6주차 2차시 트리의 운행법과 균형트리

【학습목표】

1. 트리의 운행법을 이해할 수 있다.
2. 균형트리를 이해할 수 있다.

지난 강의 정리

1. 순차 탐색(sequential search)

서로 이웃한 데이터를 하나씩 차례차례로 탐색키와 비교하면서 찾는 방법이다.

* 비순서 파일의 탐색에 적합하다.
- 키 값이 순서에 관계없이 무순서로 연속해서 저장된 경우.

* n개 데이터의 순차탐색
- 탐색 실패 시 : n번 비교
- 탐색 성공 시 : 최소 1번, 평균 (n+1)/2번 ☞ O(n)

2. 이진 탐색법

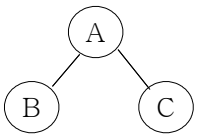
반드시 정렬되어 있는 순서 파일에서만 가능한 검색 방법으로 분할 정복(divide and conquer) 개념에 기본을 둔 방법이다. 레코드 수가 많은 큰 파일에서 효율적으로 전체 파일을 두 부분으로 나누어 찾아야 되는 레코드가 어느 부분에 있는 가를 결정한 후 그 부분으로 가서 원하는 레코드를 찾을 때까지 이 과정을 반복한다.
① 이진탐색 탐색시간은 O(logn) 이다.
② 이진 탐색법은 성능은 우수하나 삽입이나 삭제시 정렬 상태를 유지하기 위해 평균적으로 n/2 레코드를 이동해야 하므로 삽입/삭제가 많은 응용 분야에는 적절치 못하다.
③ 이진 탐색법의 단점은 삽입과 삭제에 시간이 많이 걸린다.

3. 이진탐색나무(binary search tree)

이진나무 로서 왼쪽 부분나무에는 부모보다 작거나 같은 킷값이, 오른쪽 부분나무에는 크거나 같은 키 값이 오도록 조직된 형태.
① 이진탐색나무의 장점은 삽입과 삭제가 용이하다. 데이터가 고정되지 않고 삽입과 삭제가 빈번한 경우에 적합하다.
② 탐색 시간은 최악의 경우(경사 나무가 생성되는 경우)에는 O(n), 평균 O(logn)이 된다.

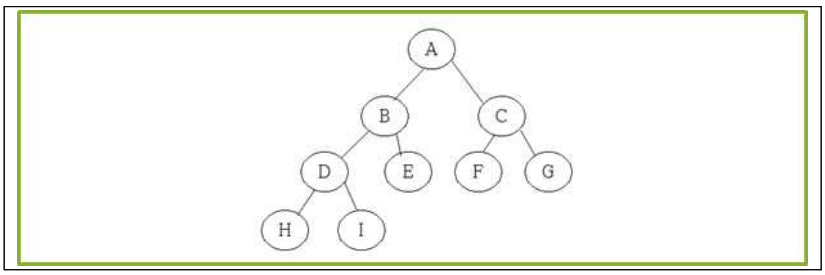
학습내용1 : 트리의 운행법

이진트리의 운행법은 다음 세 가지가 있다.



- **Preorder 운행 : Root → Left → Right 순서대로 운행한다. (A, B, C)**
- **Inorder 운행 : Left → Root → Right 순서대로 운행한다. (B, A, C)**
- **Postorder 운행 : Left → Right → Root 순서대로 운행한다. (B, C, A)**

1. 이진트리의 운행법

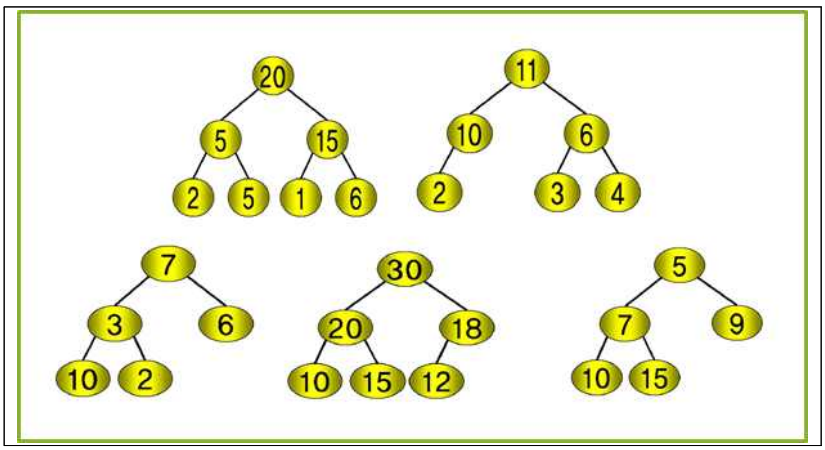


Preorder 운행 : Root → Left → Right 순서대로 운행한다.(A-B-D-H-I-E-C-F-G)
Inorder 운행 : Left → Root → Right 순서대로 운행한다.(H-D-I-B-E-A-F-C-G)
Postorder 운행 : Left → Right → Root 순서대로 운행한다.(H-I-D-E-B-F-G-C-A)

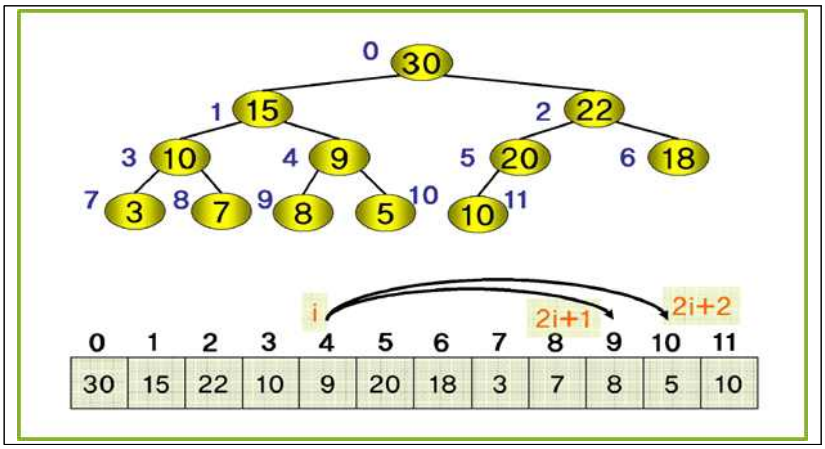
2. 히프

- 완전 이진나무
- 부모노드는 자식노드 보다 커야한다.(Max-heap), 부모노드는 자식노드 보다 작아야 한다.(Min-hep)

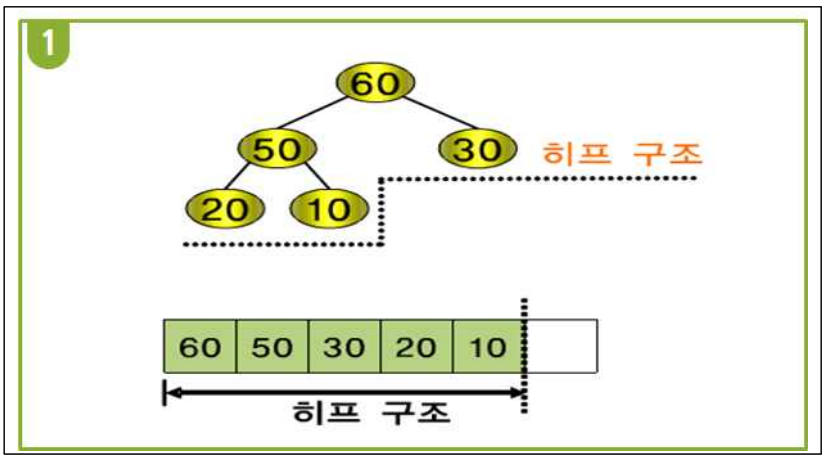
아래 이진나무에서 히프인 것은?

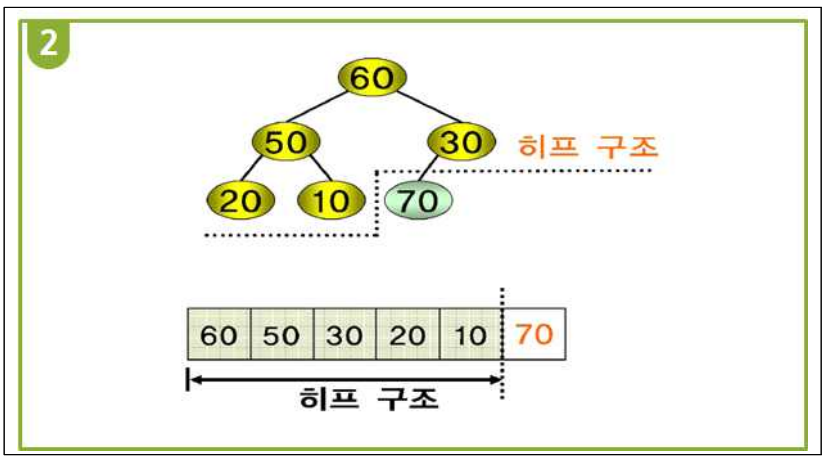


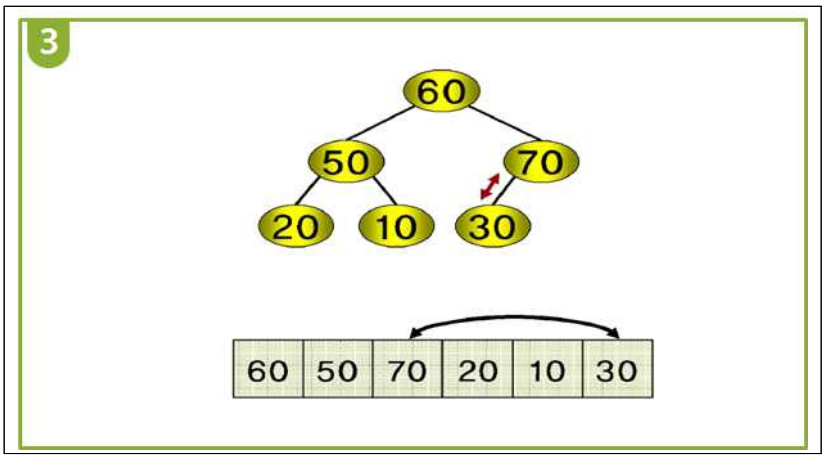
* 히프의 구현

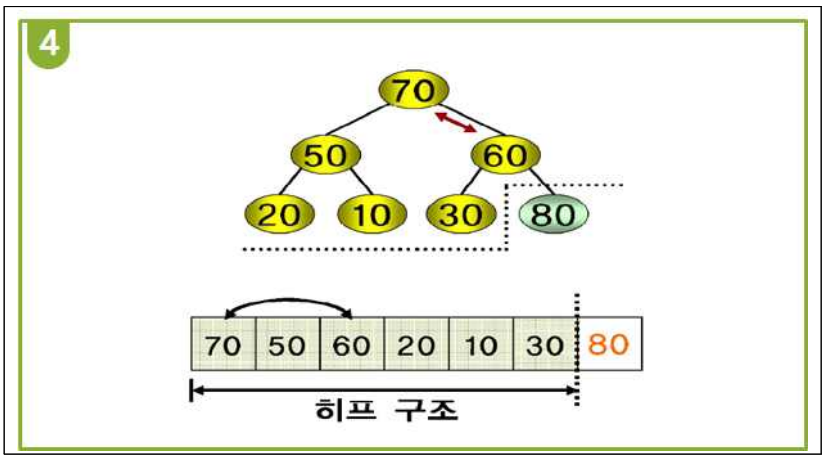

0 30
1 15
2 22
3 10
4 9
5 20
6 18
7 3
8 7
9 8
10 5
11 10
i
2i+1
2i+2
0 1 2 3 4 5 6 7 8 9 10 11
30 15 22 10 9 20 18 3 7 8 5 10

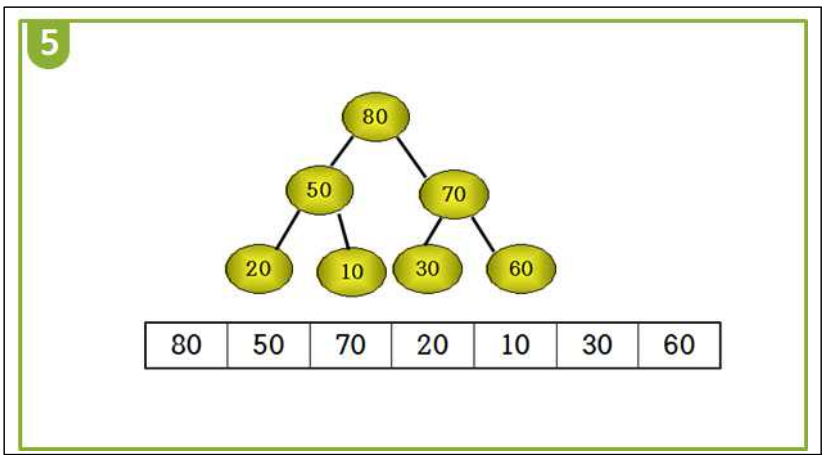
* 삽입 과정


1
60
50
30
히프 구조
20
10
60 50 30 20 10
히프 구조


2
60
50
30
히프 구조
20
10
70
60 50 30 20 10 70
히프 구조

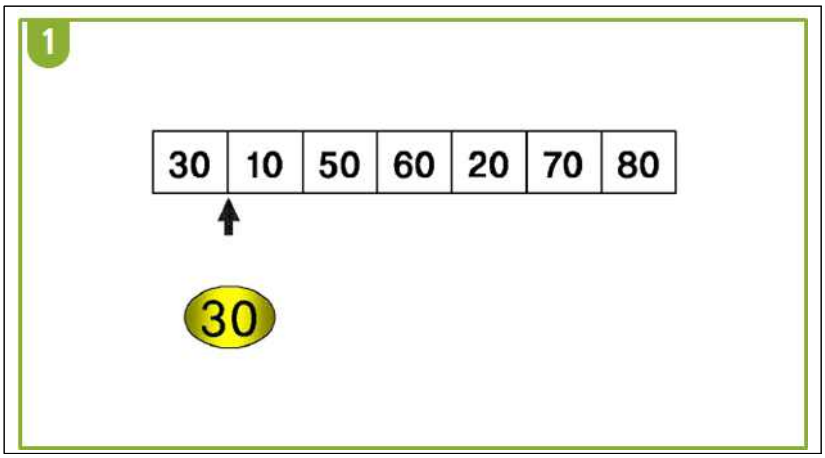

3
60
50
70
20
10
30
60 50 70 20 10 30

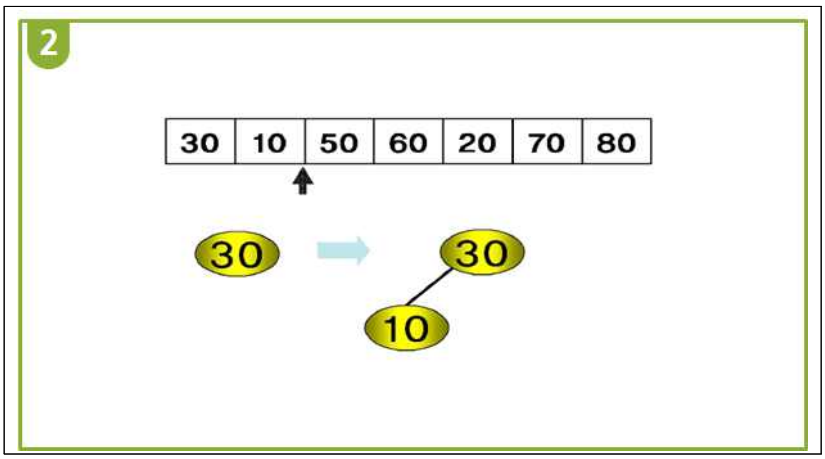

4
70
50
60
20
10
30
80
70 50 60 20 10 30 80
히프 구조

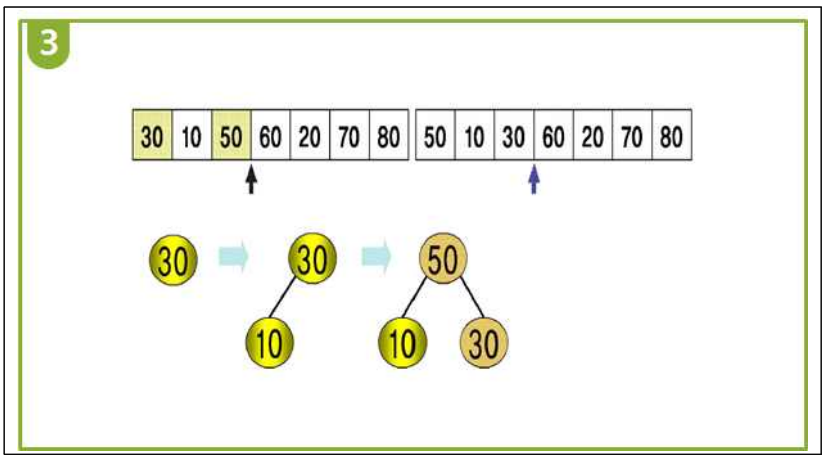

5
80
50
70
20
10
30
60
80 50 70 20 10 30 60

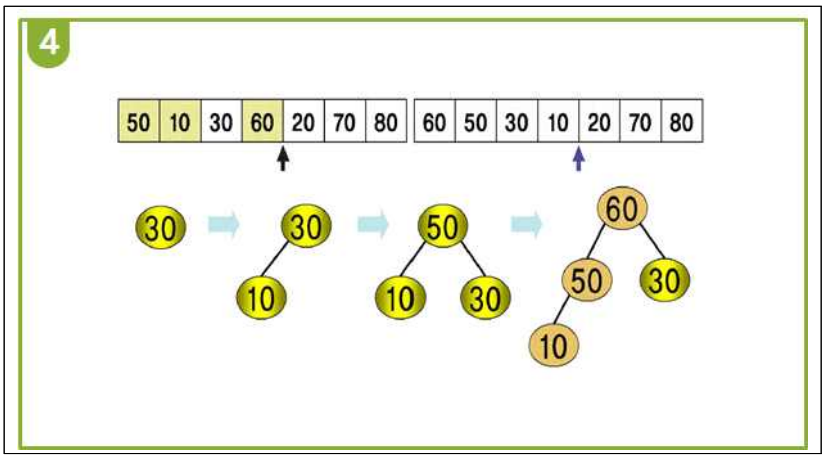
3. 히프의 변환

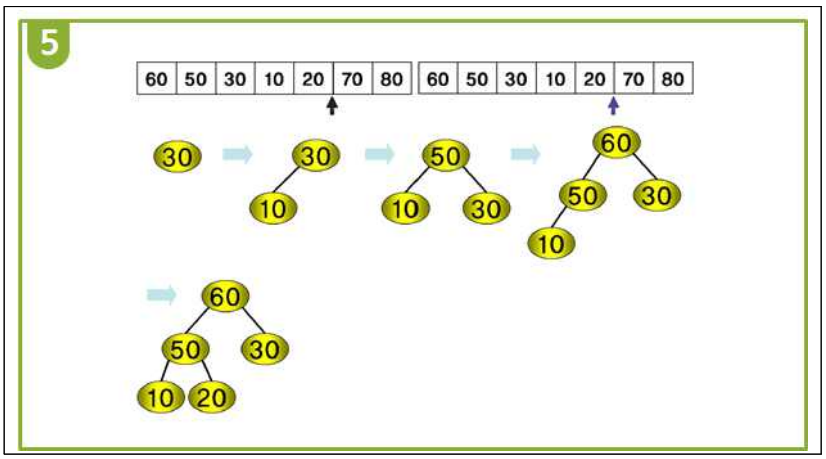
* 상향식

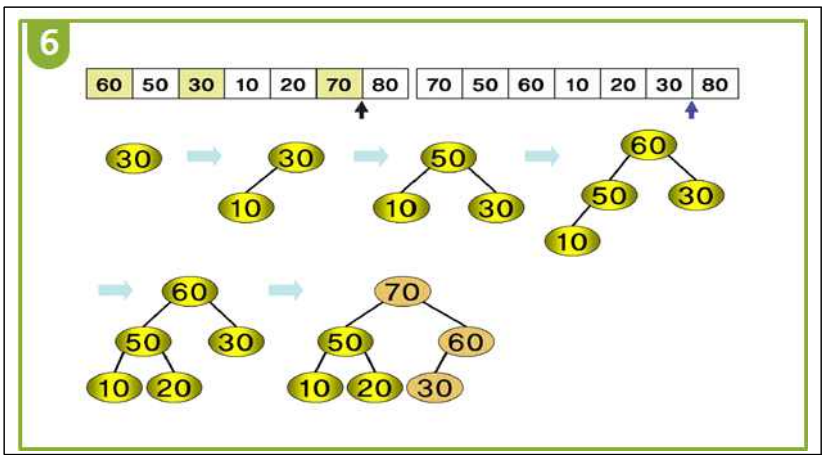

1
30 10 50 60 20 70 80
30

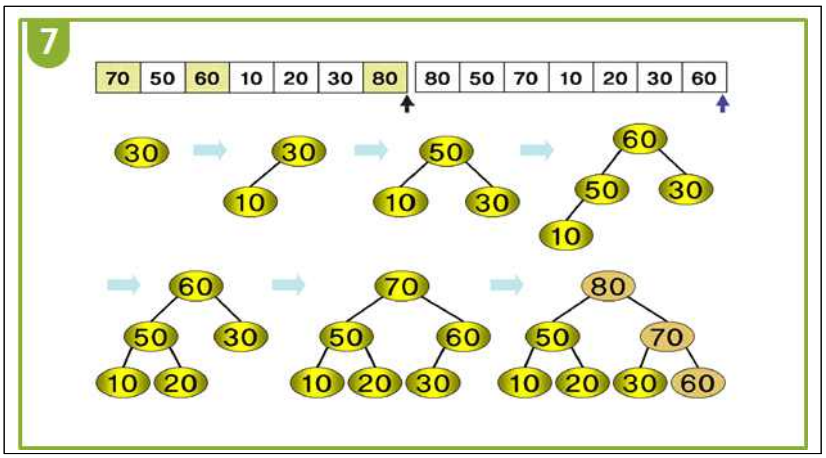

2
30 10 50 60 20 70 80
30
30
10


3
30 10 50 60 20 70 80 50 10 30 60 20 70 80
30
30
50
10
10
30

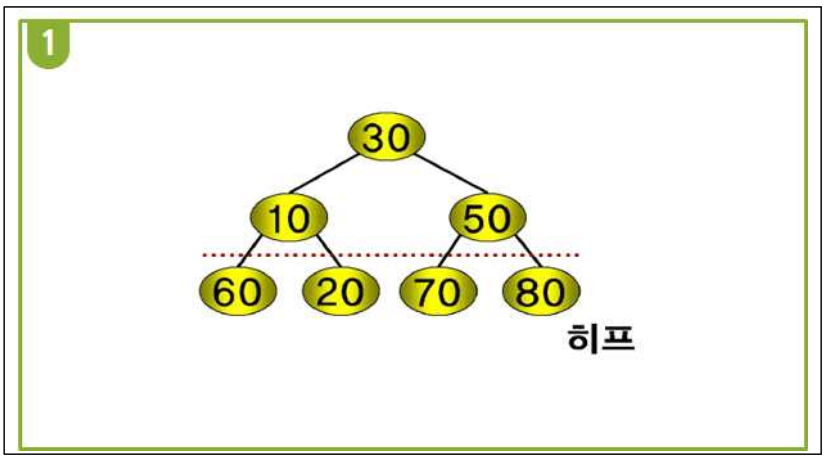

4
50 10 30 60 20 70 80
60 50 30 10 20 70 80
30
30
10
50
10
30
60
50
30
10

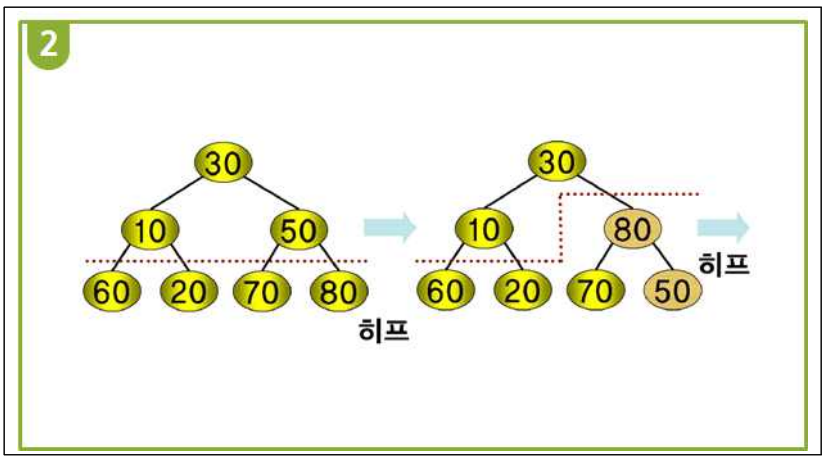

5
60 50 30 10 20 70 80
60 50 30 10 20 70 80
30
30
10
50
10
30
60
50
30
10
60
50
30
10
20

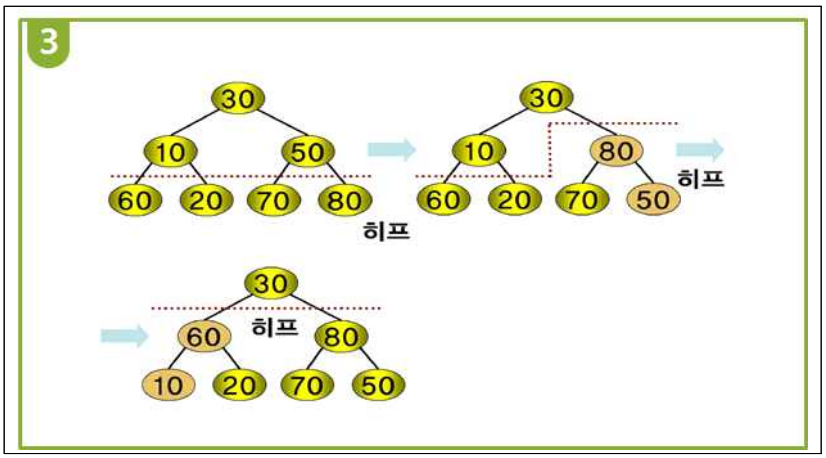

6
60 50 30 10 20 70 80
70 50 60 10 20 30 80
30
30
10
50
10
30
60
50
30
10
60
50
30
10
20
70
50
60
10
20
30

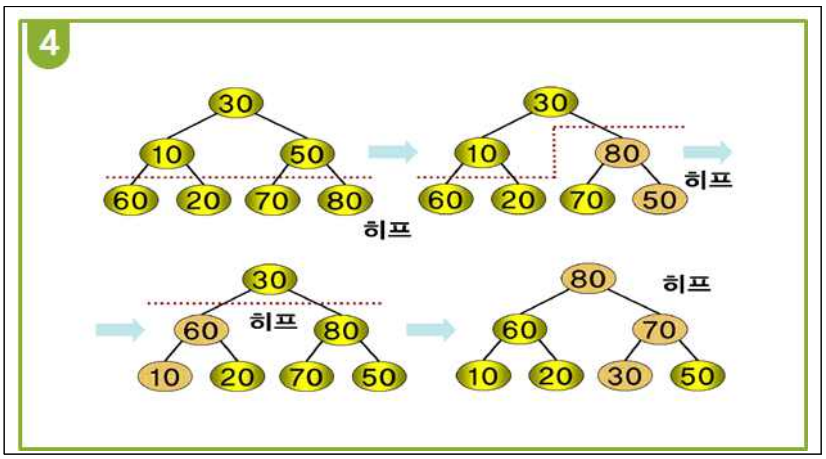

7
70 50 60 10 20 30 80 80 50 70 10 20 30 60

* 하향식


1
30
10
50
60
20
70
80
히프


2
히프
히프





학습내용2 : 균형트리

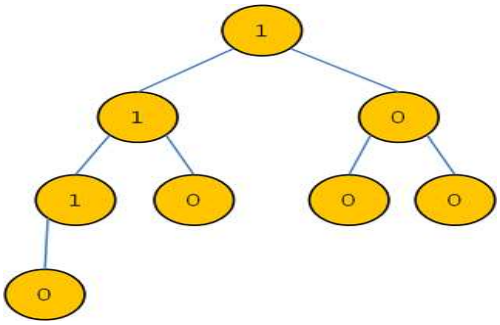
1. 균형 나무

균형나무란 좌, 우 부분나무의 높이가 같은 나무를 말하며, AVL 트리, 2-3-4 트리, 흑적나무가 있다.

2. AVL 트리

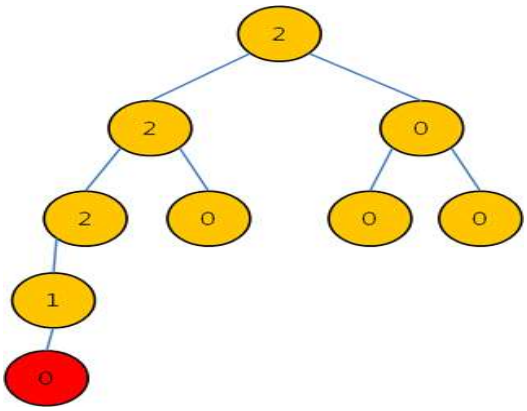
AVL 트리란 바로 균형이 갖춰진 이진트리를 의미하며, 균형이 갖춰져 있다는 말은 거의 완전 이진트리를 갖춘다는 것을 의미한다. 완전이진트리는 검색시 O(log n)에 해당하는 검색속도를 유지한다. 이에 따라 AVL 트리는 최적의 검색 속도를 보장하기 때문에 삽입과 삭제가 적은 경우에 아주 효율적이다.
AVL 트리에서 가장 중요한 개념은 높이 차이를 나타내는 균형인수(Balance factor)이다.

균형 인수란 ?
간단하게 말하면 특정 노드에서 자식 노드들의 높이 차이를 말한다. 즉, 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이의 차이를 현재 노드의 균형인수라고 한다.
균형인수 = |왼쪽자식높이 - 오른쪽자식높이|



위 트리를 보면 맨 밑단의 단말노드의 균형인수는 0인 것을 알 수 있다. 왜냐하면 자식들이 없으니까 높이 차이는 0 이다.
이로써 제일 왼쪽의 단말노드의 부모의 균형인수는 1이 되는 이유가 설명이 된다. 이렇게 전체적으로 ±1을 이하를 이루는 트리를 AVL 트리라고 한다.

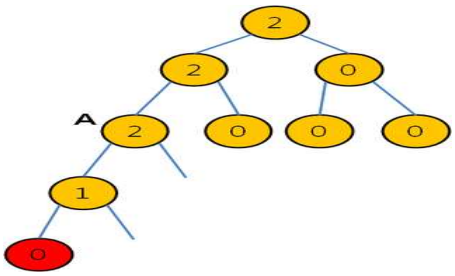
만약 제일 왼쪽에 또 하나의 단말노드를 붙여보자.



그렇다면 이처럼 균형인수가 2가되어 AVL 트리 조건이 깨져버리게 된다.
이 때 AVL 트리 구성을 위해 균형을 맞추기 위해 재구축해야 한다.
일반적으로 AVL 트리에서는 4가지 형태의 삽입으로 AVL 트리가 조건이 깨져버린다.

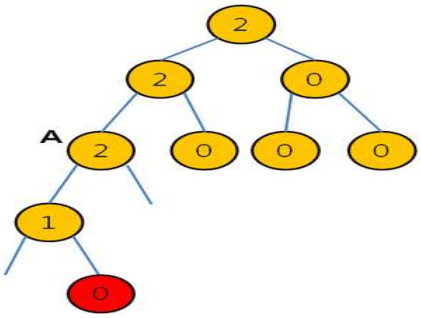
* LL 형식, LR 형식, RR 형식, RL 형식

LL 형식



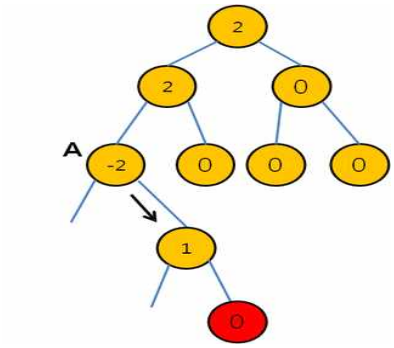
- A의 왼쪽 자식, 그 자식의 왼쪽으로 넣는 경우

LR 형식



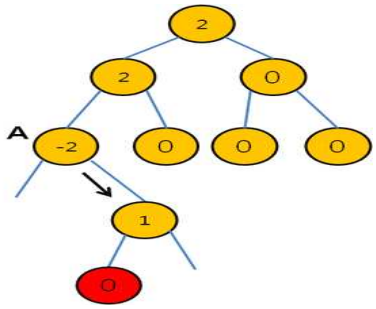
- A의 왼쪽 자식, 그 자식의 오른쪽으로 넣는 경우

RR 형식



- A의 오른쪽 자식, 그 자식의 오른쪽으로 넣는 경우

RL 형식



- A의 오른쪽 자식, 그 자식의 왼쪽으로 넣는 경우

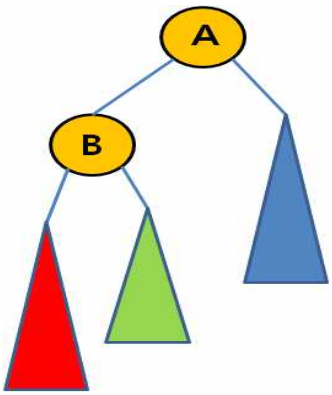
* 삽입시 AVL 트리의 조건이 깨지는 4가지 경우
1) LL 형식 : A의 왼쪽(L) 자식의 왼쪽(L)에넣는 경우
2) LR 형식 : A의 왼쪽(L) 자식의 오른쪽(R)에 넣는 경우
3) RR 형식 : A의 오른쪽(R) 자식의 오른쪽(R)에 넣는 경우
4) RL 형식 : A의 오른쪽(R) 자식의 왼쪽(L)에 넣는 경우
여기서 보아야할 것은 LL의 대칭은 RR 이고 LR의 대칭은 RL이라는 점이다.

이러한 4가지 경우에 따라 트리를 재구축하는 방법

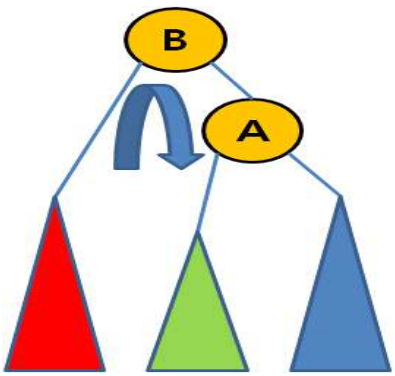
일단 AVL 트리는 균형인수가 ±1 이하인 트리이므로 삽입시 균형인수가 틀어지기 시작한 노드를 중심으로 재구축을 진행한다.

* 우선 LL 형식을 먼저 보자.

일단 아래를 보며 신규노드를 삽입했을 경우 빨간색 자식노드에 삽입되어 불균형이 발생하게 된다.



<LL 형식>



<재구축 : 오른쪽 회전>

LL 형식은 오른쪽 회전(rotation)이라는 작업을 통해 재구축 한다.

알고리즘은 다음과 같다.

roatateLL(A)
  B = A의 왼쪽 자식
  A 왼쪽 자식 = B의 오른쪽 자식
  B의 오른쪽 자식 = A
return B

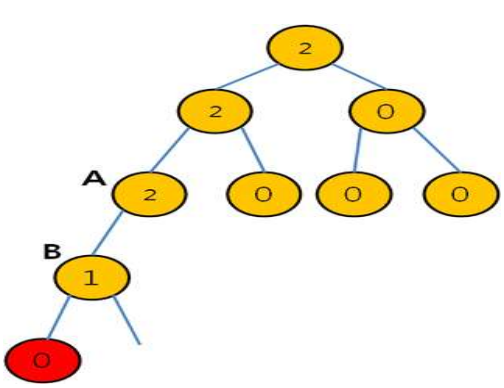
이를 코드로 표현하면 다음과 같다.

```
065: Node* rotateLL(Node *node)
066: {
067:        Node *child=node->left;
068:        node->left=child->right;
069:        child->right=node;
070:        return child;
071: }
```

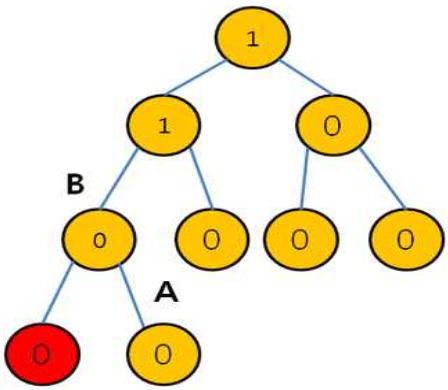
왜 B를 리턴하는가? 원래의 A 자리에 B가 결국에는 올라왔기 때문에 이에 포인터를 리턴하여 이 위치를 저장하기 위한 것이다.

- 신규 A' = 기존의 A를 대신하는 B 포인터

이처럼 포인터의 위치 변경만을 통해 LL 형식은 재구축 할 수 있다.



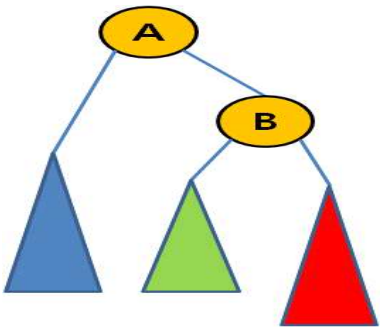
<AVL 트리 깨짐 : LL 형식>



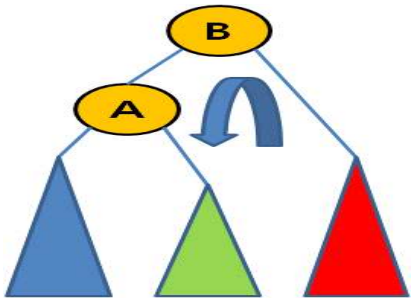
<재구축 : LL 오른쪽 회전>

여기서 신규노드를 삽입했을 경우 부모 쪽으로 타고 올라가다보면 A 노드에서 균형인수 2로 불균형이 발생했다. 재구축의 기준은 삽입한 노드로부터 조상 쪽으로 올라가며 만나는 위치다. A를 토대로 LL 회전을 발생하여 재구축한다.

* RR의 형식에 대한 회전
RR형식은 LL의 대칭 개념이라고 했다. LL이 왼쪽의 왼쪽에 집어넣어진 것이라면 RR은 오른쪽의 오른쪽에 넣어진 것이다.



<AVL 트리 깨짐 : RR 형식>



<재구축 : RR 왼쪽 회전>

RR회전 역시 포인터의 위치만 바꾸면 된다.

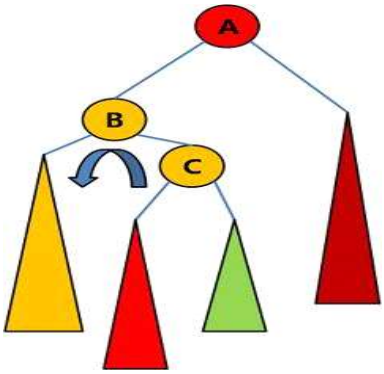
알고리즘은 다음과 같다.
roatateRR(A)
  B = A의 오른쪽 자식
  A 오른쪽 자식 = B의 왼쪽 자식
  B의 왼쪽 자식 = A
return B

이를 코드로 표현하면 다음과 같다.
```
073: Node* rotateRR(Node *node)
074: {
075:        Node *child=node->right;
076:        node->right=child->left;
077:        child->left=node;
078:        return child;
079: }
```

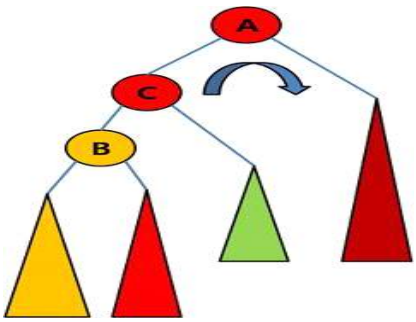
RR은 LL의 대칭이다. 순서만 바뀐 것으로 보일 것이다.

이제 LR 형식과 RL 형식에 대해 알아보자. 마찬가지로 LR과 RL은 대칭이다.
LR만 제대로 이해한다면 RL은 금방 이해 될 것이다.

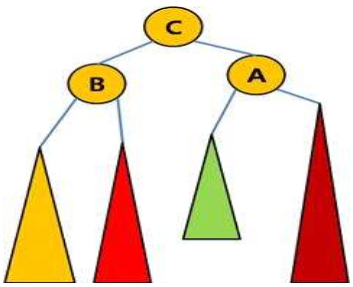
우선 LR 형식에 대해 알아보자.



LR은 왼쪽 자식트리의 오른쪽 트리에 삽입되는 형식이다.
즉, A의 노드 왼쪽 자식 B의 오른쪽 자식 C에 삽입되는 순간 AVL 트리의 구조가 깨진다.
위의 트리를 보는 것과 같이 일단 B 에서 RR 회전을 해주면 조금 해결이 될 것 같다.



회전을 시켰는데 C와 A의 균형인수가 ±2가 되면 다시 A를 기준으로 LL 회전을 해주어야 한다.



그렇다면 이렇게 AVL 트리 구조완성 된다.
LR 회전은 다시 말해서 RR 회전과 LL회전으로 이루어진다.

좀 더 다시 말하면 LR 회전의 기준이 되는 노드(A노드) 일 때
왼쪽 자식을 기준(B노드)으로 RR 회전,
본인 노드를 기준(A노드)으로 LL회전을 한다.

이것을 알고리즘으로 구현하면
roatateLR(A)
  B = A의 왼쪽 자식
  A 왼쪽 자식 = rotateRR(B)
return rotateLL(A)

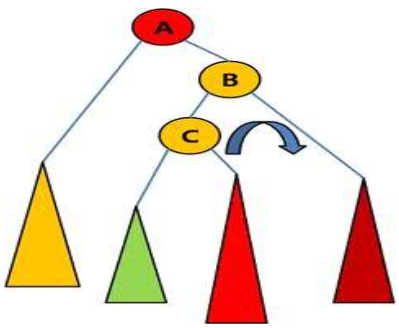
이렇게 된다. 일단 B를 기준으로 RR 회전을 돌리면 원래 B의 자리로 오게 되는 노드를 A의 왼쪽 자식으로 놓는다.
그리고 A를 기준으로 LL 회전을 시키면 원래의 A 자리에도 다른 노드가 오게 될 수 있으니
rotateLR 함수를 호출하는 곳으로 return 시켜서 마찬가지로 엮어줘야 한다.

코드는 아래와 같다.

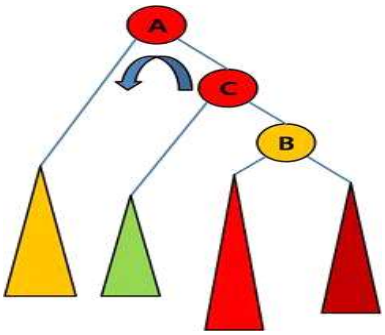
```
081: Node* rotateLR(Node *node)
082: {
083:         Node *child=node->left;
084:         node->left=rotateRR(child);
085:         return rotateLL(node);
086: }
```

RL 회전은 LR 회전의 대칭이라고 했다.

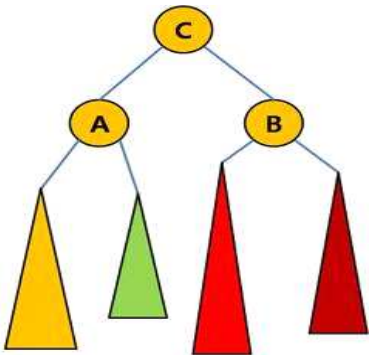
일단 RL 형식은 현재 노드(A)의 오른쪽 서브트리(B)의 왼쪽 서브트리(C)에 삽입되는 형태라고 했다.
아래처럼 A노드의 오른쪽(B)의 왼쪽(C)에 삽입되어 AVL트리가 깨지는 것을 의미한다.



LR 회전과 마찬가지로 B를 기준으로 LL회전(오른쪽)을 한다.



그러나 그래도 AVL 트리가 깨진 것을 확인 할 수 있을 것이다. 이제는 A를 기준으로 RR회전(왼쪽)을 한다.



이처럼 균형이 잡힌다. 이것을 알고리즘으로 표현하면 아래와 같다.

roatateLR(A)
  B = A의 오른쪽 자식
  A 오른쪽 자식 = rotateRR(B)
return rotateLL(A)
코드로 표현하면 다음과 같다.

```
088: Node* rotateRL(Node *node)
089: {
090:        Node *child=node->right;
091:        node->right=rotateLL(child);
092:        return rotateRR(node);
093: }
```

【학습정리】

1. 이진트리의 운행법은 다음 세 가지가 있다.
- Preorder 운행 : Root → Left → Right 순서대로 운행한다. (A, B, C)
- Inorder 운행 : Left → Root → Right 순서대로 운행한다. (B, A, C)
- Postorder 운행 : Left → Right → Root 순서대로 운행한다. (B, C, A)

2. 히프
완전 이진나무이며 부모노드는 자식노드 보다 커야한다.(Max-heap), 부모노드는 자식노드 보다 작아야 한다.(Min-hep)

3. 균형 나무
균형나무란 좌, 우 부분나무의 높이가 같은 나무를 말하며, AVL 트리, 2-3-4 트리, 흑적나무가 있다.

4. AVL 트리
AVL 트리란 바로 균형이 갖춰진 이진트리를 의미하며, 균형이 갖춰져 있다는 말은 거의 완전 이진트리를 갖춘다는 것을 의미한다. 완전이진트리는 검색시 O(log n)에 해당하는 검색속도를 유지한다. 이에 따라 AVL 트리는 최적의 검색 속도를 보장하기 때문에 삽입과 삭제가 적은 경우에 아주 효율적이다.
AVL 트리에서 가장 중요한 개념은 높이 차이를 나타내는 균형인수(Balance factor)이다.
균형 인수란 ?
간단하게 말하면 특정 노드에서 자식 노드들의 높이 차이를 말한다. 즉, 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이의 차이를 현재 노드의 균형인수라고 한다.
균형인수 = |왼쪽자식높이 - 오른쪽자식높이|