8장 이원 탐색 트리

순서

8.1 이원 탐색 트리

8.2 히프

8.3 선택 트리

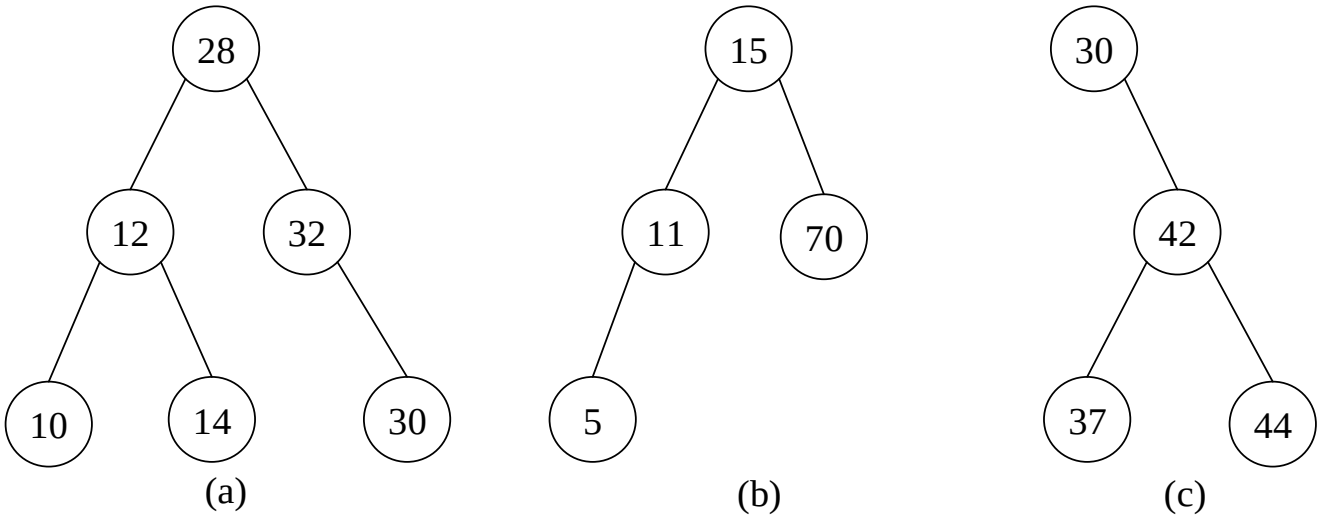
이원 탐색 트리 (binary search tree) (1)

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

- 정의 : 이원 탐색 트리(binary search tree:BST)
  - 이진 트리
  - 공백이 아니면 다음 성질을 만족
    - 모든 원소는 상이한 키를 갖는다.
    - 왼쪽 서브 트리 원소들의 키 < 루트의 키
    - 오른쪽 서브 트리 원소들의 키 > 루트의 키
    - 왼쪽 서브 트리와 오른쪽 서브 트리 : 이원 탐색 트리

이원 탐색 트리 (binary search tree) (2)

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



이원 탐색 트리에서의 탐색 (순환적 기술) (1)

- **키값이 x인 원소를 탐색**
  - 시작 : 루트
  - 이원 탐색 트리가 공백이면, 실패로 끝남
  - 루트의 키값 = x 이면, 탐색은 성공하며 종료
  - 키값 x < 루트의 키값이면, 루트의 왼쪽 서브트리만 탐색
  - 키값 x > 루트의 키값이면, 루트의 오른쪽 서브트리만 탐색

- **연결 리스트로 표현**
  - 노드 구조 :

left	key	right

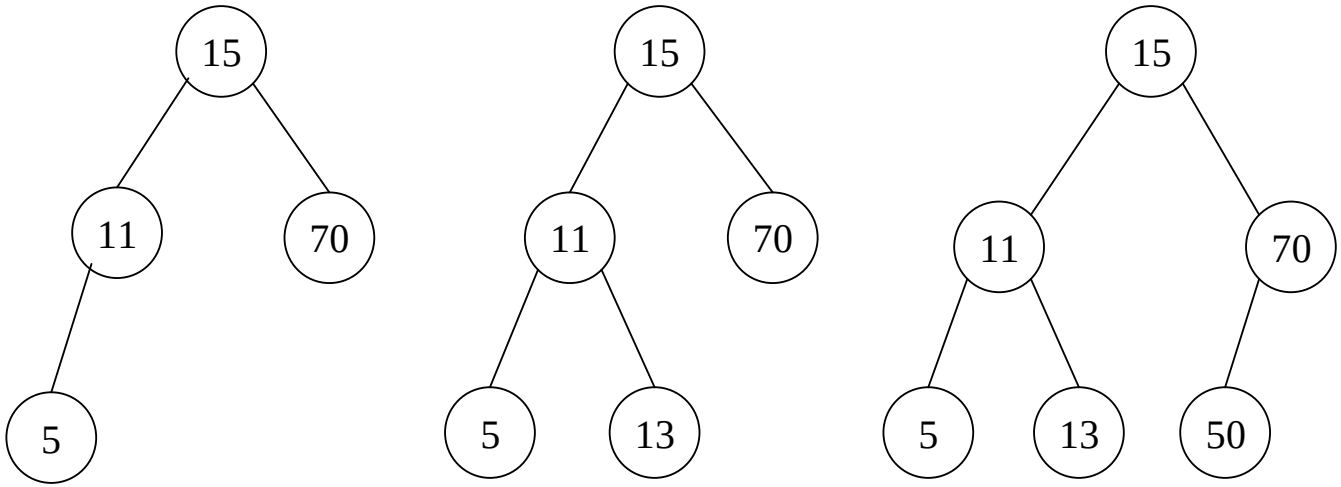
이원 탐색 트리에서의 탐색 (순환적 기술) (2)

- 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)

- **키값이 x인 새로운 원소를 삽입**
  - x를 키값으로 가진 원소가 있는가를 탐색
  - 탐색이 실패하면, 탐색이 종료된 위치에 원소를 삽입
- **예 : 키값 13, 50의 삽입 과정**



(a) 이진 탐색 트리 (b) 원소 13을 삽입 (c) 원소 50을 삽입

이원 탐색 트리에서의 삽입 (2)

- 삽입 알고리즘

```
insertBST(B, x)
    // B는 이원 탐색 트리, x는 삽입할 원소 키 값
    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()
```

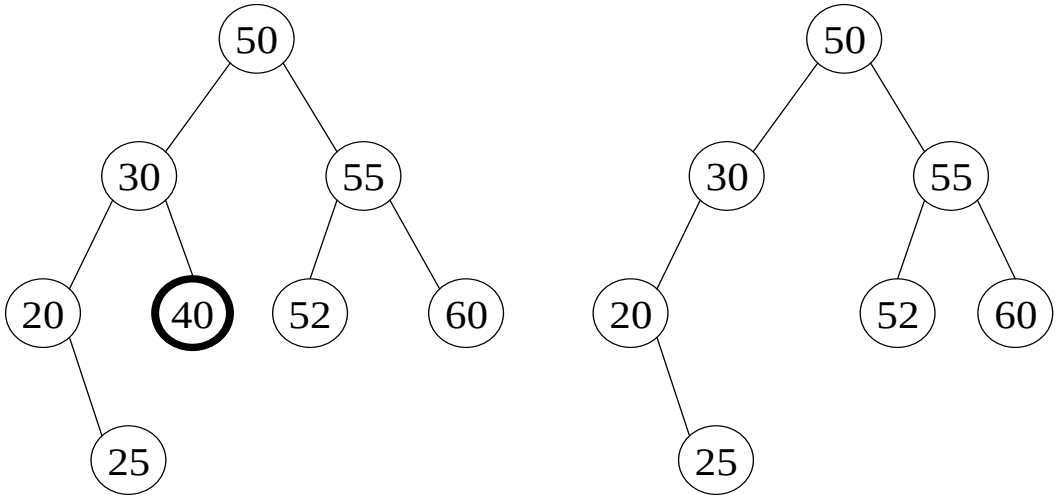
이원 탐색 트리에서의 원소 삭제 (1)

- 삭제하려는 원소의 키값이 주어졌을 때
  - 이 키값을 가진 원소를 탐색
  - 원소를 찾으면 삭제 연산 수행

- 해당 노드의 자식수에 따른 세가지 삭제 연산
  - 자식이 없는 리프 노드의 삭제
  - 자식이 하나인 노드의 삭제
  - 자식이 둘인 노드의 삭제

이원 탐색 트리에서의 원소 삭제 (2)

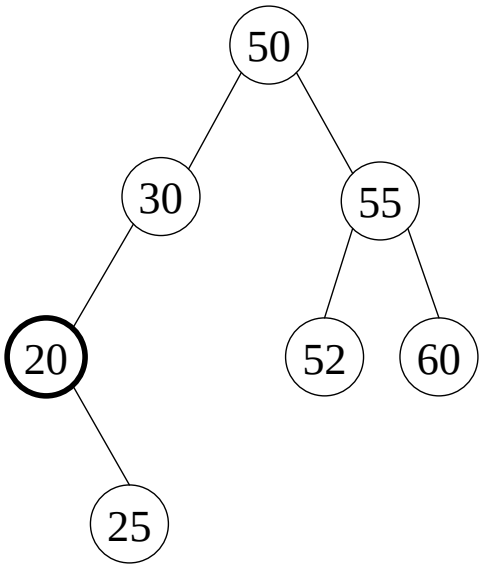
- 자식이 없는 리프 노드의 삭제
  - 부모 노드의 해당 링크 필드를 널(null)로 만들고 삭제한 노드 반환
  - 예 : 키값 40을 가진 노드의 삭제시



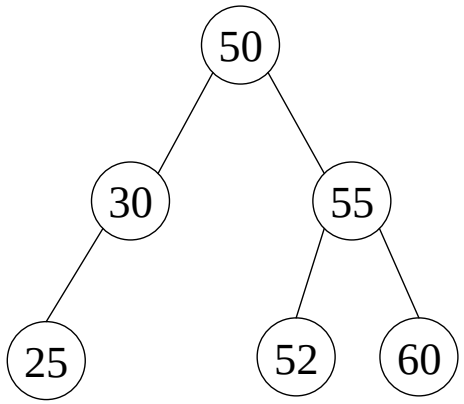
(a) 삭제전 (b) 삭제후

이원 탐색 트리에서의 원소 삭제 (3)

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



(a) 삭제전



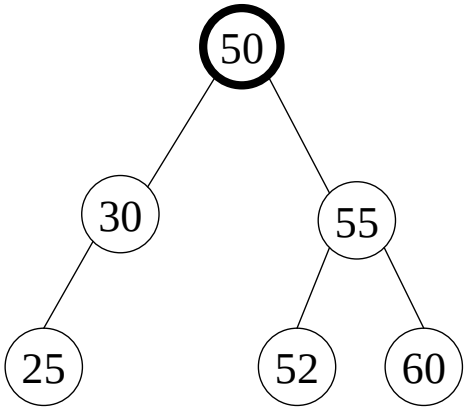
(b) 삭제후

이원 탐색 트리에서의 원소 삭제 (4)

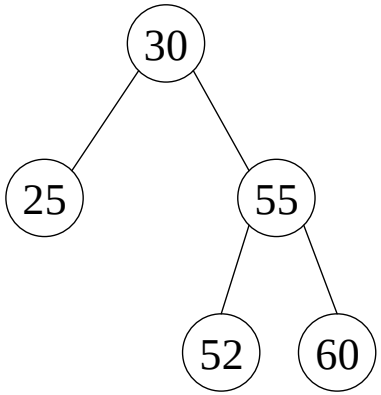
- 자식이 둘인 노드의 삭제
  - 삭제되는 노드 자리
    - 왼쪽 서브트리에서 제일 큰 원소
    - 또는 오른쪽 서브트리에서 제일 작은 원소로 대체
  - 해당 서브트리에서 대체 원소를 삭제
  - 대체하게 되는 노드의 차수는 1 이하가 됨

이원 탐색 트리에서의 원소 삭제 (5)

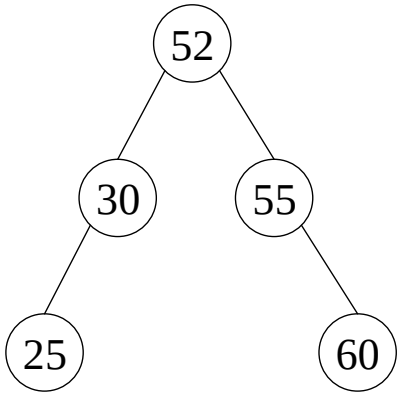
- 예 : 키값이 **50**인 루트 노드의 삭제시



(a) 삭제전



(b) 왼쪽 서브트리의
최대 원소로 대체



(c) 오른쪽 서브트리의
최소 원소로 대체

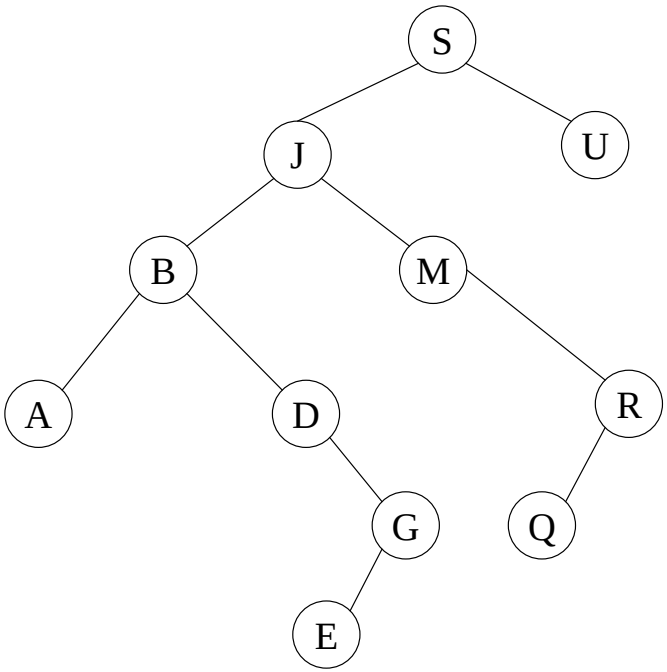
이원 탐색 트리에서의 원소 삭제 (6)

- 삭제 알고리즘의 골격
  (왼쪽 서브트리에서의 최대 원소값으로 대체시)

```
deleteBST(B, x)
    p ← the node to be deleted;       //주어진 키값 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 키값을 가진 원소를 탐색
                        p.key ← q.key;
            deleteBST(p.left, p.key);
    }
end deleteBST()
```

이원 탐색 트리의 Java 구현 및 검색 (1)

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



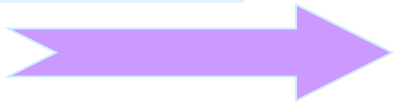
이원 탐색 트리의 Java 구현 및 검색 (2)

- 이원 탐색 트리 구축 및 탐색 프로그램

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


class BinarySearchTree {
    private TreeNode rootNode; ;
```

이원 탐색 트리의 Java 구현 및 검색 (3)

```
private TreeNode insertKey(TreeNode T, String x) {
            // insert() 메소드에 의해 사용되는 보조 순환 메소드
   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;
   }
} // end insertKey()
```



이원 탐색 트리의 Java 구현 및 검색 (4)

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

TreeNode find(String x) {     // 키값 x를 가지고 있는 TreeNode의
    TreeNode T = rootNode;    // 포인터를 반환
    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;
        } // end if
    }
    return T;
} // end find()
```

이원 탐색 트리의 Java 구현 및 검색 (5)

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

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

이원 탐색 트리의 Java 구현 및 검색 (6)

```
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();
        // 스트링 "R"을 탐색하고 프린트
```

이원 탐색 트리의 Java 구현 및 검색 (7)

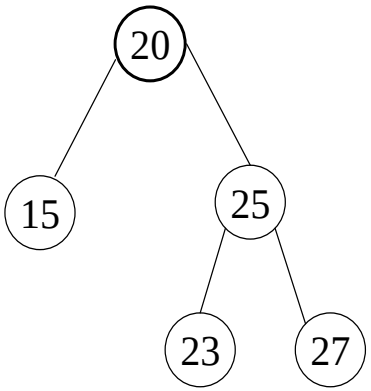
```
        System.out.println(" Search For \"R\"");
         TreeNode N = T.find("R");
         System.out.println("Key of node found = " + N.key);
         System.out.println();
         // 스트링 "C"를 탐색하고 프린트
         System.out.println(" Search For \"C\"");
         TreeNode P = T.find("C");
     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

   } // end BinarySearchTreeTest class
```

이원 탐색 트리의 결합과 분할 (1)

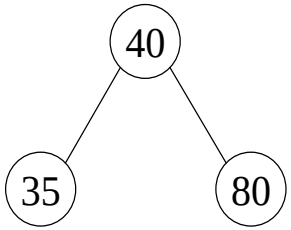
- **3원 결합 : threeJoin (aBST, x, bBST, cBST)**
  - 이원 탐색 트리 aBST와 bBST에 있는 모든 원소들과 키값 x를 갖는 원소를 루트 노드로 하는 이원 탐색 트리 cBST를 생성
  - 가정
    - aBST의 모든 원소 < x < bBST의 모든 원소
    - 결합이후에 aBST와 bBST는 사용하지 않음
- **3원 결합의 연산 실행**
    - 새로운 트리 노드 cBST를 생성하여 key값으로 x를 지정
    - left 링크 필드에는 aBST를 설정
    - right 링크 필드에는 bBST를 설정

이원 탐색 트리의 결합과 분할 (2)

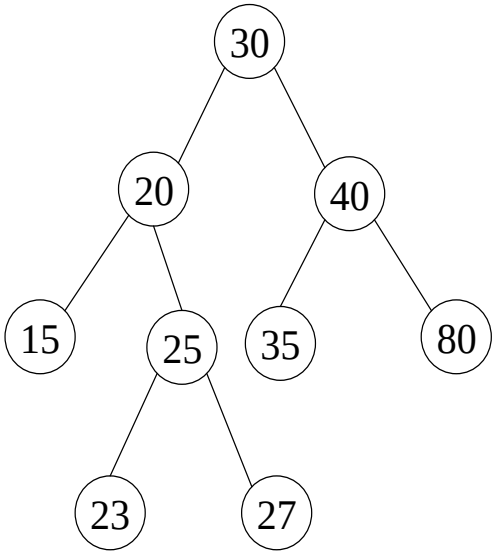
- **3**원 결합의 예



(a) aBST



(b) bBST



(c) cBST

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

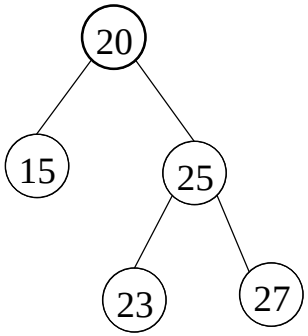
이원 탐색 트리의 결합과 분할 (3)

- **2원 결합 : 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 키값 < bBST의 모든 키값
    - 연산후 aBST와 bBST는 사용 하지 않음

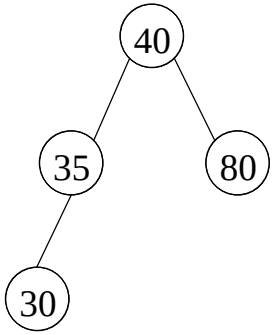
- **2원 결합 연산 실행**
  - aBST나 bBST가 공백인 경우
    - cBST : 공백이 아닌 aBST 혹은 bBST
  - aBST와 bBST가 공백이 아닌 경우
    - 두 이원 탐색 트리 결합 방법은 두가지로 나뉨

이원 탐색 트리의 결합과 분할 (4)

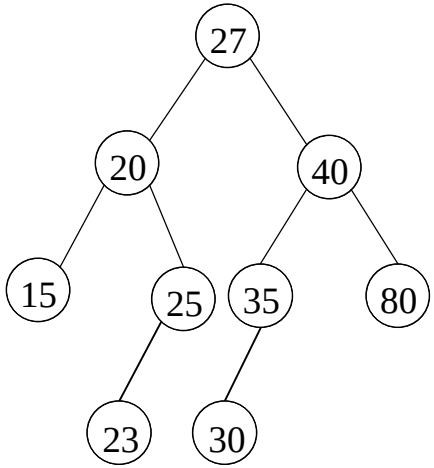
- **aBST**에서 키 값이 가장 큰 원소를 삭제
  - 이 결과 이원 탐색트리 : aBST'
  - 삭제한 가장 큰 키값 : max
  - threeJoin(aBST', max, bBST, cBST) 실행



(a) aBST



(b) bBST

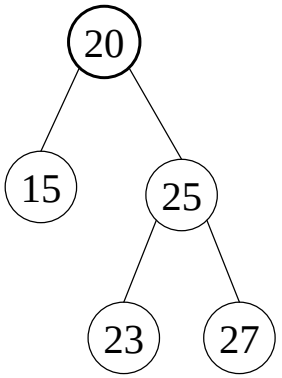


(c) cBST

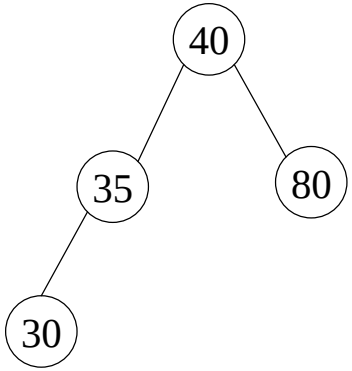
- 실행 시간 : **O(height (aBST))**
- **cBST**의 높이 : **max{height(aBST), height(bBST)} + 1**

이원 탐색 트리의 결합과 분할 (5)

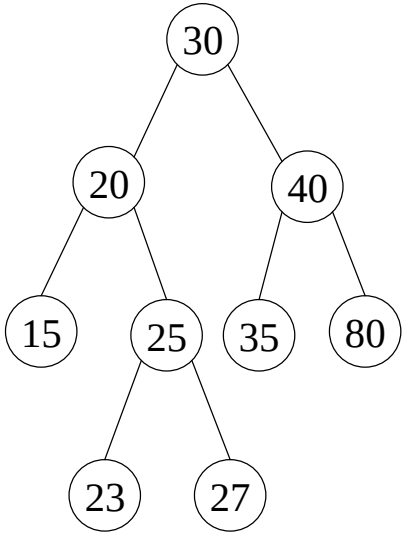
- **bBST**에서 가장 작은 키값을 가진 원소를 삭제
  - 이 결과 이원 탐색 트리 : bBST'
  - 삭제한 가장 작은 키값 : min
  - threeJoin(aBST, min, bBST', cBST)를 실행



(a) aBST



(b) bBST



(c) cBST

이원 탐색 트리의 결합과 분할 (6)

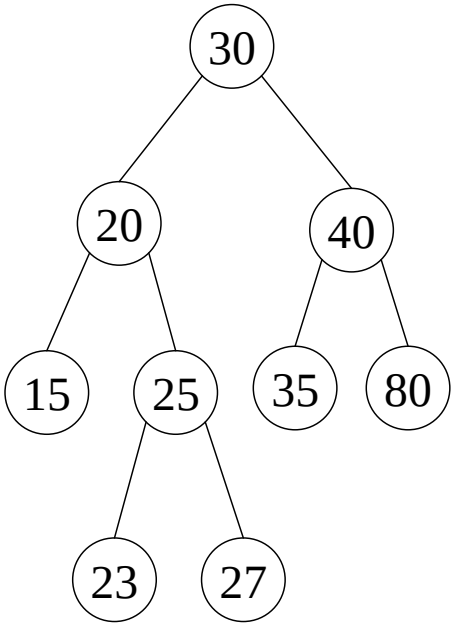
- **분할 : split(aBST, x, bBST, cBST)**
  - aBST 를 주어진 키값 x를 기준으로 두 이원 탐색 트리 bBST와 cBST로 분할
  - bBST : x보다 작은 키값을 가진 aBST의 모든 원소 포함
  - cBST : x보다 큰 키값을 가진 aBST의 모든 원소 포함
  - bBST와 cBST는 각각 이원 탐색 트리 성질을 만족
  - 키값 x가 aBST에 있으면 true 반환 , 아니면 false 반환

이원 탐색 트리의 결합과 분할 (7)

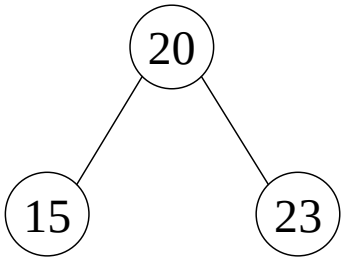
- 분할 연산 실행
  - aBST의 루트 노드가 키값 x를 가질 때
    - 왼쪽 서브트리 : bBST
    - 오른쪽 서브트리 : cBST
    - True 반환
  - x < 루트 노드의 키값
    - 루트와 그 오른쪽 서브트리는 cBST에 속한다.
  - x > 루트 노드의 키값
    - 루트와 그 왼쪽 서브트리는 bBST에 속한다.
  - 키값 x를 가진 원소를 탐색하면서 aBST를 아래로 이동

이원 탐색 트리의 결합과 분할 (8)

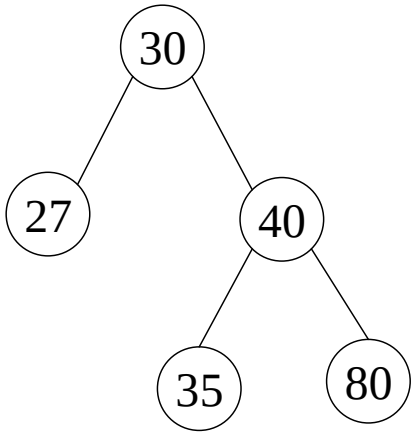
- **Split(aBST , 25 , bBST , cBST)**



(a) aBST



(b) bBST



(c) cBST

이원 탐색 트리의 결합과 분할 (9)

- 분할 연산 알고리즘

```
splitBST(aBST, x, bBST, cBST)
            // x는 aBST를 분할하는 키값
            // 트리 노드는 left, key, right 필드로 구성
Small ← getTreeNode();   // 키값 x보다 큰 원소로 된 BST
Large ← getTreeNode();   // 키값 x보다 작은 원소로 된 BST
S ← Small;   // Small BST의 순회 포인터
L ← Large;   // Large BST의 순회 포인터
P ← aBST;   // aBST의 순회 포인터
```

이원 탐색 트리의 결합과 분할 (10)

- 분할 연산 알고리즘

```
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;   // 키값 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;   // 키값 x는 aBST에 없음
}
end splitBST()
```

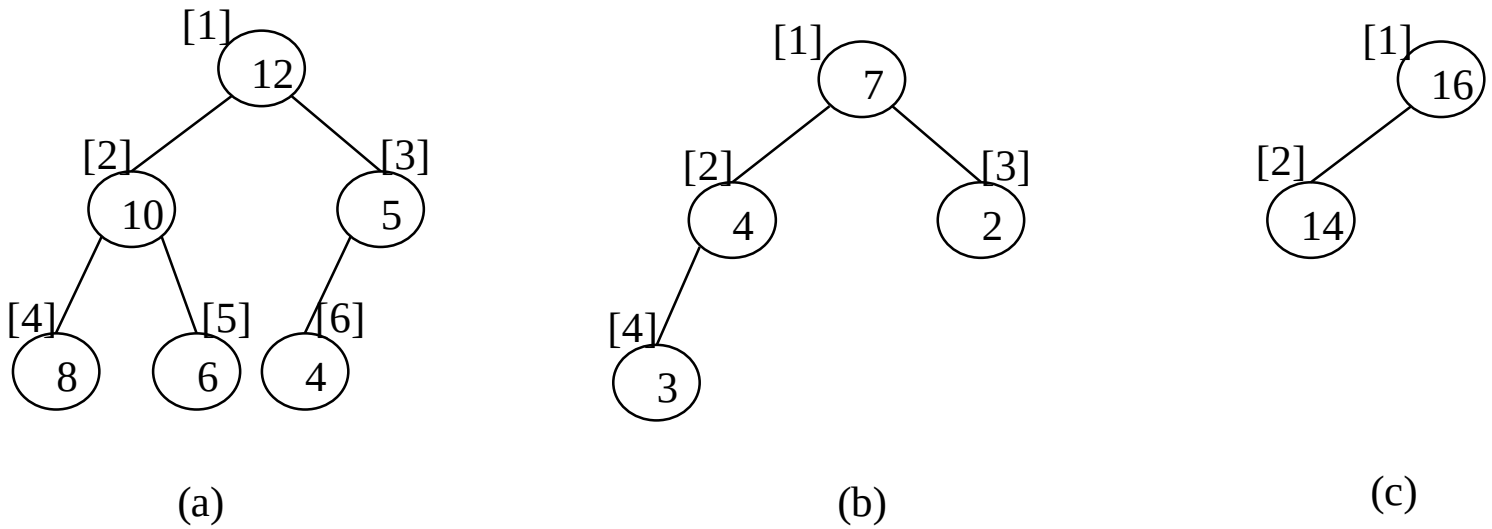
이원 탐색 트리의 결합과 분할 (11)

- **이원 탐색 트리의 높이**
  - 이원 탐색 트리의 높이가 크면
    원소의 검색,삽입,삭제 연산 수행 시간이 길어진다.
  - n개의 노드를 가진 이원 탐색 트리의 최대 높이
    : n - 1
  - 평균적인 이원 탐색 트리의 높이
    : O(log n)

- **균형 탐색 트리 (balanced search tree)**
  - 탐색 트리의 높이가 최악의 경우 O(logn)이 되는 경우
  - 검색, 삽입, 삭제 연산을 O(height)시간에 수행
  - 예) AVL , 2-3, 2-3-4 , red-black, B-tree

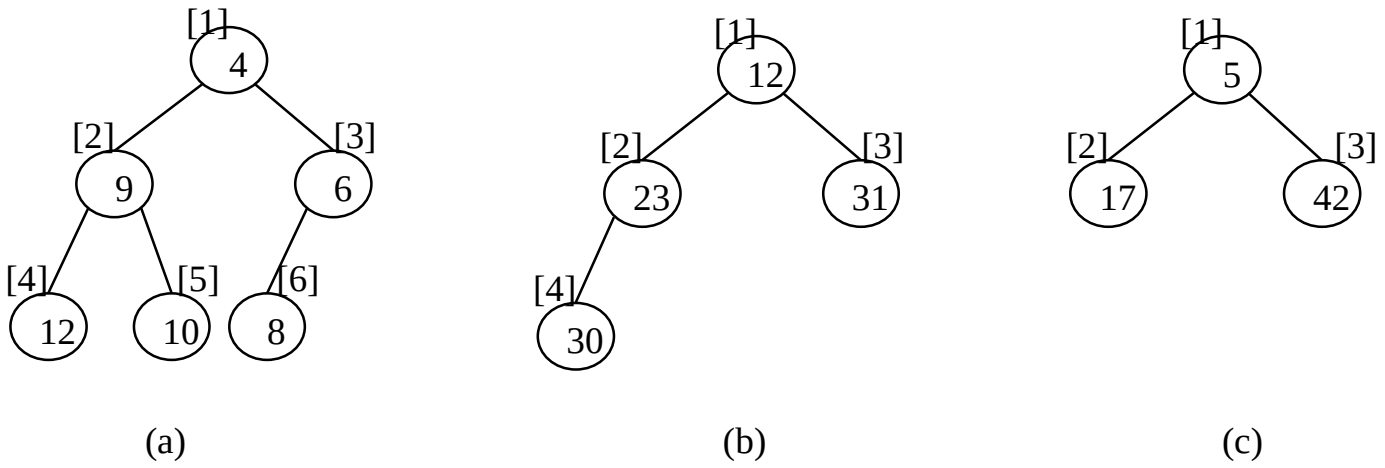
히프 (1)

- 특성
  - 최대 히프(max heap): 각 노드의 키값이 그 자식의 키값보다 작지 않은 완전 이진 트리
  - 최소 히프(min heap): 각 노드의 키값이 그 자식의 키값보다 크지 않은 완전 이진 트리
  - 최대 히프 예



히프 (2)

- 최소 히프 예



- 최소 히프의 루트는 그 트리에서 가장 작은 키값 가짐
- 최대 히프의 루트는 그 트리에서 가장 큰 키값 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) : 히프로부터 키값이 가장 큰 원소를 삭제하고 반환

히프 추상 데이타 타입 (2)

ADT Heap

데이타 : n>0 원소로 구성된 완전 이진 트리로 각 노드의 키값은 그의 자식 노드의 키값보다 작지 않다.

연산 :

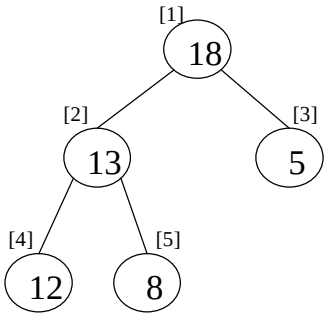
H∈Heap; e ∈Element;
createHeap() := create an empty heap;
insertHeap(H,e) := insert a new item e into H
isEmpty(H) := if the heap H is empty
then return true
else return false
deleteHeap(H) := if isEmpty(H) then null
else {
e ← the largest element in H;
remove the largest element in H;
return e;
}

End Heap

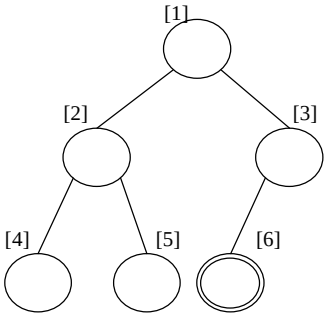
히프에서의 삽입 (1)

- 삽입 예
  - 5개의 원소로 된 히프의 예(그림 (a))
  - 이중 원으로 표현한 노드: 새로 확장될 노드의 위치 (그림 (b))
  - 키값 3 삽입시: 바로 완료 (그림 (c))
  - 키값 8 삽입시: 부모 노드와 교환후 완료 (그림 (d), (e))
  - 키값 19 삽입시: 부모 노드와 루트와 연속 교환후 완료 (그림 (f), (g), (h))

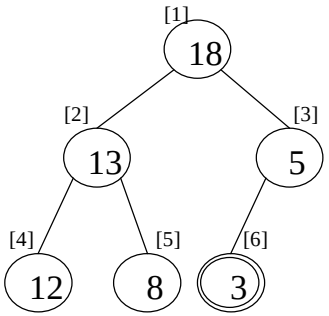
히프에서의 삽입 (2)



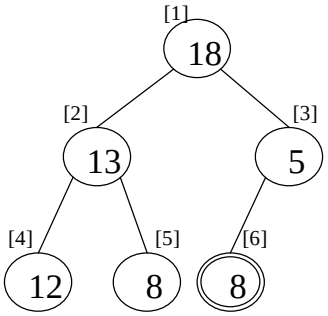
(a)히프



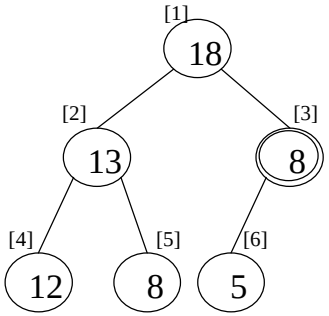
(b) 삽입후의 히프구조



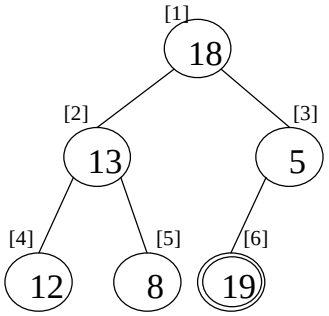
(c) 원소 3을 삽입



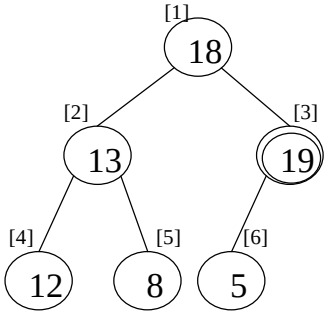
(d)히프 (a)에 원소 8 삽입



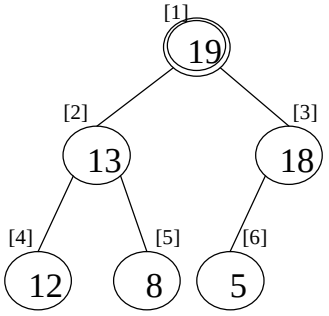
(e) 원소 8을 삽입 후



(f) 히프 (a)에 원소 19삽입



(g) 원소의 이동



(h) 원소 19 삽입 후

히프에서의 삽입 (3)

- 부모 노드 위치 결정 가정
  - 연결 표현 사용시: 각 노드에 parent 필드 추가
  - 순차 표현 사용시: 위치 i의 부모노드는 ⌊i / 2⌋
- 히프에 대한 삽입 알고리즘

```
insertHeap(Heap,e)
        // 순차 표현으로 구현된 최대 히프
        // 원소 e를 히프 Heap에 삽입, n은 현재 히프의 크기(원소 수)
   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(e.key ≤ Heap[ ⌊i/2⌋ ].key) then exit;  // 삽입할 노드의 키값과
                                                // 부모 노드 키값을 비교
      Heap[i] ← Heap[ ⌊i/2⌋ ];  // 부모 노드 키값을 자식노드로 이동
      i ← ⌊i/2⌋;
   }
   Heap[i] ← e;

end insertHeap()
```

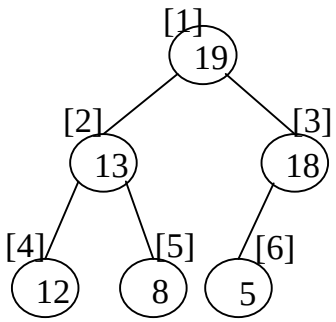
히프에서의 삭제 (1)

- 루트 원소를 삭제
- 나머지 트리가 다시 히프가 되도록 재조정

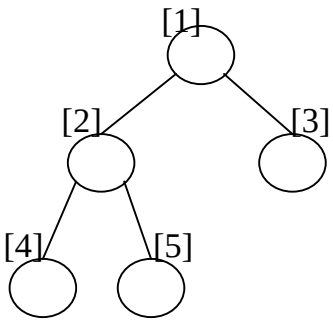
히프에서의 삭제 (2)

- 첫 번째 삭제 예 (루트 키값 19)
  - 히프의 예 (그림 (a))
  - 루트 원소 19의 삭제후 구조 (그림 (b))
  - 위치 6의 노드를 삭제하고 키값 5를 루트로 이동 (그림 (c))
  - 이 이동 원소값을 두 자식 노드 중에서 큰 원소값과 비교
    - 비교 결과 자식 노드 원소값이 크면 그 값을 부모 노드로 옮긴 다음 이동 원소가 그 자식 노드로 내려가서 다시 히프 검사
    - 만일 자식 노드 원소보다 크면 삭제 연산 종료
    - 이 예의 경우, 5와 18 교환후 종료 (그림 (d))
- 두번째 삭제 예 (루트 키값 18)
  - 삭제(루트 18)후 구조 (그림 (e))
  - 위치 5의 노드를 삭제하고 키값 8을 루트로 이동 (그림 (f))
  - 이 이동 원소값을 두 자식 노드 중에서 큰 원소값과 비교
    - 이 예의 경우, 8과 13, 다시 8과 12 교환후 종료 (그림 (g), (h))

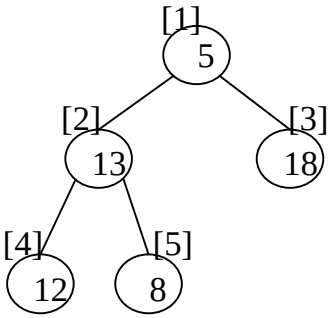
히프에서의 삭제 (3)



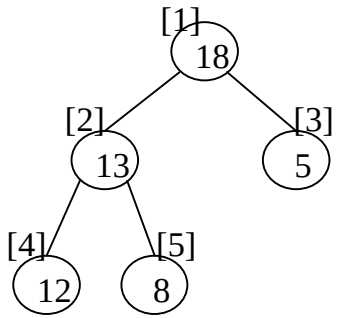
(a)히프



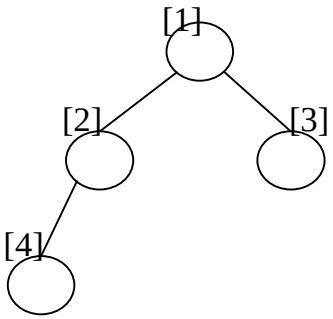
(b)삭제 후의 히프 구조



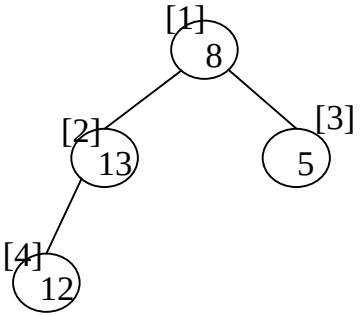
(c)삭제 중간 단계



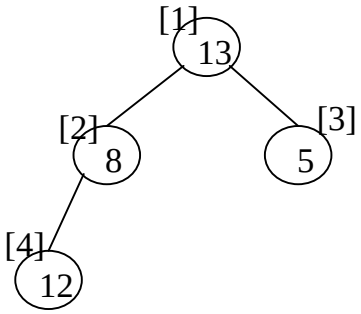
(d)첫번째 삭제 뒤의 히프



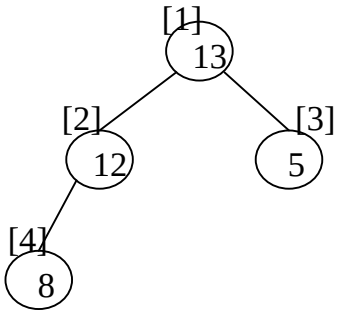
(e)두번째 삭제 뒤의 히프 구조



(f)삭제 중간 단계



(g)삭제 중간 단계



(h)두번째 삭제가 완료된 히프

히프에서의 삭제 (4)

- 히프에서의 삭제 알고리즘

```
deleteHeap(heap)
        // 히프로부터 원소 삭제, n은 현재의 히프 크기(원소 수)
    if (n=0) then return error;   // 공백 히프
    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)

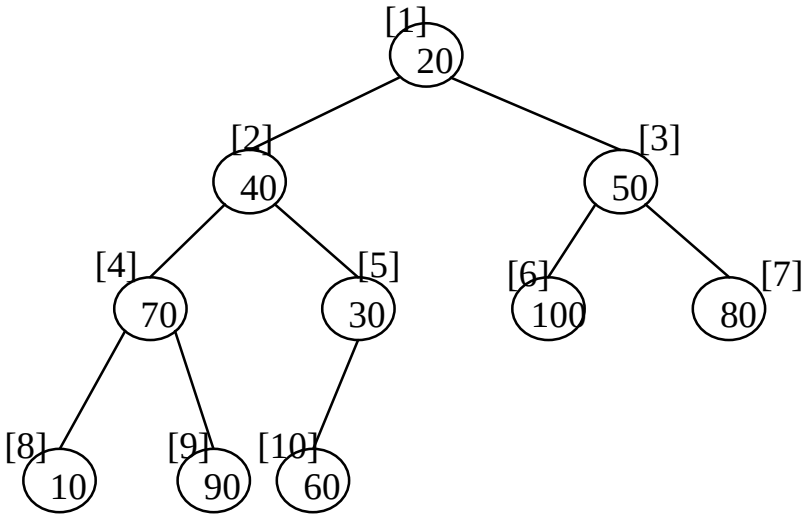
- 초기에 히프로 되어 있지 않은 완전이원트리 H를 히프로 변환
  - 역레벨순서로 H의 내부 노드 각각을 루트로 하는 서브트리를 차례로 히프로 만들어 나가면 됨

- 완전 이원트리를 히프로 변환하는 알고리즘

```
makeTreeHeap(H, n)
    // H는 히프가 아닌 완전 이진 트리
  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]) exit;
      temp ← H[p];
      H[p] ← H[j];
      H[j] ← temp
      p ← j;  // 부모 노드를 한 레벨 밑으로 이동
    }
  }
end makeTreeHeap()
```

완전 이진 트리를 히프로 변환 (2)

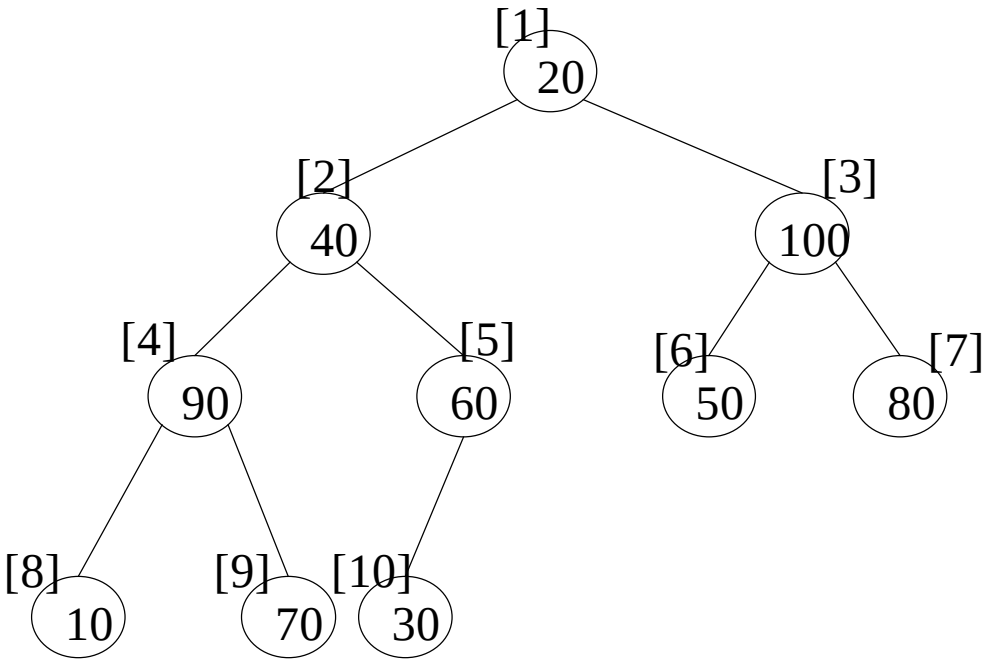
- 변환예
  - 히프가 아닌 완전 이진 트리



  - 내부 노드의 역레벨순서: 5, 4, 3, 2, 1
  - 5번 노드를 루트로 하는 서브 트리에서 히프 연산 시작
    - 이 노드는 루트보다 키값(60)이 큰 자식을 가지므로 교환(30↔60)
  - 다음으로 4번 노드를 루트로 하는 서브 트리 조사
    - 자식 중에 큰 키값(90)을 루트의 키값과 교환(70 ↔ 90)

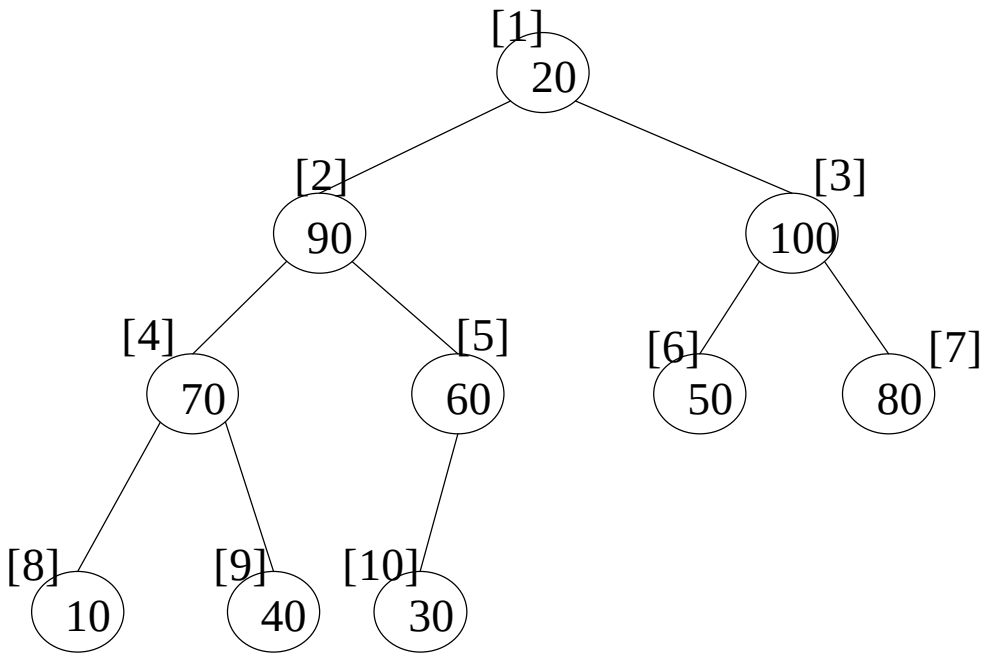
완전 이진 트리를 히프로 변환 (3)

- 다음으로 3번 노드를 루트로 하는 서브 트리 조사
  - 노드6과 노드 3과의 큰 키값 교환(50 ↔ 100)
  - 여기까지의 과정에서 얻어진 트리



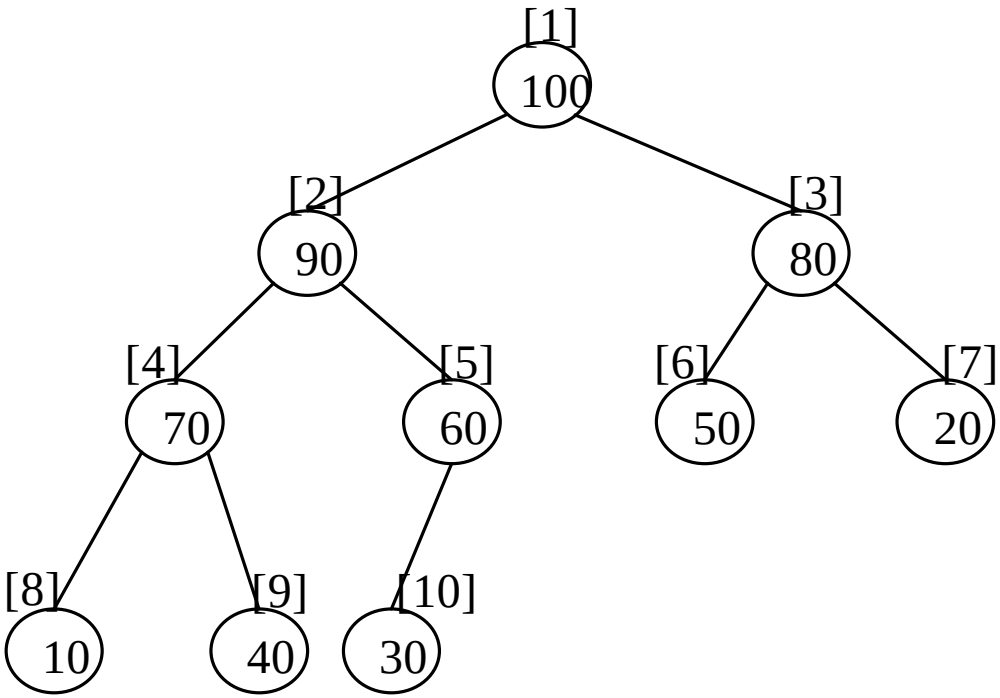
완전 이진 트리를 히프로 변환 (4)

- 다음으로 2번 노드를 루트로 하는 서브트리 조사
  - 키값 40을 왼쪽 자식의 키값(90)과 교환(40 ↔ 90)
  - 다시 계속해서 9번 노드의 키값(70)과 교환(40 ↔ 70)
  - 이 결과로 얻어진 이진 트리



완전 이진 트리를 히프로 변환 (5)

- 끝으로 역레벨순서 마지막 노드인 루트 노드(1번 노드)를 고려
  - 이 루트 노드는 먼저 3번 노드의 키값(100)과 교환(20 ↔ 100)
  - 다시 계속해서 7번 노드의 키값(80)과 교환(20 ↔ 80)
  - 최종 히프 구조



히프를 이용한 우선 순위큐 표현 (1)

- 히프는 우선 순위큐 표현에 효과적
- 우선 순위가 제일 높은 원소를 찾거나 삭제하는 것은 아주 쉬움
  - 노드 삭제시: 나머지 노드들을 다시 히프가 되도록 재조정
  - 노드 삽입시: 삽입할 원소의 우선 순위에 따라 히프가 유지되도록 해야 됨

히프를 이용한 우선 순위큐 표현 (2)

- 배열(순차표현)으로 구현된 히프로 표현한 우선 순위큐(PriorityQueue) 클래스
  - 6.7.2와 6.7.3절에서 사용된 다른 PriorityQueue 클래스 구현과 대체 가능
  - 우선순위큐 정렬 메소드를 정의한 프로그램 6.6에 대해 PriorityQueue 클래스를 사용시 히프정렬(heapsort) 버젼이 됨
    - 6.7.1절의 우선순위큐 정렬: $O(n^2)$
    - 히프정렬: $O(n \log n)$

- 우선순위큐 정렬: **$O(n^2)$**
- 히프정렬: **$O(n \log n)$**

히프를 이용한 우선 순위큐 표현 (3)

- 히프로 표현한 우선순위 큐 클래스

```
class PriorityQueue{
    private int count;              // 우선순위 큐의 현재 원소수
    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;
    }
```

히프를 이용한 우선 순위큐 표현 (4)

```
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
} // end insert()
```

히프를 이용한 우선 순위큐 표현 (5)

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

public PriorityElement delete() {
            // 우선순위 큐로부터 원소 삭제
  if (count == 0) {              // 우선순위 큐가 공백인 경우
      return null;
  }
  else {
        // 변수 선언
      int currentLoc;
      int childLoc;
      PriorityElement itemToMove;   // 이동시킬 원소
      PriorityElement deletedItem;    // 삭제한 원소
      deletedItem = itemArray[1];        // 삭제하여 반환할 원소
```

히프를 이용한 우선 순위큐 표현 (6)

```
        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;
            }
        } // end while
        itemArray[currentLoc] = itemToMove;     // 최종 위치에 원소 저장
        return deletedItem;
      } // end if
    } // end delete()
} // end PriorityQueue class
```

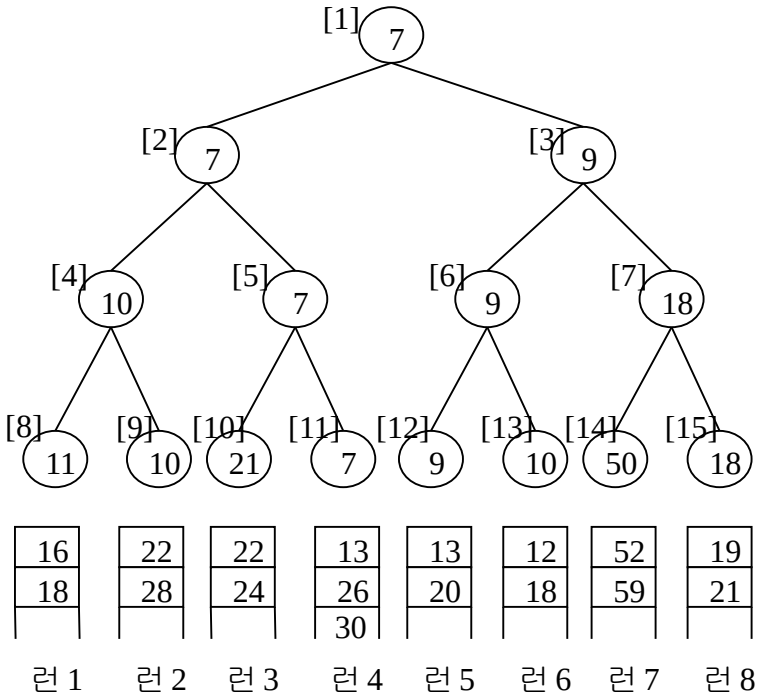
heap_adt.java

선택 트리 (1)

- 개요
  - k개의 런에 나뉘어져 있는 n개의 원소들을 하나의 순서순차로 합병하는 경우
    - 런(run): 원소들이 정렬되어 있는 순서순차(ordered sequence)
    - 각 런은 키(key)값에 따라 원소들을 오름차순으로 정렬
    - K개의 런 중에서 가장 작은 키값을 가진 원소를 계속적으로 순서순차로 출력
      - k개의 원소 중에서 가장 작은 키 값을 가진 원소를 선택: k-1번 비교
      - 선택 트리(selection tree) 자료 구조 이용시: 비교회수 줄임
  - 선택트리의 종류
    - 승자트리
    - 패자트리

선택 트리 (2)

- 승자 트리(**winner tree**)
  - 특징
    - 완전 이원트리
    - 각 단말 노드는 각 런의 최소키 값 원소를 나타냄
    - 내부 노드는 그의 두 자식 중에서 가장 작은 키 값을 가진 원소를 나타냄
  - 런이 8개(k=8)인 경우 승자 트리 예

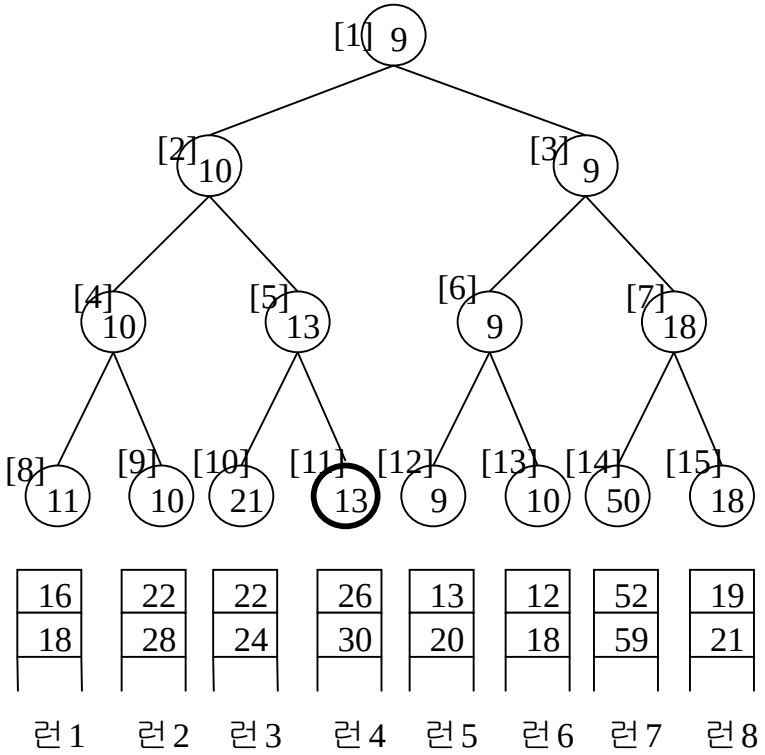


선택 트리 (3)

- 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 값 가진 원소
- 승자 트리의 표현
  - 승자트리는 완전 이원트리이기 때문에 순차 표현이 유리
  - 인덱스 값이 i인 노드의 두 자식 인덱스는 2i와 2i+1
- 합병의 진행
  - 루트가 결정되는 대로 순서순차에 출력 (여기선 7)
  - 다음 원소 즉 키값이 13인 원소가 승자트리로 들어감
  - 승자 트리를 다시 재구성
    - 노드 11에서부터 루트까지의 경로를 따라가면서 형제 노드간 토너먼트 진행

선택 트리 (4)

- 다시 만들어진 승자트리의 예



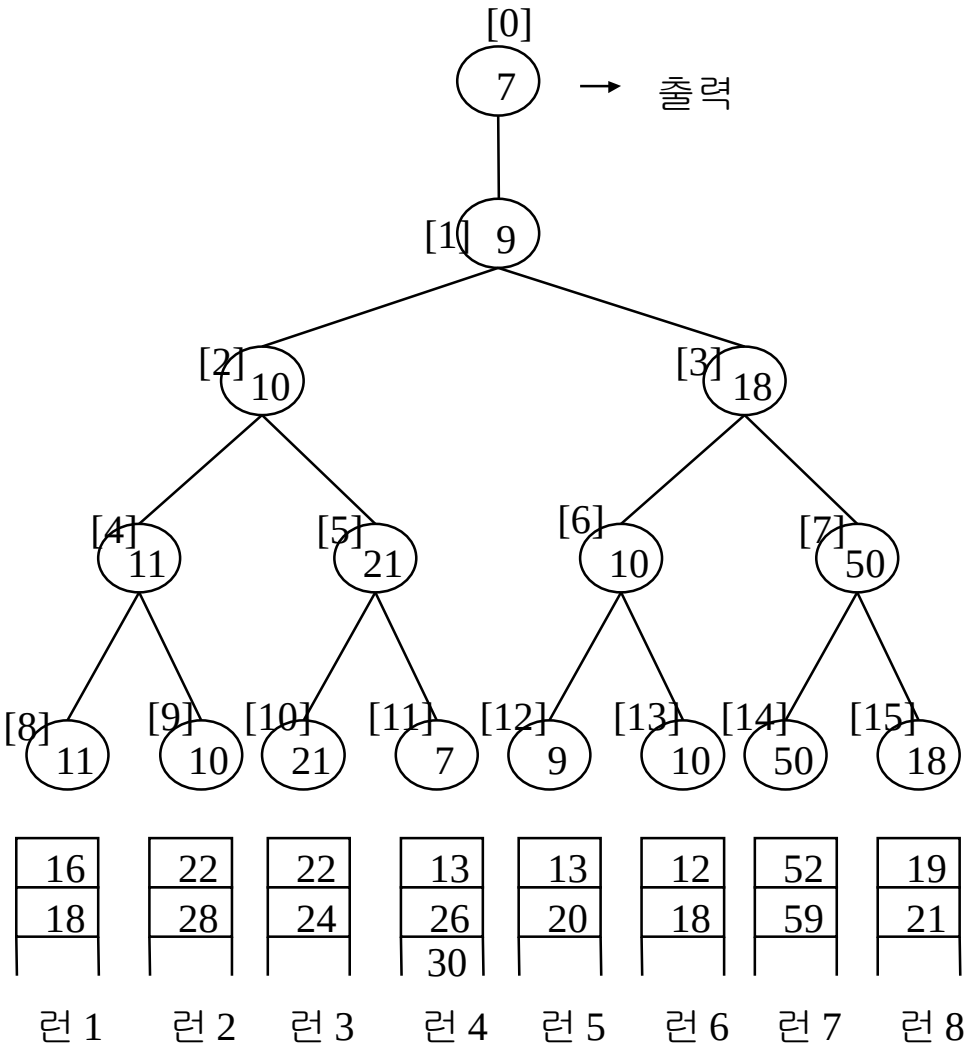
- 이런 방식으로 순서 순차구축을 계속함

선택 트리 (5)

- **패자 트리(loser tree)**
  - 루트 위에 0번 노드가 추가된 완전 이원트리
    - 성질
      (1) 단말노드 :각 런의 최소 키값을 가진 원소
      (2) 내부 노드 : 토너먼트의 승자대신 패자 원소
      (3) 루트(1번 노드) : 결승 토너먼트의 패자
      (4) 0번 노드 : 전체 승자(루트 위에 별도로 위치)

선택 트리 (6)

- 런이 8개(k=8)인 패자 트리의 예

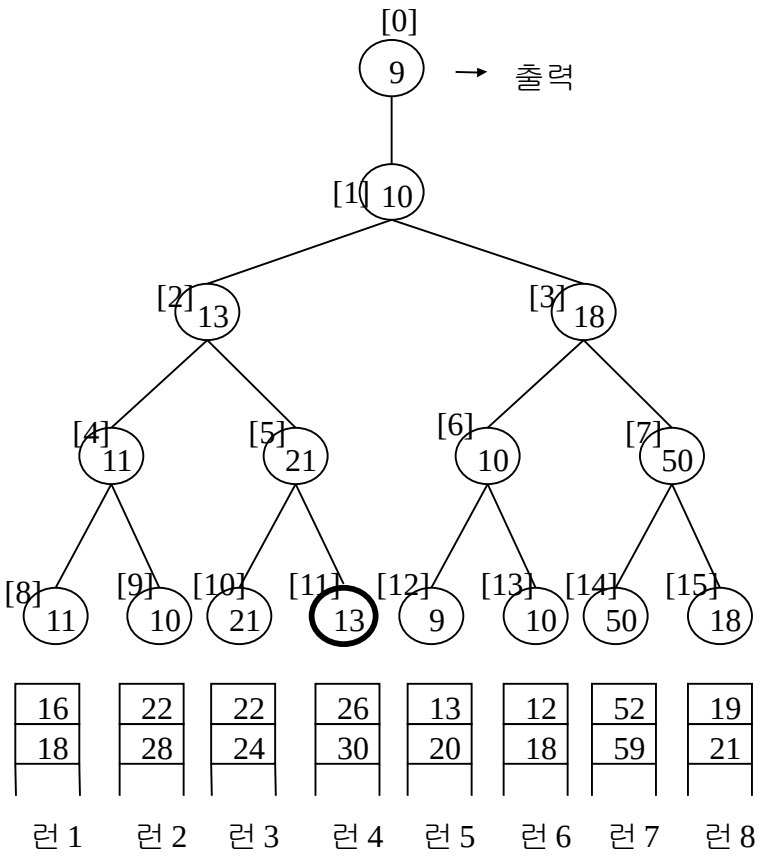


선택 트리 (7)

- 패자 트리 구축 과정
  - 단말 노드: 각 런의 최소 키값 원소
  - 내부 노드
    - 두 자식 노드들이 부모노드에서 토너먼트 경기를 수행
    - 패자는 부모 노드에 남음
    - 승자는 그 위 부모 노드로 올라가서 다시 토너먼트 경기를 계속
  - 1번 루트 노드
    - 마찬가지로 패자는 1번 루트 노드에 남음
    - 승자는 전체 토너먼트의 승자로서 0번 노드로 올라가 순서순차에 출력됨
- 합병의 진행
  - 출력된 원소가 속한 런 4의 다음 원소, 즉 키값이 13인 원소를 패자트리
  - 노드 11에 삽입
  - 패자 트리를 다시 재구성
    - 토너먼트는 노드 11에서부터 루트 노드 1까지의 경로를 따라 경기를 진행
    - 다만 경기는 형제 노드 대신 형식상 부모 노드와 경기를 함

선택 트리 (8)

- 다시 만들어진 패자 트리의 예



◆ 모든 원소가 순서 순차에 출력될때까지 이 과정을 반복