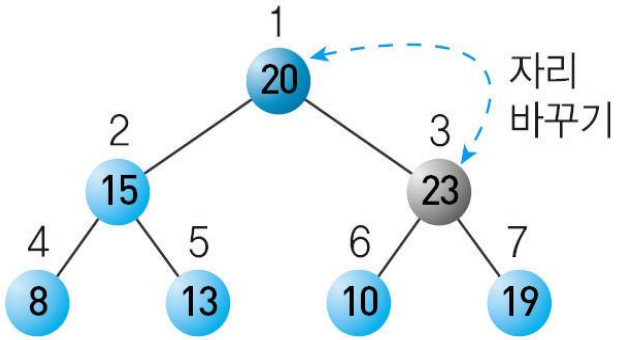


삽입전의 히프

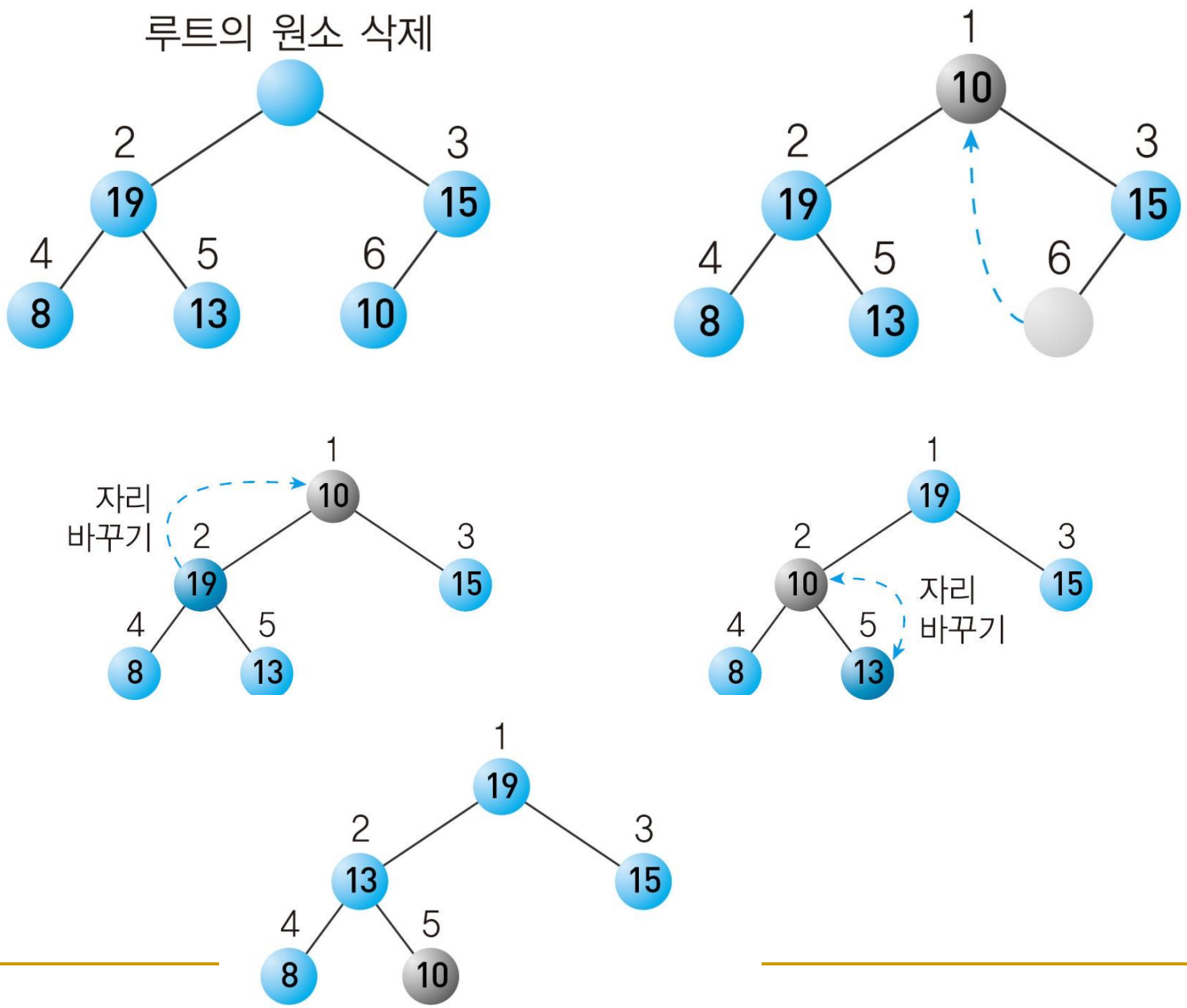


(1) 삽입할 자리 확장





- **노드의 삭제 – 가장 값이 큰 노드(root)를 삭제**
  - 강등(Down Heap, Demotion)시키는 것과 유사
    - heap이 empty가 아니면
    - 일단, root를 반환할 변수에 저장하고 마지막 노드(위치를 찾아야할 요소)의 값을 임시변수에 저장한 후 heapSize를 하나 감소
    - 마지막 노드를 root로 설정하여 아래로 내려가면서 자식 노드와 비교하면서 위치 교환
    - 더 이상 위치 교환이 안 될때까지 반복하여 위치지정하고 반환변수 (root 저장)를 리턴


루트의 원소 삭제
2
19
3
15
4
8
5
13
6
10
1
10
2
19
3
15
4
8
5
13
6
1
10
자리
바꾸기
2
19
3
15
4
8
5
13
1
19
2
10
3
15
자리
바꾸기
4
8
5
13
1
19
2
13
3
15
4
8
5
10

element delete_maxHeap(element item)

```
{
    int parent, child;
    int item, temp;

    if(heapEmpty(heapSize) {
        printf("The heap if empty₩n");
        exit(1);
    }
    item = heap[1]; // ① 루트 노드 heap[1] 저장
    temp = heap[heapSize]; //② 마지막 노드 저장
    heapSize-- ; //③ 전체 노드의 개수 하나 감소
    //④ 마지막 노드 temp의 임시 저장위치를
    parent로 하고 루트 노드(1번)에 배치
    parent = 1;
    child = 2;
    while(child <= heapSize) {
    //⑤ Parent의 자식 노드 중 큰 노드 탐색
        if((child < heapSize) &&
          (heap[child]).key < heap[child+1].key)
           child++;
        //마지막 노드 값이 자식 노드 값 이상
        이면 현재 위치가 마지막 노드의 위치
        if (temp >= heap[child].key)  break;
        //⑥ 그렇지 않으면 아래로 강등
        heap[parent] = heap[child];
        parent = child;
        child = child*2;
     }
     //⑦ 찾은 위치에 마지막 노드를 저장하고
          root 노드 반환
    heap[parent] = temp;
    //⑧ 루트 노드 반환
    return item;
}
```