11장. 우선순위 큐

- **시간에 우선순위**
  - 스택, 큐
- **일반화 된 우선순위**
  - 우선순위 큐
- **학습목표**
  - 우선순위 큐의 개념을 충분히 이해한다.
  - 배열, 연결 리스트, 이진 탐색트리로 구현할 때의 효율차이를 이해한다.
  - 힙으로 구현할 때의 삽입, 삭제 방법을 이해한다.
  - 힙 정렬의 방법과 효율을 이해한다.
  - 힙 정렬에서 힙을 구성하기 위한 두 가지 방법을 이해한다.

우선순위 큐

- **환자 치료의 예**
  - 큐: 먼저 온 사람을 먼저 치료
  - 스택: 나중 온 사람을 먼저 치료
  - 우선순위 큐: 위급한 사람을 먼저 치료

우선순위 큐

우선순위

- 시간: 스택, 큐
- 다른 가치: 우선순위 큐
- 따라서 우선순위 큐는 스택이나 큐 보다 일반적인 구조
- 키 (= 우선순위 값) 필드가 필요.
  - 스택, 큐에서는 시간에 따라 자료구조를 조직화
  - 따라서 키 필드가 불필요

추상자료형 우선순위 큐

작업

- Create: 새로운 우선순위 큐를 만들기
- Destroy: 사용되던 우선순위 큐를 파기하기
- Add: 현재 우선순위 큐에 새로운 레코드를 삽입하기
- Remove: 가장 우선순위가 높은 레코드를 삭제하기
- IsEmpty: 현재 우선순위 큐가 비어있는지 확인하기

삭제작업

- 어떤 레코드가 우선순위가 가장 높은지는 큐 자체가 알고 있으므로 호출 함수 쪽에서는 아무런 파라미터를 넘길 필요가 없다.

배열에 의한 구현

정렬된 배열

- 우선순위를 기준으로 오름차순으로 정렬
- 가장 우선순위가 큰 레코드는 항상 배열의 마지막에 위치
- 삭제함수는 마지막 레코드를 리턴, 배열 레코드의 개수를 하나 줄임
- 이 작업의 시간적 효율은 O(1)
- 삽입함수는 일단 키 값을 기준으로 이진탐색 하는데 O(lgN)
- 데이터 이동에 O(N)

Count

4

0	1	2	3		
2	5	7	10	...	

배열에 의한 구현

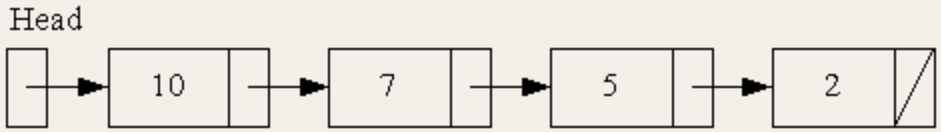
❑ 정렬 안 된 배열

- 새로운 레코드를 무조건 배열 끝에 붙임
- 삽입의 효율은 O(1)
- 삭제의 효율은 O(N). 처음부터 끝까지 뒤져야 함.
- 정렬된 배열과 정렬 안 된 배열을 사용할 때의 삽입, 삭제 효율은 서로 뒤바뀐 모습
- 삭제가 빨라야 한다면 정렬된 배열, 삽입이 빨라야 한다면 정렬 안 된 배열
- 삽입 삭제 전체를 감안하면 두 방법 모두 O(N)의 효율이다.

연결 리스트에 의한 구현

❑ 우선순위를 기준으로 내림차순으로 정렬

- 우선순위가 가장 높은 레코드를 헤드 포인터가 직접 가리킴.
- 삭제시간은 O(1)
- 삽입 함수는 키 값을 기준으로 삽입위치를 찾아야 함.
- 최악의 경우 마지막 위치. O(N)
- 배열처럼 이동은 불필요.
- 삭제와 삽입 모두를 감안한 효율은 O(1)+O(N)=O(N)이다.



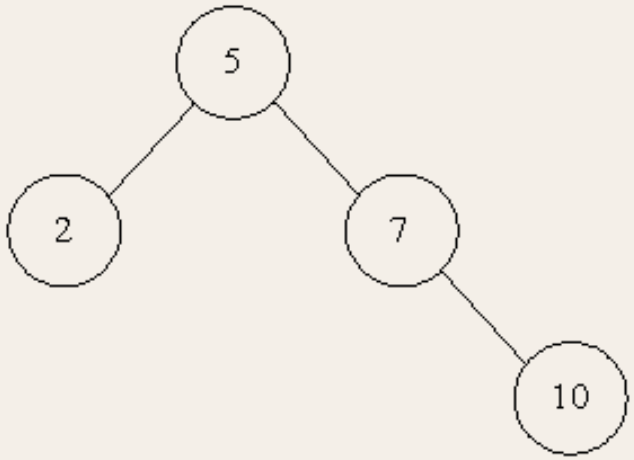
❑ 정렬 안 된 연결 리스트

- 삽입될 레코드를 가장 첫 노드로 만들면 삽입은 O(1)
- 삭제를 위해 가장 큰 레코드를 찾는데O(N)
- 삭제와 삽입 모두를 감안한 효율은 O(N) + O(1) = O(N)

이진 탐색트리에 의한 구현

- **가장 큰 키 값을 지닌 노드**
  - 는 루트로부터 출발해서 RChild를 계속 따라 감.
  - 더 이상 RChild가 없으므로 자식이 하나이거나 없는 노드
  - 상대적으로 쉬운 삭제

- **효율**
  - 삭제함수의 효율은 O(lgN). 루트로부터 리프까지 내려감.
  - 삽입 위치를 찾아 리프 노드까지 내려가는데 O(lgN)
  - 효율은 O(logN) + O(logN) = O(logN). 단 균형 트리에 한함.



우선순위 큐의 구현

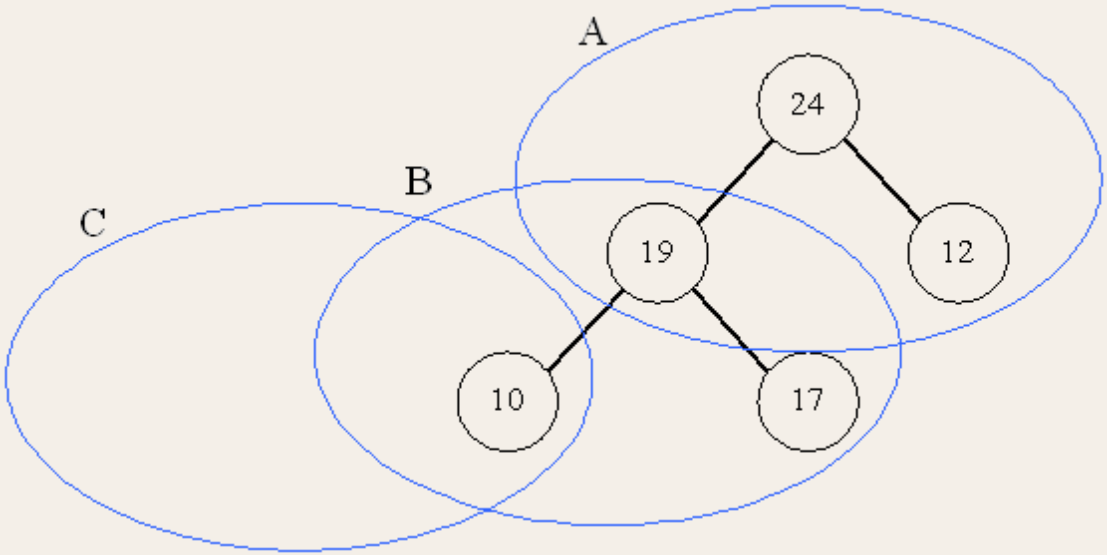
❑ 구현방법별 효율비교

	삽입(Add)	삭제(Remove)	탐색(Retrieve)
정렬된 배열	N	1	1
정렬 안 된 배열	1	N	N
정렬된 리스트	N	1	1
정렬 안 된 리스트	1	N	N
이진 탐색트리	lgN	lgN	lgN

힙

❑ 항상 완전 이진트리 모습

- 빈 트리거나,
- 루트의 키가 왼쪽자식, 오른쪽 자식의 키보다 크거나 같다. 왼쪽 자식과 오른쪽 자식 사이에는 어느 쪽 키가 크던 상관이 없다. 단, 왼쪽 자식, 오른쪽 자식을 루트로 하는 서브트리는 힙이어야 한다.



힙

- 맥스 힙(Max Heap)
  - 키 값이 큰 레코드가 우선순위가 높은 것으로 간주
  - 루트노드의 키가 가장 크다.
- 민 힙(Min Heap)
  - 은 맥스 힙과는 정 반대
  - 키 값이 작을수록 우선순위가 높다고
- 정렬
  - 이진 탐색트리가 약한 의미로 정렬. 왼쪽 보다 오른쪽이 크다.
  - 힙은 더 약한 의미로 정렬. 왼쪽과 오른쪽이 무관하다.

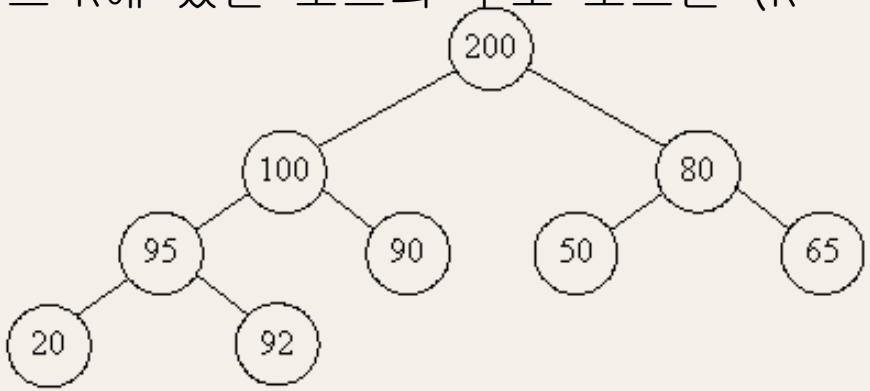
힙

- 쌓아놓은 더미

힙은 배열로 표시

- 완전 이진트리
  - 배열로 표시하는 것이 가장 효율적
  - 루트부터 시작해서 위에서 아래로, 왼쪽에서 오른쪽으로 진행
  - 노드 필기하는 순서로 트리를 순회하면서 인덱스를 부여
  - 루트노드는 배열 인덱스 0.

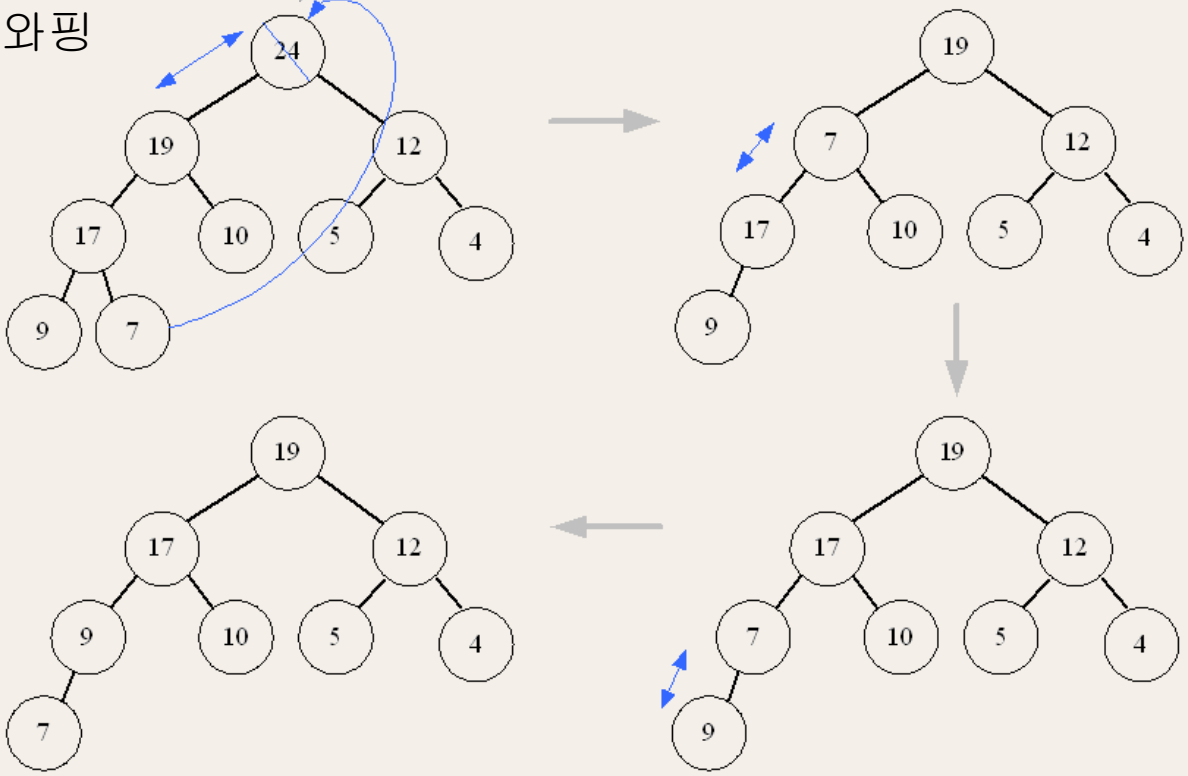
- 트리의 부모 자식 관계는 다음과 같은 배열의 인덱스 연산으로 바뀐다
  - 인덱스 K에 있는 노드의 왼쪽 자식은 (2K + 1)에 오른쪽 자식은 (2K + 2)에
  - 인덱스 K에 있는 노드의 보모 노드는 (K - 1) / 2에



0	1	2	3	4	5	6	7	8	9	10	11

힙의 삭제

- 우선순위가 가장 큰 루트노드를 삭제
  - 루트를 직접 삭제하면 이후 힙을 재구성하는 것이 복잡
  - 배열 마지막 요소를 루트노드 위치에 덧씌움.
- 다운 힙(Down Heap)
  - 힙 모습의 복원(Heap Rebuilding).
  - 루트로부터 시작해서 제자리를 찾기까지 굴러 떨어짐
  - 왼쪽 자식과 오른쪽 자식 모두를 비교해서 가장 큰 것과 루트를 스와핑



힙의 삭제

- 힙의 삭제

힙의 삭제

```
Remove (Items[ ])
{ Return Items[0]              현재의 루트노드를 되돌려 주기
  Items[0] = Items[Count-1]    트리의 마지막 노드를 루트노드 위치로 복사
  Count - -                    삭제이므로 개수를 감소
  DownHeap(Items[ ], 0)        0번 인덱스로부터 굴러 떨어지기 호출
}
DownHeap(Items[ ], Current)    Current는 현재 위치의 인덱스
{ if (Current = = Leaf)        굴러 떨어져서 리프까지 왔으면
    do nothing;                더 이상 내려갈 곳 없음
  else
  { Child = 2 * Current + 1    일단 왼쪽 자식이 오른쪽 자식보다 크다고 간주
    if (Currrent has RChild)   오른쪽 자식이 존재하면
    {  RChild = Child + 1      그 인덱스는 왼쪽자식 보다 하나 많음
       If Items[RChild] > Items[Child]  왼쪽자식 보다 오른쪽 자식이 크면
         Child = RChild;       가장 큰 것의 인덱스를 오른쪽 자식 인덱스로
    }
    if (Items[Current] < Items[Child]  현재 레코드가 자식보다 작으면
    {  Swap Items[Current] and Items[Child] 자식을 올리고, 자신은 내려감
       DownHeap(Items, Child) 내려간 위치에서 다시 재귀호출
    }
}
```

힙 삭제작업의 효율

O(lgN)

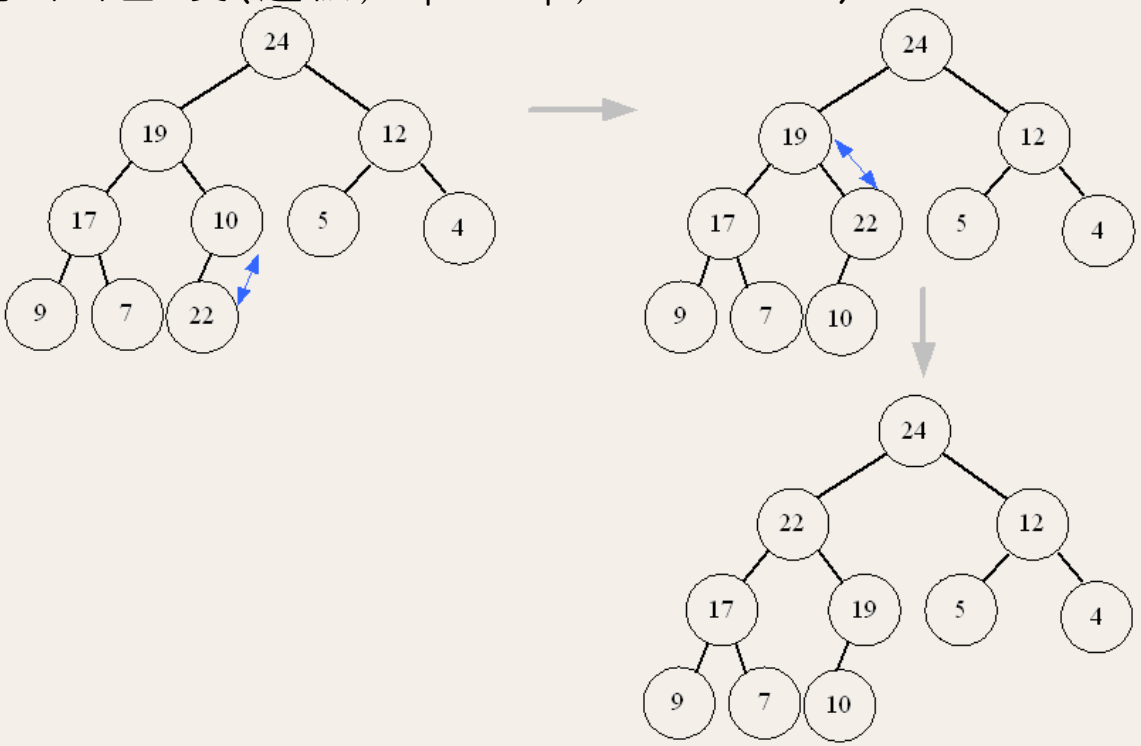
- 배열의 마지막 레코드를 처음으로 복사하는 데 O(1)
- 최악의 경우 루트로부터 리프까지 굴러 떨어짐..
- 비교의 횟수는 총 2lgN이 된다.
- 스와핑은 Temp = A, A = B, B = Temp라는 3번의 복사.
- 최악의 경우 스와핑에 의한 복사의 횟수는 3lgN

힙이 이진 탐색트리보다 유리

- 트리의 높이
- 이진 탐색트리는 최악의 경우 연결 리스트와 유사. O(N)의 효율
- 힙은 완전 이진트리로서 균형 트리
- 균형 트리의 높이는 항상 lg(N)에 가까움
- 배열로 표시되어야 하므로 최대 레코드 개수를 미리 예상해야 함

힙의 삽입

- 힙의 삭제
  - "사장자리가 비면 말단 사원을 그 자리에 앉힌 다음, 바로 아래 부하직원보다는 능력이 좋다고 판단될 때까지 강등시키는 것(降等, Down Heap, Demotion)"
- 힙의 삽입
  - "신입사원이 오면 일단 말단 자리에 앉힌 다음, 능력껏 위로 진급시키는 것(進級, Up Heap, Promotion)"



힙의 삽입

❑ 코드 11-2: 힙의 삽입작업

```
Add (Items[ ], Data)              Data는 삽입될 레코드
{  Items[Count] = Data            배열의 마지막에 삽입
   Current = Count;               그 위치의 인덱스
   Parent = (Current - 1) / 2     부모 노드의 인덱스
   while ((Current != 0) && (Items[Current] > Items[Parent])) 부모보다
    클 동안
  {    Swap Items[Current] with Items[Parent] 부모와 스왑
       Current = Parent            스왑 된 위치를 새로운 인덱스로
       Parent = (Current - 1) / 2 스왑 된 위치에서의 부모 인덱스
  }
  Count ++                        레코드 수 증가
}
```

❑ 힙의 삽입작업은 O(lgN)

- 비교의 횟수
- 삭제에서 두 개의 자식노드를 모두 비교. 삽입에서 자신과 부모 노드만 비교

이진 탐색트리와 힙

❑ 효율비교

	삽입(Insert)	삭제(Remove)	탐색(Retrieve)
이진 탐색트리	lgN	lgN	lgN
힙	lgN(보장)	lgN(보장)	1

힙 정렬

❑ 힙을 정렬 목적으로 사용

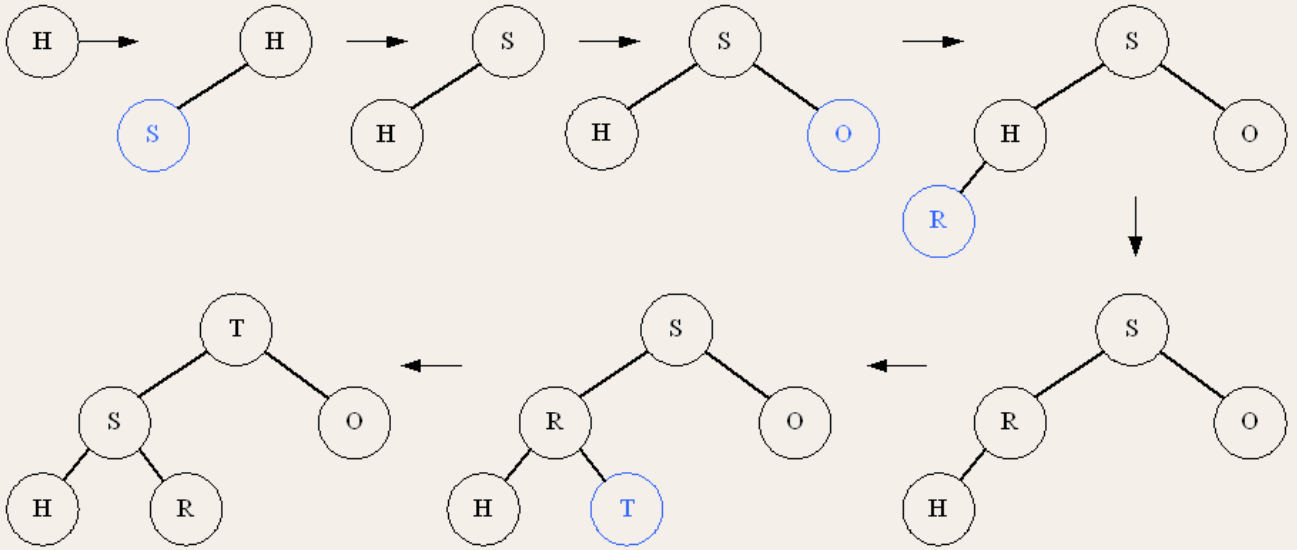
- 힙에서 Remove를 가하면 키가 제일 큰 레코드가 빠져 나옴.
- 빈 트리가 될 때까지 계속적으로 삭제를 가함.
- 빠져 나온 레코드를 순차적으로 적어나가면 그것이 바로 정렬된 결과
- 내림차순: 맥스 힙
- 오름차순: 민 힙

❑ 가장 먼저 해야 할 것은 힙 구성(Heap Building)

- 주어진 레코드로부터 일단 힙을 만들어야, 빼내 오면서 정렬이 가능
- 하향식(탑 다운) 힙 구성과 상향식(바텀 업) 힙 구성

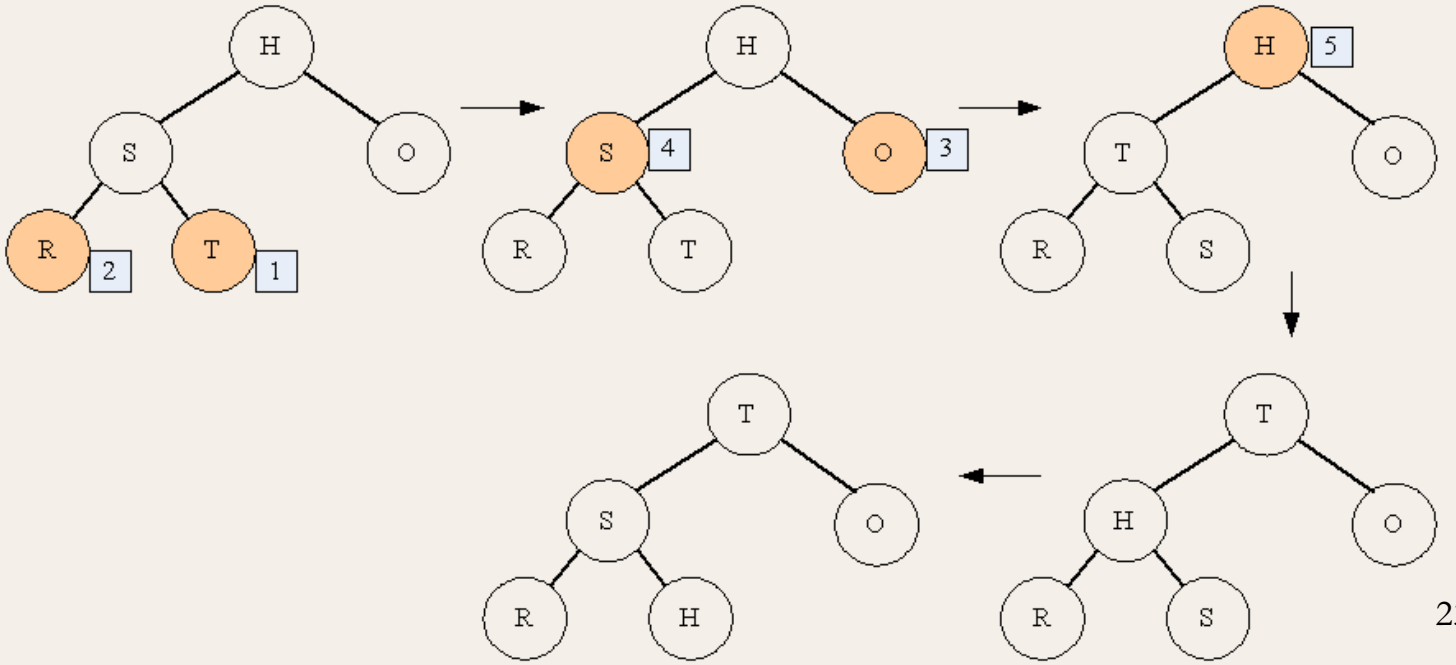
하향식 힙 구성

- 빈 힙에서 출발하여 삽입에 의해서 힙을 구성
  - 노드가 삽입될 때마다 대략 lgN의 경로를 거쳐 위로 올라감.
  - N 개가 삽입된다면 NlgN의 효율
  - 삭제 역시 하나씩 빠져 나가면서 루트로 간 레코드가 lgN 경로를 타고 내려와야 하므로 NlgN에 해당
  - 정렬의 효율은 O(NlgN) + O(NlgN) = O(NlgN)



상향식 힙 구성

- 일단 트리를 구성
  - H, S, O, R, T 순으로 노트 필기하듯이 일단 트리를 구성
- 힙 구조가 되도록 재조정
  - 가장 아래쪽부터 검사하되 오른쪽에서 왼쪽으로 진행
  - 어떤 노드가 힙의 요건을 만족시키는지 검사
  - 해당 노드와 그 아래쪽 노드 간의 키만 비교. 아래쪽으로만 스와핑
  - 상향식 힙 구성 결과와 하향식 힙 구성 결과가 다를 수 있음.



효율 비교

- 하향식 힙 구성이 O(NlgN).상향식 힙 구성은 O(N)
- 삭제단계에서는 두 방법 모두 O(NlgN). 따라서 힙 정렬은 O(NlgN)

  - 쾌속정렬은 일반적으로 O(NlgN). 합병정렬이 O(NlgN)을 보장하지만 추가의 메모리가 필요. 힙 정렬은 O(NlgN)을 보장하면서 추가의 메모리가 불필요. 그러나 내부 루프 처리시간이 쾌속정렬보다 길기 때문에 일반적으로는 쾌속정렬보다는 느림.