8장 이원 탐색 트리

순서

8.1 이원 탐색 트리 (binary search tree)

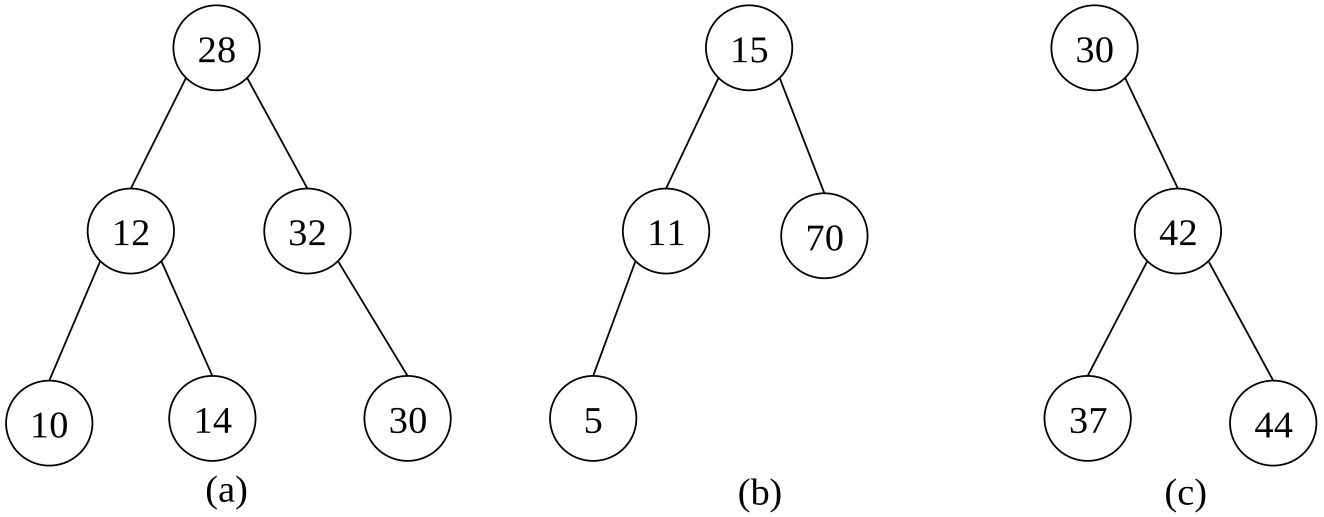
이원 탐색 트리 (binary search tree)

- 임의의 키를 가진 원소를 삽입, 삭제, 검색하는데 효율적인 자료구조
- 모든 연산은 키 값을 기초로 실행

- 정의 : 이원 탐색 트리(binary search tree:BST)
  - 이진 트리
  - 공백이 아니면 다음 성질을 만족
    1. 모든 원소는 상이한 키 값을 갖는다.
    2. 왼쪽 서브트리 원소들의 키 값은 루트 노드의 키 값보다 작다.
    3. 오른쪽 서브트리 원소들의 키 값은 루트 노드의 키 값보다 크다.
    4. 왼쪽 서브트리와 오른쪽 서브트리는 모두 이원 탐색 트리이다.

이원 탐색 트리 예

- 그림 (a): 이원 탐색 트리가 아님
- 그림 (b), (c): 이원 탐색 트리임



이원 탐색 트리에서의 탐색

- 키 값이 x인 원소를 탐색
  - 이원 탐색 트리가 공백이면, 실패로 종료
  - 루트의 키 값이 x와 같으면, 탐색은 성공으로 종료
  - 키 값 x 가 루트의 키 값보다 작으면, 루트의 왼쪽 서브트리만 다시 탐색
  - 키 값 x가 루트의 키 값보다 크면, 루트의 오른쪽 서브트리만 다시 탐색

- 연결 리스트로 표현
  - 노드 구조 :

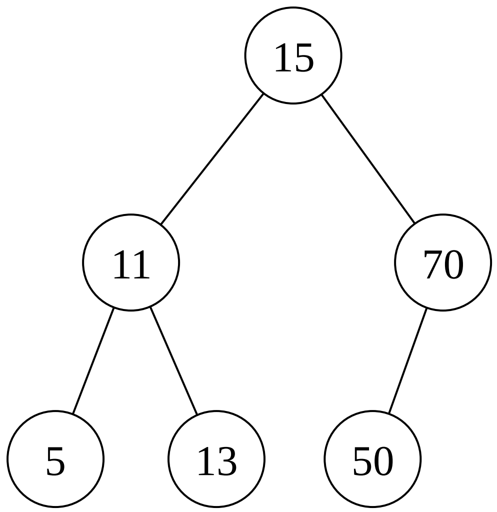
left	key	right

이원 탐색 트리에서의 탐색 알고리즘

```
searchBST(B, x)
        // B는 이원 탐색 트리
        // x는 탐색 키 값
    p ← B;
    if (p = null) then                      // 공백 이진 트리로 실패
        return null;
    if (p.key = x) then                     // 탐색 성공
        return p;
    if (p.key < x) then                     //  오른쪽 서브트리 탐색
        return searchBST(p.right, x);
    else return searchBST(p.left, x);       // 왼쪽 서브트리 탐색
end searchBST()
```

이원 탐색 트리에서의 삽입 (1)

- 키 값이 x인 새로운 원소를 삽입
  - x를 키 값으로 가진 원소가 있는가를 탐색
  - 탐색이 실패하면, 탐색이 실패로 끝난 위치에 삽입
- 예 : 키 값 13, 50의 삽입 과정



이원 탐색 트리에서의 삽입 (2)

```
// B는 이원 탐색 트리, x는 삽입할 원소 키 값
insertBST(B, x)
     p ← B;
     while (p ≠ null) do {
         // 삽입하려는 키 값을 가진 노드가 이미 있는지 검사
         if (x = p.key) then return;
         q ← p;                         // q는 p의 부모 노드를 지시
         if (x < p.key) then p ← p.left;
         else p ← p.right;
     }
     newNode ← getNode();               // 삽입할 노드를 만듦
     newNode.key ← x;
     newNode.right ← null;
     newNode.left ← null;
     if (B = null) then B ← newNode;    // 공백 이원 탐색 트리인 경우
     else if (x < q.key) then           // q는 탐색이 실패로 종료하게 된 원소
           q.left ← newNode;
     else
         q.right ← newNode;
     return;
end insertBST()
```

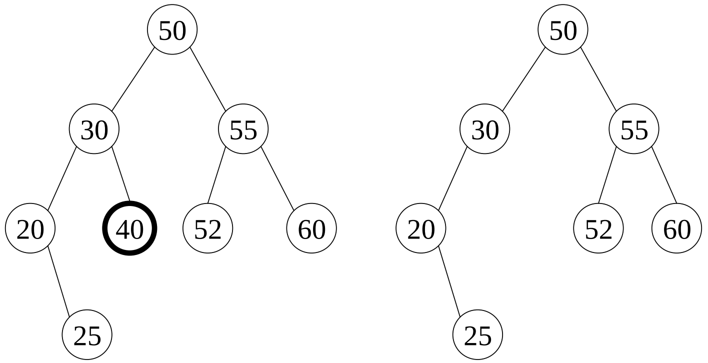
이원 탐색 트리에서의 원소 삭제 (1)

- 삭제하려는 원소의 키 값이 주어졌을 때
  - 이 키 값을 가진 원소를 탐색
  - 원소를 찾으면 삭제 연산 수행

- 해당 노드의 자식 수에 따른 3가지 삭제 연산
  1. 자식이 없는 리프 노드의 삭제
  2. 자식이 하나인 노드의 삭제
  3. 자식이 둘인 노드의 삭제

이원 탐색 트리에서의 원소 삭제 (2)

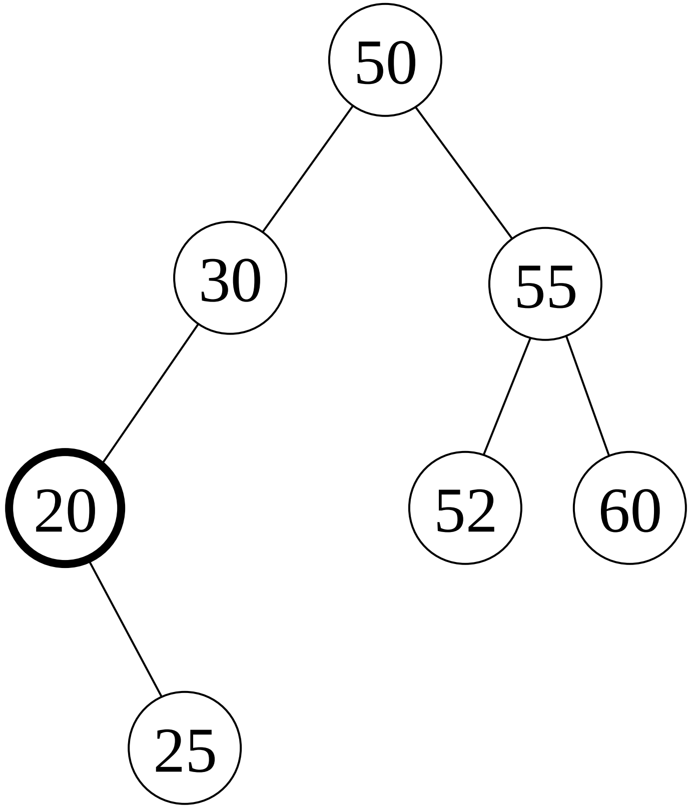
1. 자식이 없는 리프 노드의 삭제
   - 부모 노드의 해당 링크 필드를 널(null)로 만들고 삭제한 노드 반환
   - 예 : 키 값 40을 가진 노드의 삭제시



(a) 삭제전 (b) 삭제후

이원 탐색 트리에서의 원소 삭제 (3)

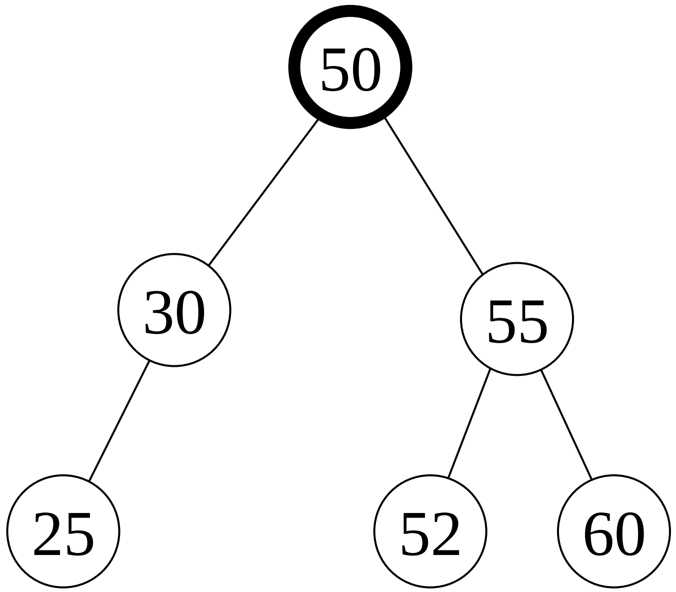
2. 자식이 하나인 노드의 삭제
   - 삭제되는 노드 자리에 그 자식 노드를 위치
   - 예 : 원소 20을 삭제

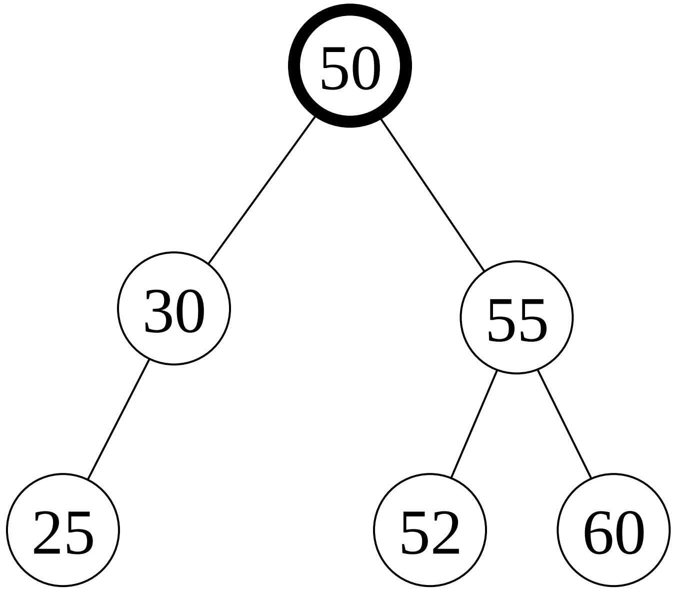


이원 탐색 트리에서의 원소 삭제 (4)

3. 자식이 둘인 노드의 삭제
   - 삭제되는 노드 자리에 왼쪽 서브트리에서 제일 큰 원소나 또는 오른쪽 서브트리에서 제일 작은 원소로 대체
   - 해당 서브트리에서 대체 원소를 삭제
   - 대체하게 되는 노드의 차수는 1 이하가 됨

예 : 키 값이 50인 루트 노드의 삭제시



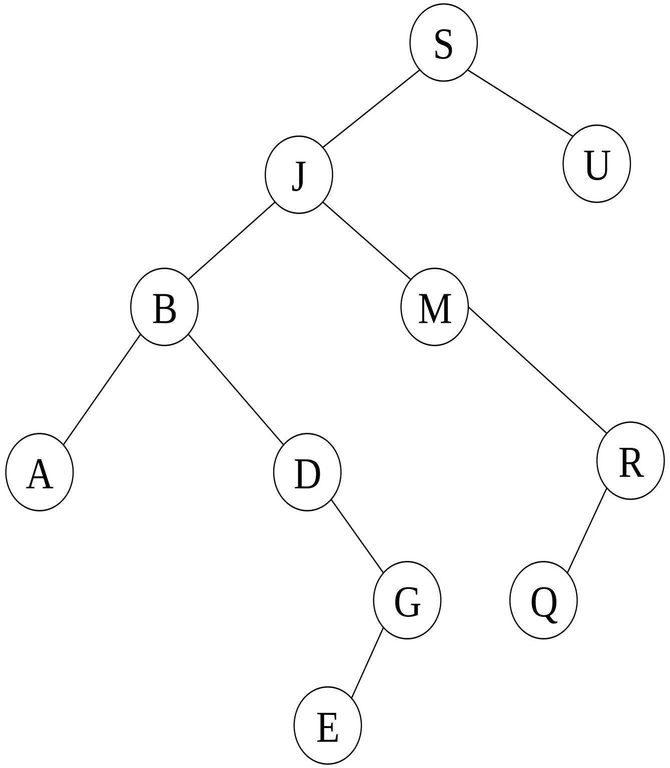


삭제 알고리즘

```
deleteBST(B, x)
    p ← the node to be deleted;                 //주어진 키 값 x를 가진 노드
    parent ← the parent node of p;              // 삭제할 노드의 부모 노드
    if (p = null) then return false;            // 삭제할 원소가 없음

    case {
        p.left = null and p.right = null :     // 삭제할 노드가 리프 노드인 경우
            if (parent.left = p) then parent.left ← null;
            else parent.right ← null;
        p.left = null or p.right = null :      // 삭제할 노드의 차수가 1인 경우
            if (p.left ≠ null) then {
                if (parent.left = p) then parent.left ← p.left;
                else parent.right ← p.left;
            } else {
                if (parent.left = p) then parent.left ← p.right;
                else parent.right ← p.right;
            }
        p.left ≠ null and p.right ≠ null :  // 삭제할 노드의 차수가 2인 경우
            q ← maxNode(p.left);  // 왼쪽 서브트리에서 최대 키 값을 가진 원소를 탐색
            p.key ← q.key;
            deleteBST(p.left, p.key);
    }
end deleteBST()
```

이원 탐색 트리의 구축과 Java 구현

- 스트링 타입의 키 값을 가진 이원 탐색 트리 구축
  - 스트링 "R"을 가지고 있는 노드 탐색
  - 트리에 없는 스트링 "C"를 탐색



```java
class TreeNode {
    String  key;
    TreeNode  left;
    TreeNode  right;
}

class BinarySearchTree {
    private TreeNode  rootNode;

    /** insert() 메소드에 의해 사용되는 보조 순환 메소드 */
    private TreeNode insertKey(TreeNode T, String x) {
        if (T == null) {
            TreeNode newNode = new TreeNode();
            newNode.key = x;
            return newNode;
        } else if (x.compareTo(T.key) < 0) {  // x < T.key이면 x를 T의 왼쪽
            T.left = insertKey(T.left, x);    // 서브트리에 삽입
            return T;
        } else if (x.compareTo(T.key) > 0) {  // x > T.key이면 x를 T의 오른쪽
            T.right = insertKey(T.right, x);  // 서브트리에 삽입
            return T;
        } else {                              // key값 x가 이미 T에 있는 경우
            return T;
        }
    }
```

```java
void insert(String x) {
    rootNode = insertKey(rootNode, x);
}

/** 키 값 x를 가지고 있는 TreeNode의 포인터를 반환 */
TreeNode find(String x) {
    TreeNode T = rootNode;
    int result;
    while (T != null) {
        if ((result = x.compareTo(T.key)) < 0) {
            T = T.left;
        } else if (result == 0) {
            return T;
        } else {
            T = T.right;
        }
    }
    return T;
}
```

```java
    /** print() 메소드에 의해 사용되는 순환 메소드 */
    private void printNode(TreeNode N) {
        if (N != null) {
            System.out.print("(");
            printNode(N.left);
            System.out.print(N.key);
            printNode(N.right);
            System.out.print(")");
        }
    }

    /** 서브트리 구조를 표현하는 괄호 형태로 트리를 프린트 */
    void printBST() {
        printNode(rootNode);
        System.out.println();
    }
}
```

```java
class BinarySearchTreeTest {

    public static void main(String args[]) {
        BinarySearchTree T = new BinarySearchTree();

        // 그림 8.6의 BST를 구축
        T.insert("S");
        T.insert("J");
        T.insert("B");
        T.insert("D");
        T.insert("U");
        T.insert("M");
        T.insert("R");
        T.insert("Q");
        T.insert("A");
        T.insert("G");
        T.insert("E");

        // 구축된 BST를 프린트
        System.out.println(" The Tree is:");
        T.printBST();
        System.out.println();
```

```java
        // 스트링 "R"을 탐색하고 프린트
        System.out.println(" Search For \"R\"");
        TreeNode N = T.find("R");
        System.out.println("Key of node found = " + N.key);
        System.out.println();

        // 스트링 "C"를 탐색하고 프린트
        System.out.println(" Search For \"C\"");
        TreeNode P = T.find("C");
        if (P != null) {
            System.out.println("Key of node found = " + P.key);
        } else {
            System.out.println("Node that was found = null");
        }
        System.out.println();
    }
}
```

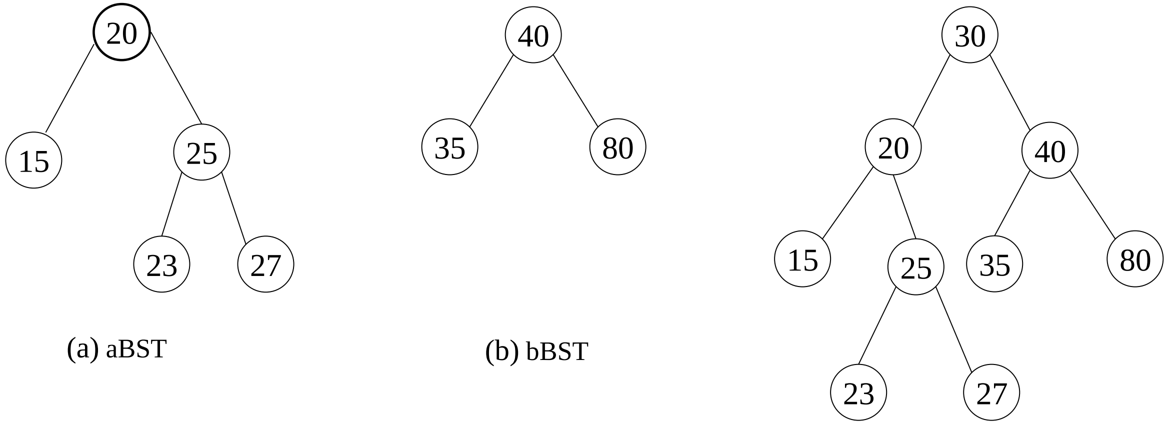
이원 탐색 트리의 결합 (1)

- 이원 탐색 트리의 결합과 분할 연산
  - 주어진 조건에 따라 이원 탐색 트리를 결합하거나 분할
- 3원 결합 : threeJoin (aBST, x, bBST, cBST)
  - 이원 탐색 트리 aBST와 bBST에 있는 모든 원소들과
    키 값 x를 갖는 원소를 루트 노드로 하는 이원 탐색 트리
    cBST를 생성
  - 가정
    - aBST의 모든 원소 < x < bBST의 모든 원소
    - 결합이후에 aBST와 bBST는 사용하지 않음

- 3원 결합의 연산 내용
    - 새로운 트리 노드 cBST를 생성하여 key값으로 x를 지정
    - left 링크 필드에는 aBST를 설정
    - right 링크 필드에는 bBST를 설정

이원 탐색 트리의 결합 (2)

- 3원 결합의 예(x=30 이라고 가정)



(a) aBST (b) bBST (c) cBST

- 이원 탐색 트리의 높이
  : max{height(aBST) , height(bBST)} + 1

이원 탐색 트리의 결합 (3)

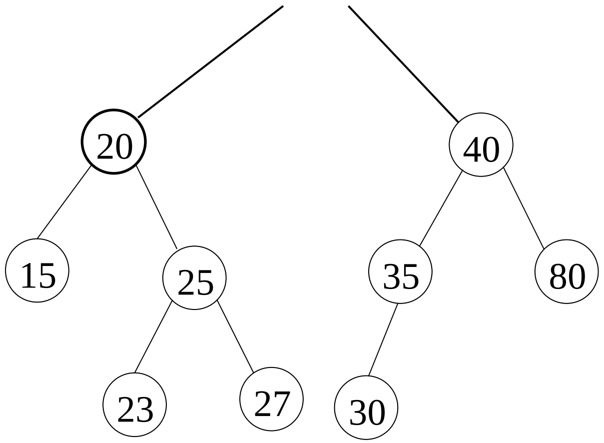
- 2원 결합 알고리즘: twoJoin(aBST, bBST, cBST)
  - 두 이원 탐색 트리 aBST와 bBST를 결합하여 aBST와 bBST에 있는 모든 원소들을 포함하는 하나의 이원 탐색 트리 cBST를 생성
  - 가정
    - aBST의 모든 키 값 < bBST의 모든 키 값
    - 연산 후 aBST와 bBST는 사용 하지 않음

- 2원 결합 연산 실행
  1. aBST나 bBST가 공백인 경우
     - cBST는 공백이 아닌 aBST 또는 bBST가 됨
  2. aBST와 bBST가 모두 공백이 아닌 경우
     - 2 가지 방법으로 실행 가능

이원 탐색 트리의 결합 (4)

(1) aBST에서 가장 큰 키 값을 루트로 하여 결합

- aBST에서 키 값이 가장 큰 원소를 삭제한 결과 트리 : aBST'
- 삭제한 가장 큰 키 값: max
- threeJoin(aBST', max, bBST, cBST)를 실행

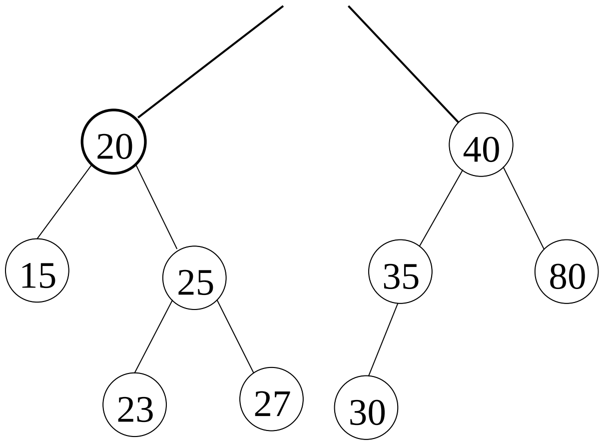


- cBST의 높이 : max{height(aBST), height(bBST)} + 1

이원 탐색 트리의 결합 (5)

(2) bBST에서 가장 작은 키 값을 루트로 하여 결합

- bBST에서 키 값이 가장 작은 가진 원소를 삭제한 결과 트리 : bBST′
- 삭제한 가장 작은 키 값: min
- threeJoin(aBST, min, bBST', cBST)를 실행



이원 탐색 트리의 분할 (1)

- 주어진 키 값 x를 기준으로 두 개의 이원 탐색 트리로 분할

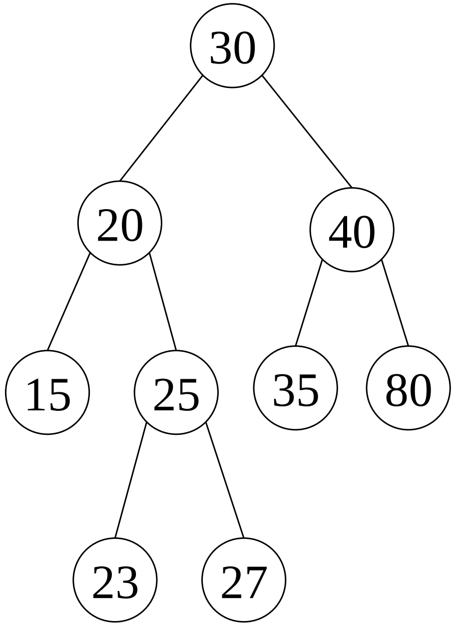
- 분할 알고리즘: split(aBST, x, bBST, cBST)
  - 주어진 이원 탐색 트리 aBST를 키 값 x를 기준으로 이원 탐색 트리 bBST와 cBST로 분할
  - bBST는 aBST의 원소 중에서 x보다 작은 키 값을 가진 모든 원소를 포함
  - cBST는 aBST의 원소 중에서 x보다 큰 키 값을 가진 모든 원소를 포함
  - bBST와 cBST는 모두 이원 탐색 트리 성질을 만족
  - 키 값 x가 aBST에 있으면 true 반환 , 아니면 false 반환

이원 탐색 트리의 분할 (2)

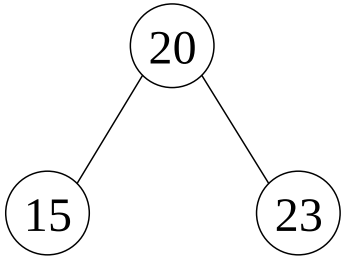
- 분할 연산 실행
  1. 키 값 x를 가진 원소를 탐색하면서 aBST를 아래로 이동
  2. X = (aBST의 루트 노드의 키 값) 일 때
     - 왼쪽 서브트리는 bBST가 되고
     - 오른쪽 서브트리는 cBST가 된다.
     - 그리고 true를 반환
  3. X < (루트 노드의 키 값) 일 때
     - 루트와 그 오른쪽 서브트리는 cBST에 속한다.
     - 그러나 현재의 cBST에 있는 키 값보다는 모두 작다.
     - 따라서 현재의 cBST의 가장 작은 키 값의 왼쪽 서브트리가 되어야 한다.
  4. X > (루트 노드의 키 값) 일 때
     - 루트와 그 왼쪽 서브트리는 bBST에 속한다.
     - 그러나 현재의 bBST에 있는 키 값보다는 모두 크다.
     - 따라서 현재의 bBST의 가장 큰 키 값의 오른쪽 서브트리가 되어야 한다.

이원 탐색 트리의 분할 (3)

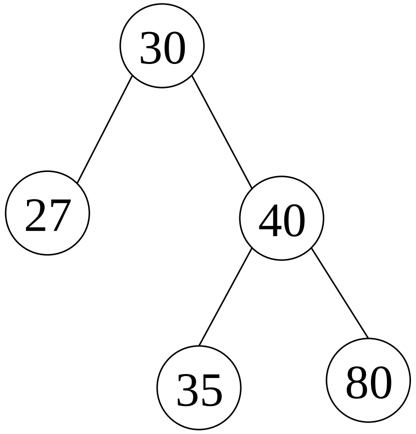
- split(aBST , 25 , bBST , cBST)



(a) aBST



(b) bBST



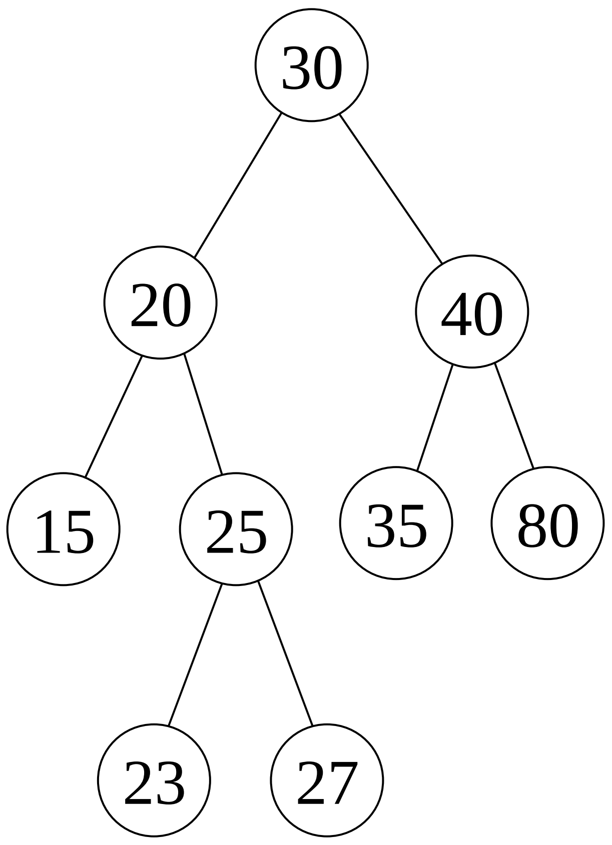
(c) cBST

이원 탐색 트리의 분할 (4)

- 분할 연산 알고리즘
  - 2 개의 dummy 루트 노드를 사용
    - Small: x보다 작은 키 값으로 된 서브트리를 오른쪽 서브트리로 지시
    - Large: x보다 큰 키 값으로 된 서브트리를 왼쪽 서브트리로 지시

분할 연산 알고리즘: splitBST(aBST, x, bBST, cBST)

```
splitBST(aBST, x, bBST, cBST)     // x는 aBST를 분할하는 키 값
    Small ← getTreeNode();
    Large ← getTreeNode();
    S ← Small;    // Small BST의 순회 포인터
    L ← Large;    // Large BST의 순회 포인터
    P ← aBST;     // aBST의 순회 포인터

    while (P ≠ null) do {
        if (x = P.key) then {
            S.right ← P.left;
            L.left ← P.right;
            bBST ← Small.right;
            cBST ← Large.left;
            return true;    // 키 값 x가 aBST에 있음
        } else if (x < P.key) then {
            L.left ← P;
            L ← P;
            P ← P.left;
        } else {
            S.right ← P;
            S ← P;
            P ← P.right;
        }
    }
    bBST ← Small.right;
    cBST ← Large.left;
    return false;   // 키 값 x는 aBST에 없음
end splitBST()
```



이원 탐색 트리의 높이

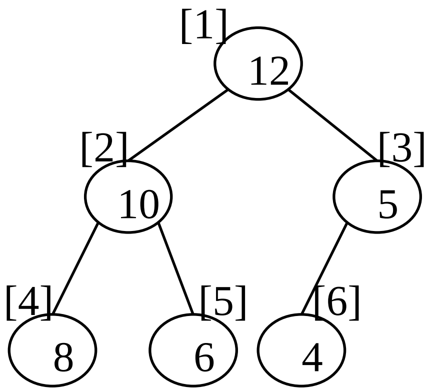
- 이원 탐색 트리의 높이
  - 이원 탐색 트리의 높이가 커지면
    원소의 검색,삽입,삭제 연산 수행 시간이 길어진다.
  - n개의 노드를 가진 이원 탐색 트리의 최대 높이는 n-1
  - 최소 이원 탐색 트리의 높이는 $O(\log n)$

- 균형 탐색 트리 (balanced search tree)
  - 탐색 트리의 높이가 최악의 경우에도 $O(\log n)$이 되는
    탐색 트리
  - 검색, 삽입, 삭제 연산 시간은 $O(height)$
    - 예) AVL , 2-3, 2-3-4 , red-black, B-tree

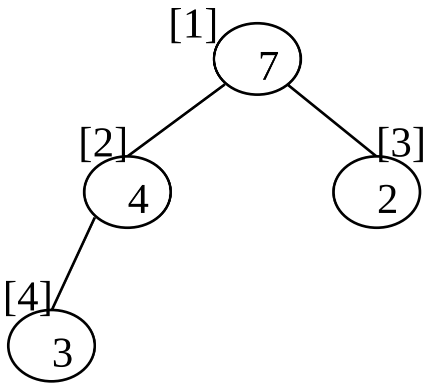
8.2 Heap

히프(heap) (1)

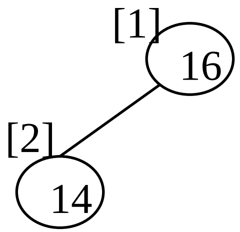
- 히프(heap) 또는 최대 히프(max heap)
  - 각 노드의 키 값이 그 자식의 키 값보다 **작지 않은** 완전 이진 트리(complete binary tree)
- 최소 히프(min heap)
  - 각 노드의 키 값이 그 자식의 키 값보다 **크지 않은** 완전 이진 트리 (complete binary tree)
- 최대 히프 예



(a)



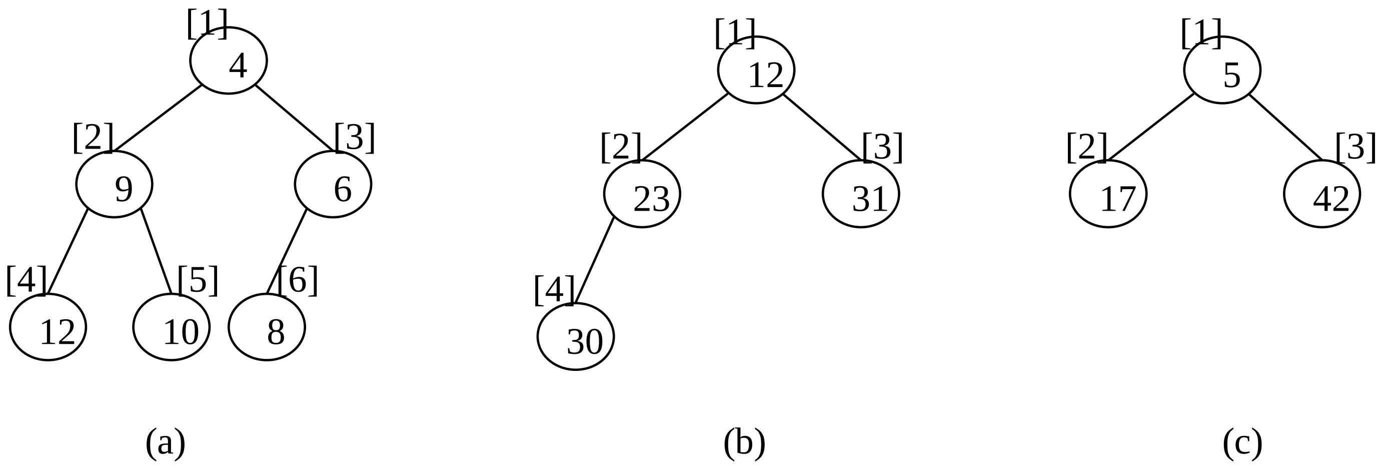
(b)



(c)

히프 (2)

- 최소 히프 예



  - 최소 히프의 루트는 그 트리에서 가장 작은 키 값을 가짐
  - 최대 히프의 루트는 그 트리에서 가장 큰 키 값을 가짐

히프 추상 데이타 타입 (1)

- 히프 추상 데이타 타입에 포함될 기본 연산

  ① 생성(create) : 공백 히프의 생성

  ② 삽입(insert) : 새로운 원소를 히프의 적절한 위치에 삽입

  ③ 삭제(delete) : 히프에서 키 값이 가장 큰 원소를 삭제하고
  원소를 반환

히프 추상 데이타 타입 (2)

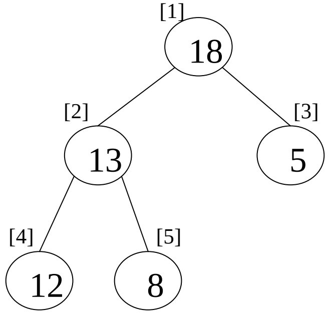
```
ADT Heap

데이타 : n>0 원소로 구성된 완전 이진 트리로 각 노드의 키 값은 그의
       자식 노드의 키 값보다 작지 않다(max heap).

연산 :
    heap ∈ Heap; item ∈ Element;
    createHeap() := create an empty heap;
    insertHeap(heap, item) := insert item into heap
    isEmpty(heap) := if (heap is empty then return true
                     else return false
    deleteHeap(heap) := if (isEmpty(heap)) then return error
                        else {
                            item ← the largest element in heap;
                            remove the largest element in heap;
                            return item;
                        }

End Heap
```

히프에서의 원소 삽입 (1)

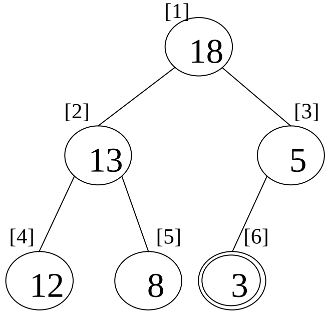
- 원소 삽입 과정
  1. 히프 끝에 새로운 원소를 첨가하여 구조를 확장
  2. 변경된 원소의 부모 원소가 히프 성질을 만족하면 종료
  3. 아니면 부모 원소가 히프 성질을 만족할 수 있도록 자식 원소와 교환
  4. 단계 2로 돌아가 다시 수행

- 히프 재 조정 작업은 삽입한 노드의 부모 노드부터 루트 노드로 올라 가면서 수행

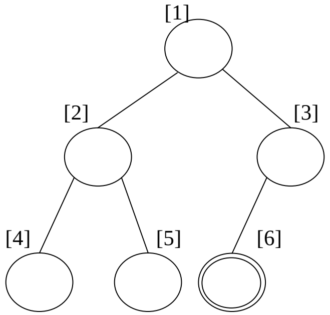
히프에서의 원소 삽입 (2)



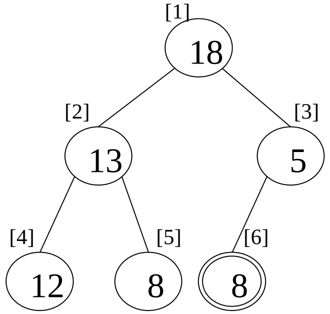
(a)히프



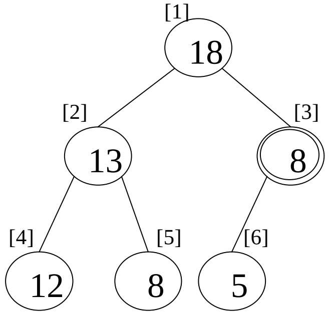
(c) 원소 3을 삽입



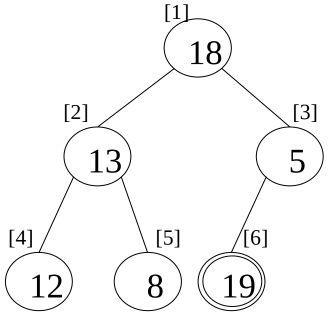
(b) 삽입후의 히프구조



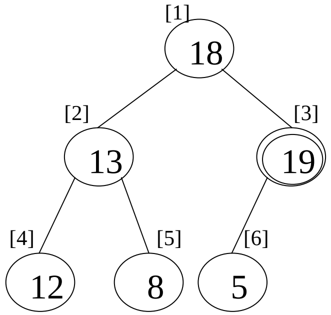
(d)히프 (a)에 원소 8 삽입



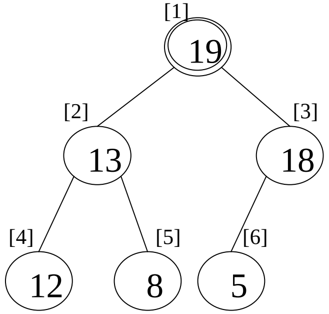
(e) 원소 8을 삽입 후



(f) 히프 (a)에 원소 19삽입



(g) 원소의 이동



(h) 원소 19 삽입 후

히프에서의 원소 삽입 (3)

- 부모 노드를 식별하는 방법이 필요
  - 연결 표현 사용 시: 각 노드에 parent 필드 추가
  - 순차 표현 사용 시: 위치 i의 부모 노드는 인덱스는 $\lfloor i/2 \rfloor$
- 히프에 대한 삽입 알고리즘

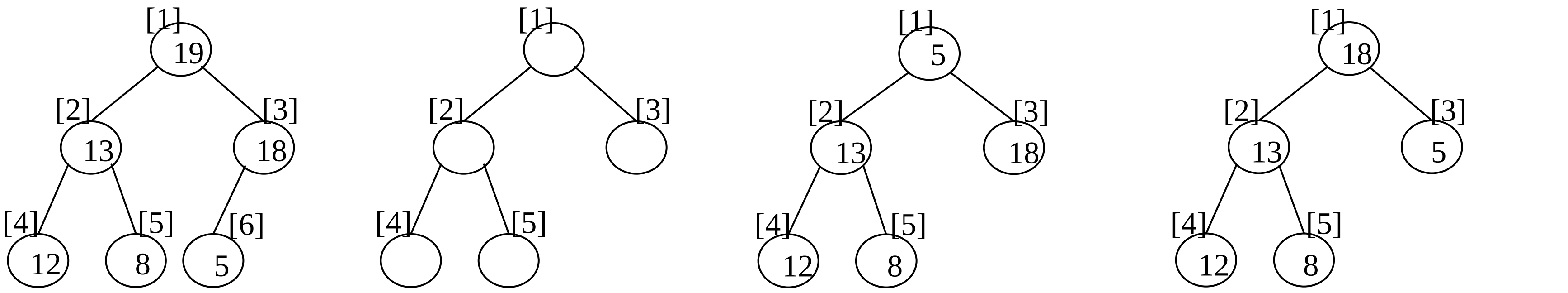
```
// 순차 표현으로 구현된 최대 히프
// 원소 item를 히프 heap에 삽입, n은 현재 히프의 크기(원소 수)
insertHeap(Heap, item)
    if (n = maxSize) then heapFull;    // 히프가 만원이면 히프 크기를 확장
    n ← n + 1;                         // 새로 첨가될 노드 위치
    for (i ← n; ; ) do {
        if (i = 1) then exit;          // 루트에 삽입
        if(item ≤ heap[ ⌊i/2⌋ ]) then exit;   // 삽입할 노드의 키 값과
                                              // 부모 노드 키 값을 비교
        heap[i] ← heap[ ⌊i/2⌋ ];       // 부모 노드 키 값을 자식노드로 이동
        i ←  ⌊i/2⌋ ;
    }
    heap[i] ← item;
end insertHeap()
```

히프에서의 원소 삭제 (1)

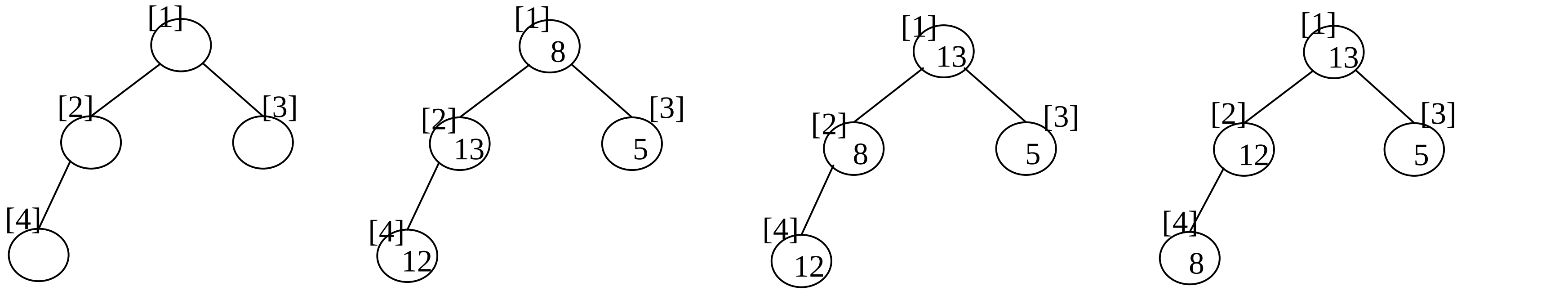
- 원소 삭제 과정
  1. 히프 끝 원소를 루트 원소와 교환하면서 루트 원소와 히프 끝 노드를 삭제
  2. 교환한 원소가 두 자식 원소에 대해 히프 성질을 만족하면 종료
  3. 아니면 히프 성질을 만족할 수 있도록 적절한 자식과 교환
  4. 단계 2로 돌아가 다시 수행

- 히프 재 조정 작업은 루트부터 자식 노드로 내려 가면서 수행

히프에서의 원소 삭제 (2)



(a)히프 (b)삭제 후의 히프 구조 (c)삭제 중간 단계 (d)첫번째 삭제 뒤의 히프



(e)두번째 삭제 뒤의 히프 구조 (f)삭제 중간 단계 (g)삭제 중간 단계 (h)두번째 삭제가 완료된 히프

히프에서의 원소 삭제 (3)

```
// 히프로부터 원소 삭제, n은 현재의 히프 크기(원소 수)
deleteHeap(heap)
    item ← heap[1];     // 삭제할 원소
    temp ← heap[n];     // 이동시킬 원소
    n ← n - 1;          // 히프 크기(원소 수)를 하나 감소
    i ← 1;
    j ← 2;              // j는 i의 왼쪽 자식 노드
    while (j ≤ n) do {
        if (j < n) then
            if (heap[j] < heap[j + 1])
                then j ← j + 1;          // j는 값이 큰 자식을 가리킨다.
        if (temp ≥ heap[j]) then exit;
        heap[i] ← heap[j];               // 자식을 한 레벨 위로 이동
        i ← j;
        j ← j * 2;                       // i와 j를 한 레벨 아래로 이동
     }
     heap[i] ← temp;
     return item;
end deleteHeap()
```

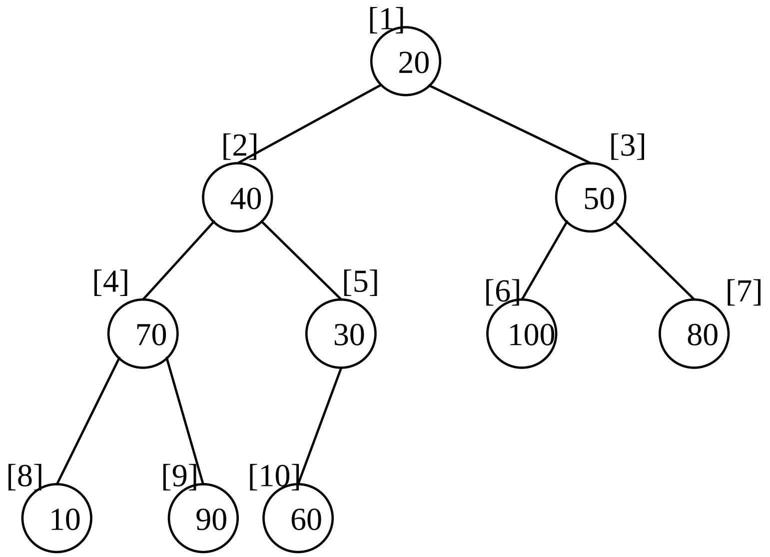
완전 이진 트리를 히프로 변환 (1)

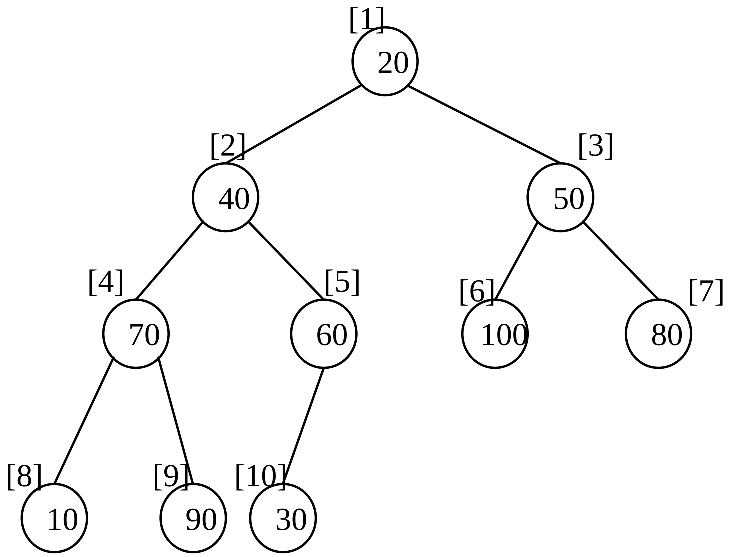
- H의 내부 노드 각각을 루트로 하는 서브트리를 역 레벨 순서에 따라 차례로 히프로 만들어 나가면 됨
  - 내부 노드는 자식을 가지고 있는 노드
  - 역 레벨 순서는 포화 이진 트리 번호의 역순 즉, 노드 번호의 오름차 순

- 각 서브트리는 그 자체로 히프가 될 때까지 반복적으로 조정 작업을 수행

완전 이진 트리를 히프로 변환 (2)

- 변환 예
  - 히프가 아닌 완전 이진 트리

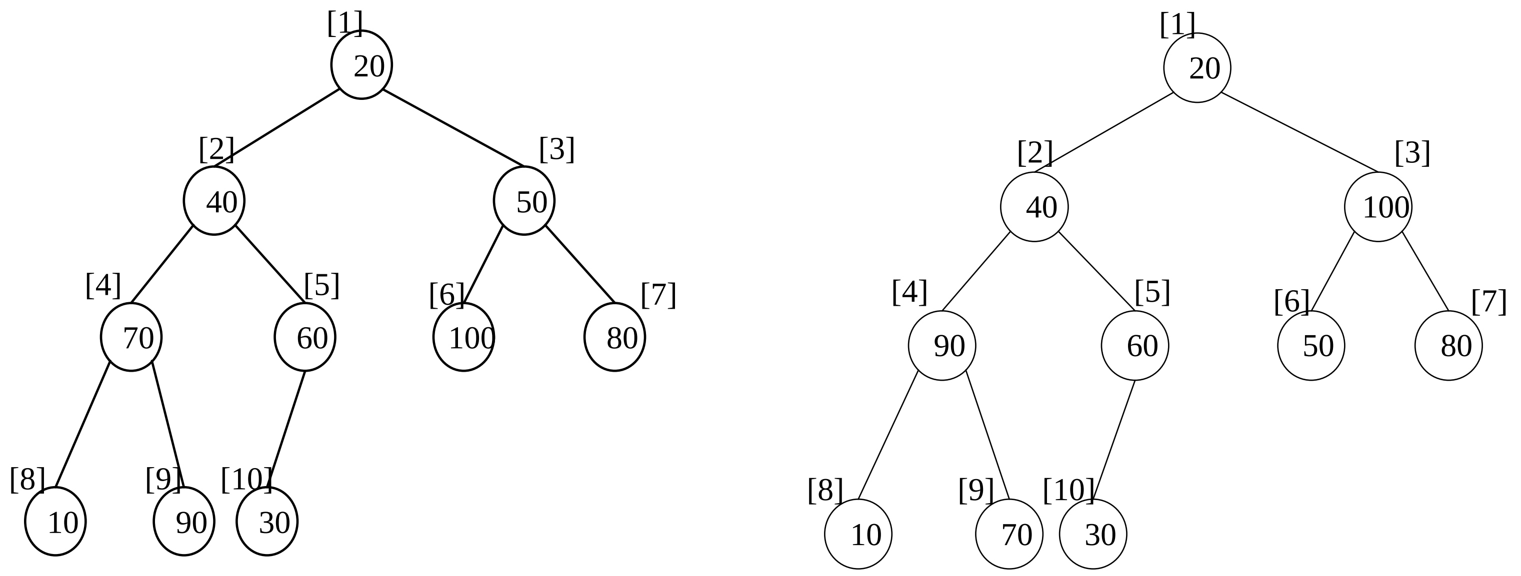




- 내부 노드의 역 레벨 순서: [5], [4], [3], [2], [1]
- [5]번 노드를 루트로 하는 서브트리에서 히프 연산 시작
  - 자식 중에 큰 키 값(60)을 가진 노드 [10]과 원소 교환(30↔60)

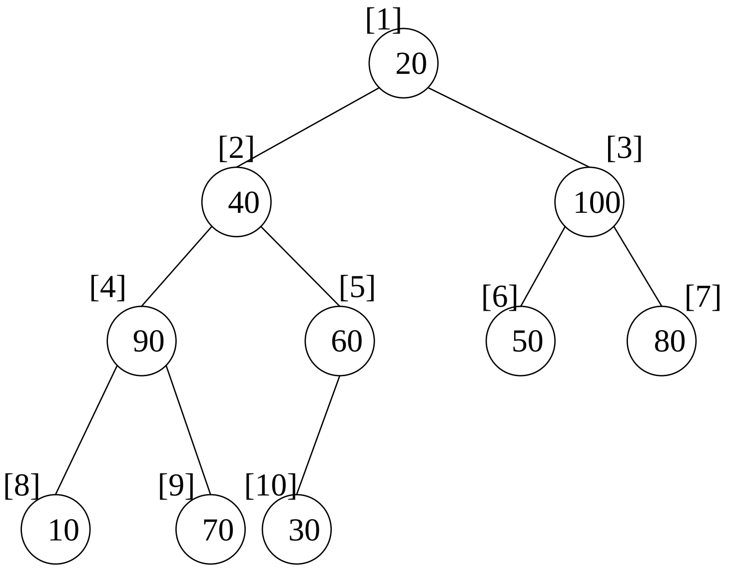
완전 이진 트리를 히프로 변환 (3)

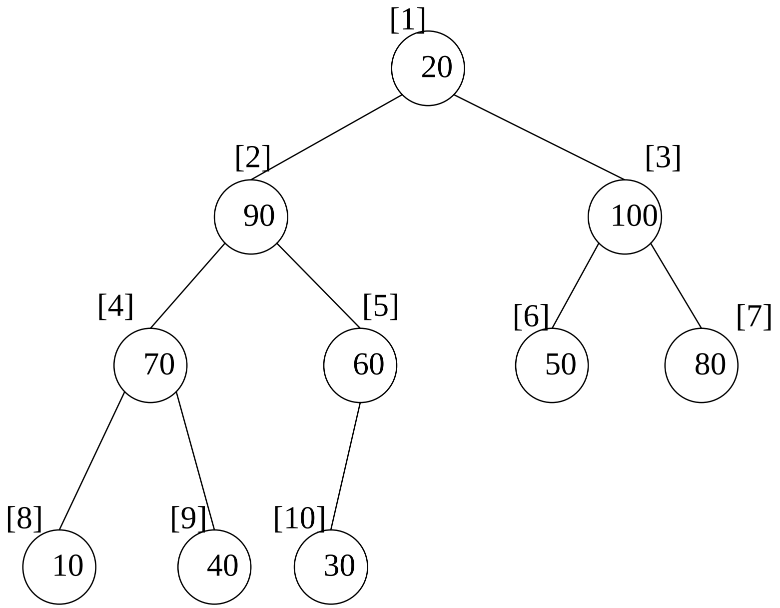
- 다음 [4]번 노드를 루트로 하는 서브트리 조사
  - 자식 중에 큰 키 값(90)을 가진 노드 [9]와 원소 교환(70 ↔ 90)
- 다음 [3]번 노드를 루트로 하는 서브트리를 조사
  - 자식 중에 큰 키 값(100)을 가진 노드 [6]과 원소 교환(50 ↔ 100)



완전 이진 트리를 히프로 변환 (4)

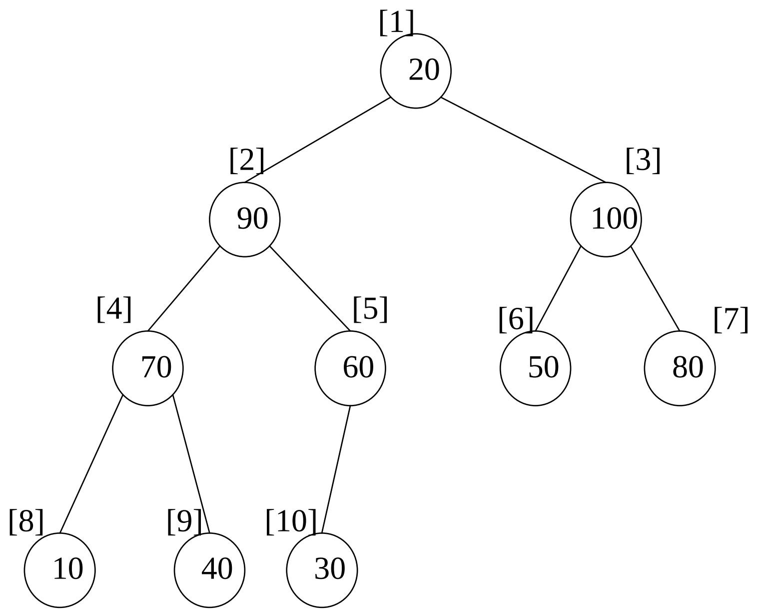
- 다음 [2]번 노드를 루트로 하는 서브트리 조사
  - 자식 중에 큰 키 값(90)을 가진 노드 [4]와 원소 교환 (40 ↔ 90)
  - 다시 계속해서 [9]번 노드의 키 값(70)과 교환(40 ↔ 70)

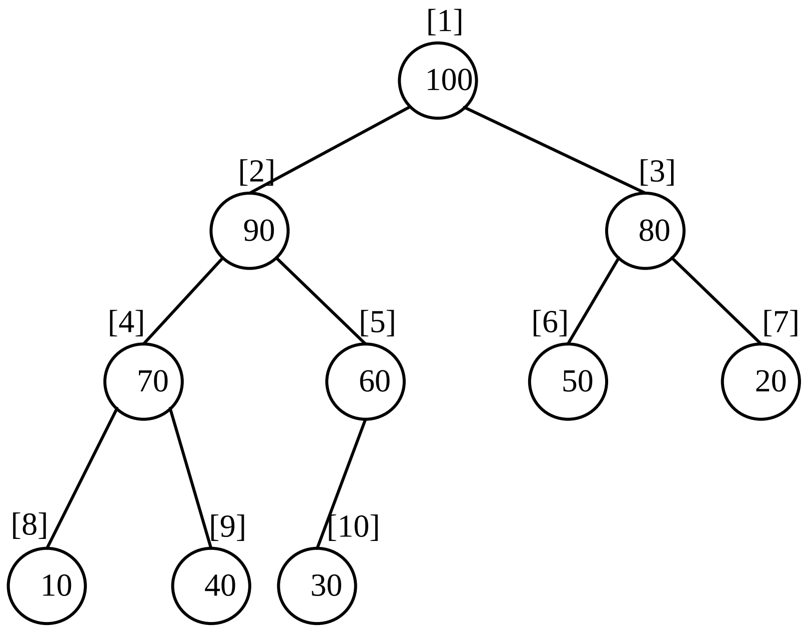




완전 이진 트리를 히프로 변환 (5)

- 끝으로 루트 노드 [1]번 노드를 고려
  - 자식 중에 큰 키 값(100)을 가진 노드 [3]과 원소 교환(20 ↔ 100)
  - 다시 계속해서 [7]번 노드의 키 값(80)과 교환(20 ↔ 80)





완전 이진 트리를 히프로 변환하는 알고리즘

```
// H는 히프가 아닌 완전 이진 트리
makeTreeHeap(H, n)
    for (i ← n / 2; i ≥ 1; i ← i - 1) do {
        // 각 내부 노드에 대해 레벨 순서의 역으로
        p ← i;
        for (j ← 2 * p; j ≤ n; j ← 2 * j) do {
            if (j < n) then
                if (H[j] < H[j + 1]) then j ← j + 1;
            if (H[p] ≥ H[j]) then exit;
            temp ← H[p];
            H[p] ← H[j];
            H[j] ← temp
            p ← j;   // 부모 노드를 한 레벨 밑으로 이동
        }
    }
end makeTreeHeap()
```

히프를 이용한 우선순위 큐 표현 (1)

- 히프는 우선 순위 큐 표현에 효과적

- 우선순위가 제일 높은 원소를 찾거나 삭제하는 것은 아주 쉬움
  - 노드 삭제 시: 나머지 노드들을 다시 히프가 되도록 재조정
  - 노드 삽입 시: 삽입할 원소의 우선순위에 따라 히프가
    유지되도록 해야 됨

- 히프 정렬
  - 정렬할 원소를 모두 히프로 이동
  - 히프로부터 원소를 하나씩 삭제하여 저장
  - 결과는 내림차순 정렬

히프를 이용한 우선순위 큐 표현 (2)

- class PriorityQueue
  - 배열을 이용해 구현한 히프로 표현한 우선순위 큐

  - 정렬된 우선순위 큐나 무정렬 배열을 이용한 큐 (6.7.2절) 구현에서 사용된 PriorityQueue class 대체 가능

  - 우선순위 큐 정렬 메소드를 정의한 프로그램에서

    PriorityQueue class를 사용하면 히프 정렬 (heapsort)

    버젼이 됨
    - 우선순위 큐 정렬: $O(n^2)$
    - 히프 정렬: O(n *log* n)

히프로 표현한 우선순위 큐 클래스

```
class PriorityQueue{
    private int count;                        // 우선순위 큐의 현재 원소 수
    private int size;                         // 배열의 크기
    private int increment;                    // 배열 확장 단위
    private PrioityElement[] itemArray;       // 우선순위 큐 원소를 저장하는 배열

    public PriorityQueue(){
        count     = 0;                        // itemArray[0]는 실제로 사용하지 않음
        size      = 16;                       //실제 최대 원소 수는 size - 1
        increment = 8;
        itemArray = new PrioityElement[size];
    }

    public int currentSize(){                 // 우선순위 큐의 현재 원소수
        return count;
    }
```

```java
public void insert(PrioityElement newKey) {
    // 우선순위 큐에 원소 삽입
    if (count == size - 1) PQFull();

    count++;   // 삽입 공간을 확보하고 원소의 삽입 위치를 밑에서부터 찾아 올라감

    int childLoc  = count;
    int parentLoc = childLoc / 2;

    while (parentLoc != 0) {
        if (newKey.compareTo(itemArray[parentLoc]) <= 0) {
            // 위치가 올바른 경우
            itemArray[childLoc] = newKey;    // 원소 삽입
            return;
        } else {   // 한 레벨 위의 위치로 이동
            itemArray[childLoc] = itemArray[parentLoc];
            childLoc  = parentLoc;
            parentLoc = childLoc / 2 ;
        }
    }

    itemArray[childLoc] = newKey;     // 최종 위치에 원소 삽입
}
```

```
public PriorityElement delete() {        // 우선순위 큐로부터 원소 삭제
    if (count == 0) {                    // 우선순위 큐가 공백인 경우
        return null;
    } else {
        int currentLoc;
        int childLoc;
        PriorityElement itemToMove;       // 이동시킬 원소
        PriorityElement deletedItem;      // 삭제한 원소

        deletedItem = itemArray[1];       // 삭제하여 반환할 원소
        itemToMove  = itemArray[count--]; // 이동시킬 원소
        currentLoc  = 1;
        childLoc    = 2 * currentLoc;

        while (childLoc <= count) {       // 이동시킬 원소의 탐색
            if (childLoc < count) {
                if (itemArray[childLoc + 1].compareTo(itemArray[childLoc]) > 0)
                    childLoc++;
            }
            if (itemArray[childLoc].compareTo(itemToMove) > 0) {
                itemArray[currentLoc]=itemArray[childLoc]; // 원소를 한 레벨
                currentLoc = childLoc;                     // 위로 이동
                childLoc   = 2 * currentLoc;
            } else {
                itemArray[currentLoc] = itemToMove;// 이동시킬 원소 저장
                return deletedItem;
            }
        }
        itemArray[currentLoc] = itemToMove;        // 최종 위치에 원소 저장
        return deletedItem;
    }
}
```

```java
// itemArray가 만원이면
    public void PQFull() {
        size += increment;     // increment만큼 더 크게 확장
        PriorityElement[] tempArray = new PriorityElement[size];
        for (int i = 1; i < count; i++) {
            tempArray[i] = itemArray[i];
        }
        itemArray= tempArray;
    }

}
```

8.3 선택 트리 (selection tree)

선택 트리(Selection tree) (1)

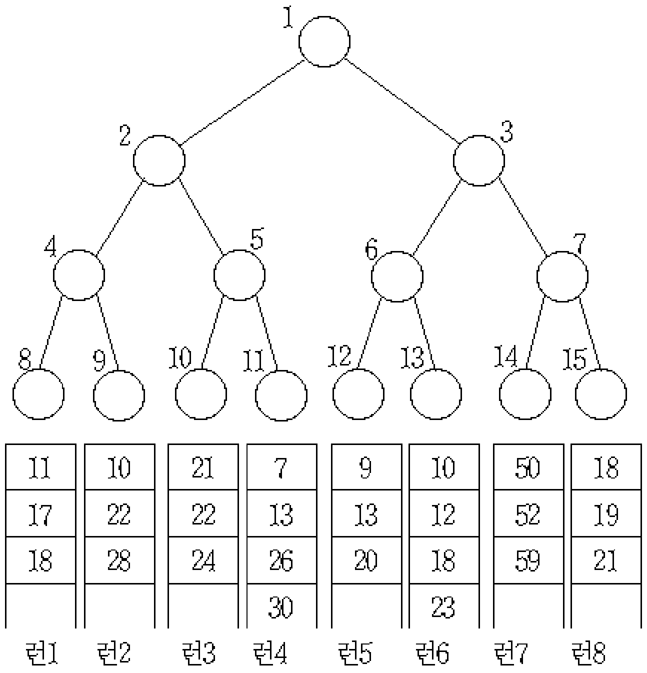
- k개의 런(run)에 나뉘어져 있는 n개의 원소들을 하나의 순서 순차로 합병할 때 사용할 수 있는 구조
  - 런(run) : 원소들이 키(key)값에 따라 오름차순으로 정렬되어 있는 순서 순차(ordered sequence)

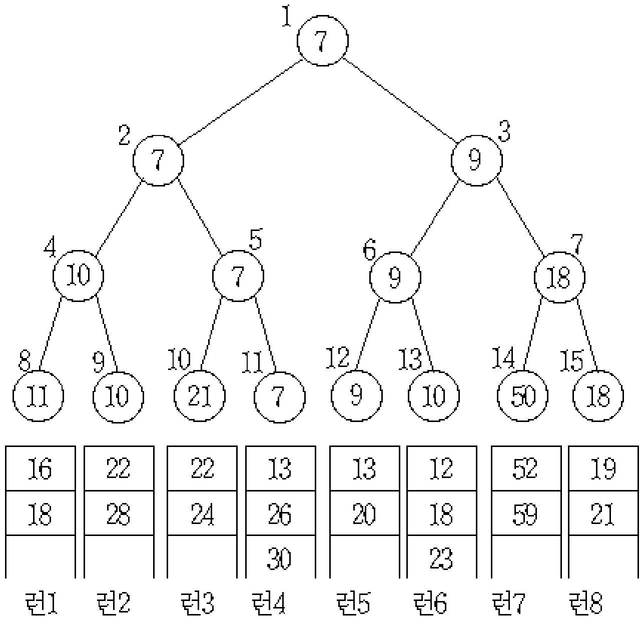
- K개의 런 중에서 가장 작은 키 값을 가진 원소를 계속적으로 선택해서 출력
  - k개의 원소 중에서 가장 작은 키 값을 가진 원소를 선택하는 경우에 보통 k - 1번 비교
  - 선택 트리를 이용하면 비교 회수를 줄임

- 선택 트리의 종류
  - 승자 트리 (winner tree)
  - 패자 트리 (loser tree)

선택 트리 (2)

- 승자 트리(winner tree)
  - 완전 이진 트리
  - 각 단말 노드는 각 런의 최소 키 값 원소를 나타냄
  - 내부 노드는 그의 두 자식 중에서 가장 작은 키 값을 가진 원소를 나타냄
- 런이 8개인 경우 승자 트리 예





선택 트리 (3)

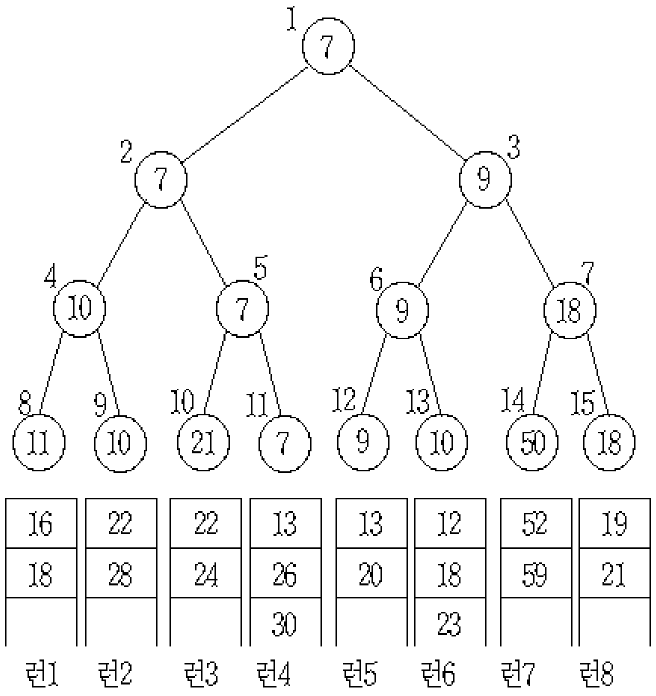
- 승자 트리 구축 과정
  - 가장 작은 키 값을 가진 원소가 승자로 올라가는 토너먼트 경기로 표현
  - 트리의 각 내부 노드: 두 자식 노드 원소의 토너먼트 승자
  - 루트 노드: 전체 토너먼트 승자, 즉 트리에서 가장 작은 키 값을 가진 원소

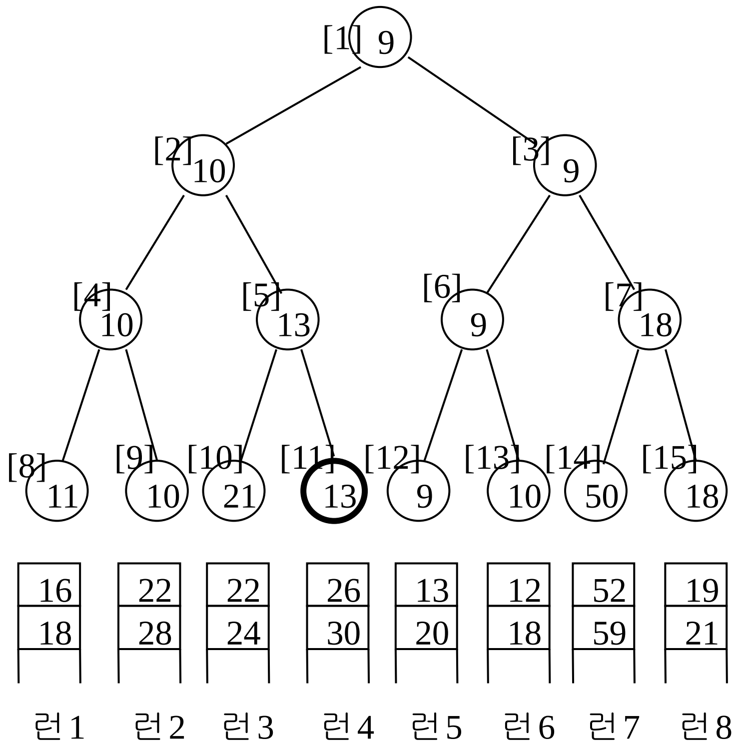
- 승자 트리의 표현
  - 완전 이진 트리이기 때문에 순차 표현이 유리
  - 인덱스 값이 i인 노드의 두 자식 인덱스는 2i와 2i+1

- 합병의 진행
  - 루트가 결정되는 대로 순서 순차에 출력 (여기선 7)
  - 다음 원소 즉 키 값이 13인 원소가 승자 트리로 들어감
  - 승자 트리를 다시 구성
    - 노드 11에서부터 루트까지의 경로를 따라가면서 형제 노드간 토너먼트를 진행

선택 트리 (4)

- 재 구성된 승자 트리의 예





- 이런 방식으로 순서 순차 구축을 계속함

선택 트리 (5)

- 패자 트리(loser tree)

  - 루트 위에 0번 노드가 추가된 완전 이진 트리

    (1) 단말 노드 : 각 런의 최소 키 값을 가진 원소

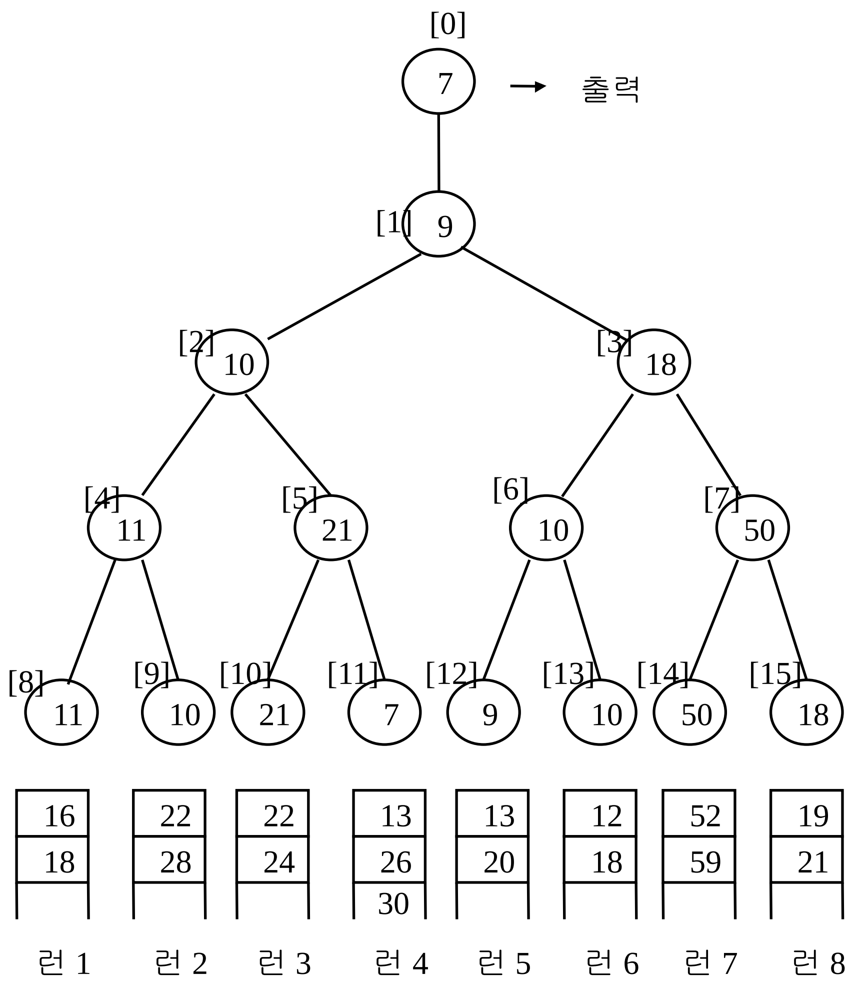
    (2) 내부 노드 : 토너먼트의 승자 대신 패자 원소

    (3) 루트(1번 노드) : 결승 토너먼트의 패자

    (4) 0번 노드 : 전체 승자(루트 위에 별도로 위치)

선택 트리 (6)

◦ 런이 8개인 패자 트리의 예

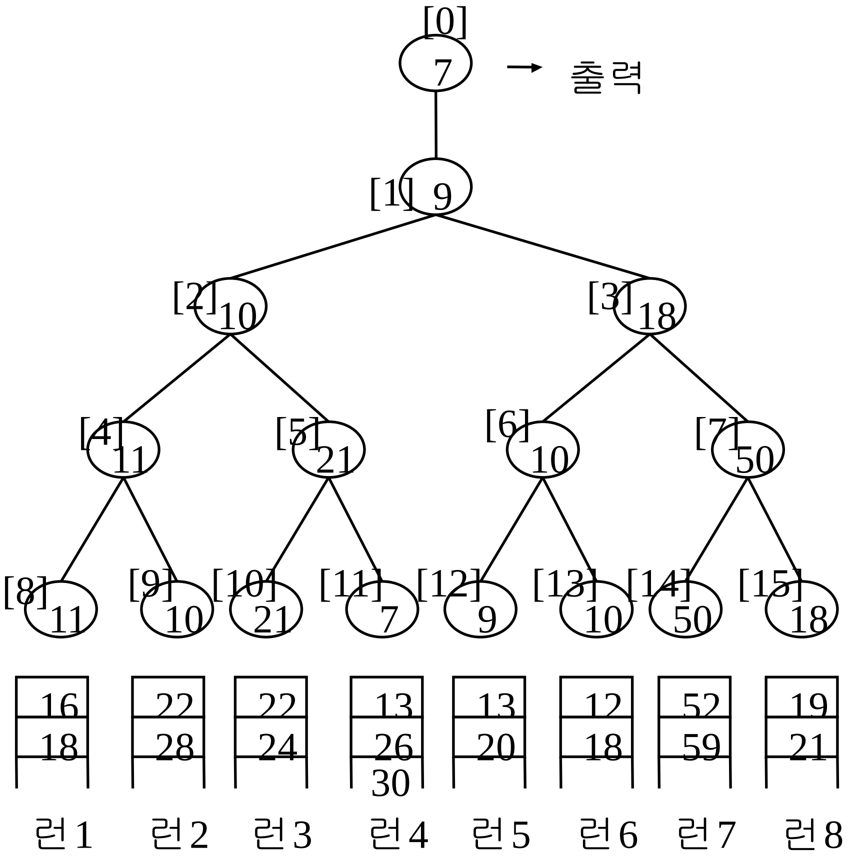


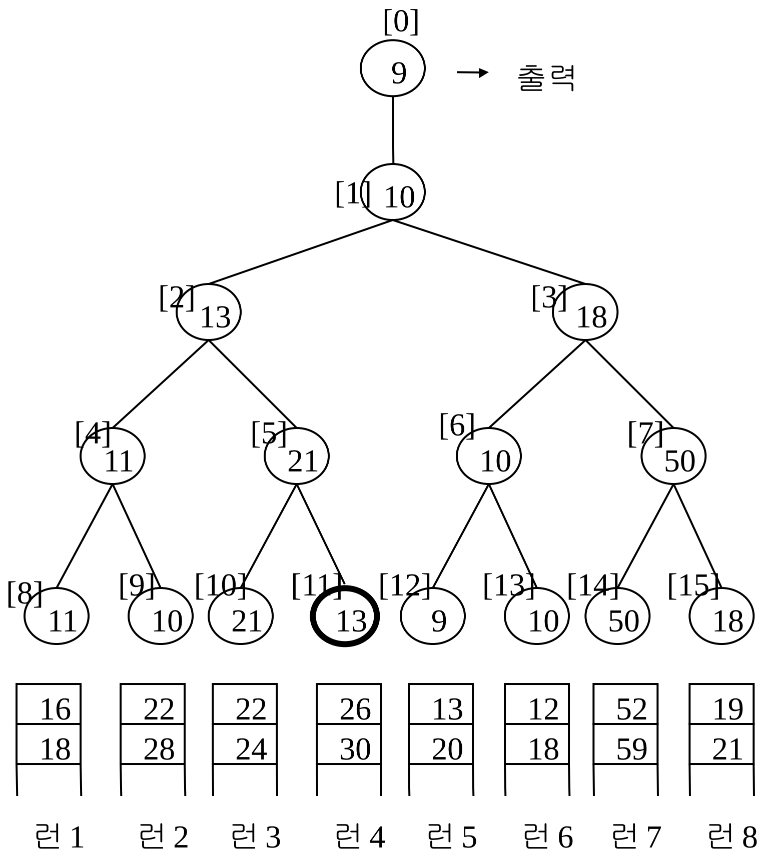
선택 트리 (7)

- 패자 트리 구축 과정
  - 단말 노드는 각 런의 최소 키 값 원소
  - 두 자식 노드들이 부모 노드에서 토너먼트 경기를 수행하여 패자는 부모 노드에 남고 승자는 다시 그 위 부모 노드로 올라가서 토너먼트 경기를 계속
  - 1번 루트 노드에서 패자는 루트 노드에 남고 승자는 전체 토너먼트의 승자로서 0번 노드로 올라가 순서 순차에 출력됨
- 합병의 진행
  - 출력된 원소가 속한 런 4의 다음 원소, 즉 키 값이 13인 원소를 패자 트리의 노드 11에 삽입
  - 패자 트리의 재 구성
    - 토너먼트는 노드 11에서부터 루트 노드 1까지의 경로를 따라 경기를 진행
    - 다만 경기는 형제 노드 대신 형식상 부모 노드와 경기를 함

선택 트리 (8)

◦ 재 구성된 패자 트리의 예





• 모든 원소가 순서 순차에 출력될 때까지 이 과정을 반복