제7장 우선순위큐

Binary Heap, Huffman Code

우선순위큐(Priority Queue)

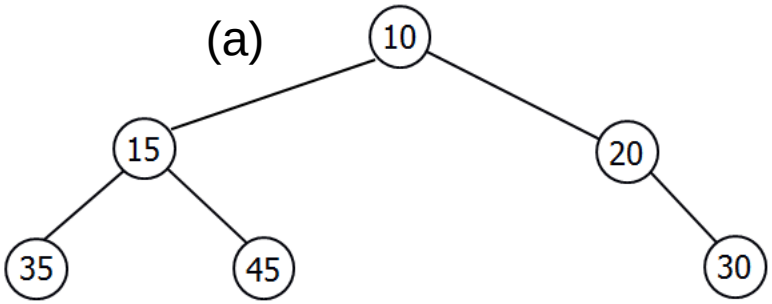
- **가장 높은 우선순위를 가진 항목에 접근, 삭제와 임의의 우선순위를 가진 항목을 삽입을 지원하는 자료구조**
- **스택이나 큐도 일종의 우선순위큐**
  - 스택: 가장 마지막으로 삽입된 항목이 가장 높은 우선순위를 가지므로, 최근 시간일수록 높은 우선순위를 부여
  - 큐: 먼저 삽입된 항목이 우선순위가 더 높다. 따라서 이른 시간일수록 더 높은 우선순위를 부여
- **스택과 큐와 같은 우선순위큐가 있는데, 왜 또 다른 우선순위큐 자료구조가 필요할까?**
  - 삽입되는 항목이 임의의 우선순위를 가지면 스택이나 큐는 새 항목이 삽입될 때마다 저장되어 있는 항목들을 우선순위에 따라 정렬해야 하는 문제점

7.1 이진힙

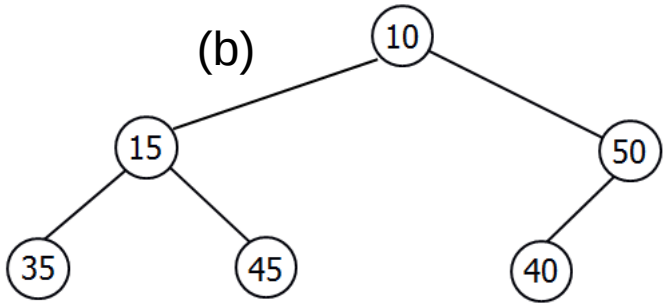
[정의] 이진힙(Binary Heap)은 완전이진트리로서 부모의 우선순위가 자식의 우선순위보다 높은 자료구조

어느 트리가 이진힙일까?

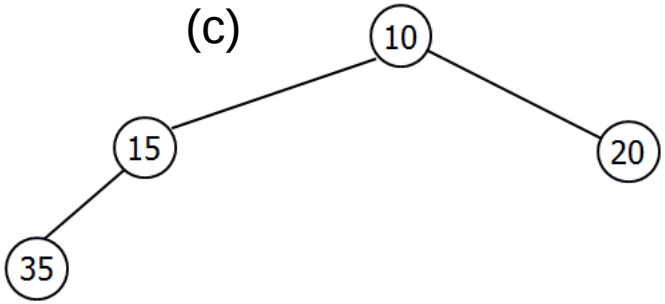
(a)



(b)



(c)



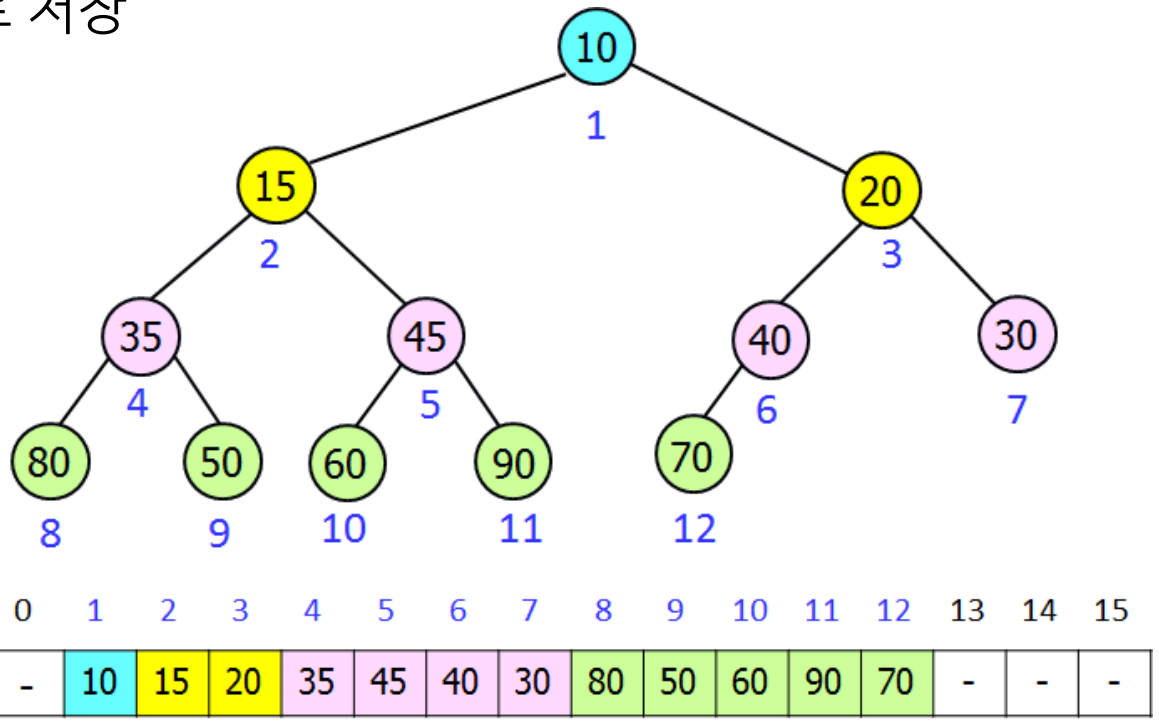
(a) 모든 노드들이 힙속성을 만족하지만 완전이진트리가 아님

(b) 루트의 오른쪽 자식인 50이 40을 자식으로 가지고 있기 때문에 힙속성에 위배

(c) 이진힙

이진힙의 구현

- **완전이진트리는 1차원 배열로 구현**
  - 배열의 2 번째 원소부터 사용. 배열 a에서 a[0]은 사용하지 않음
  - 완전이진트리의 노드들을 레벨순회(Level order Traversal) 순서에 따라 a[1] 부터 차례로 저장



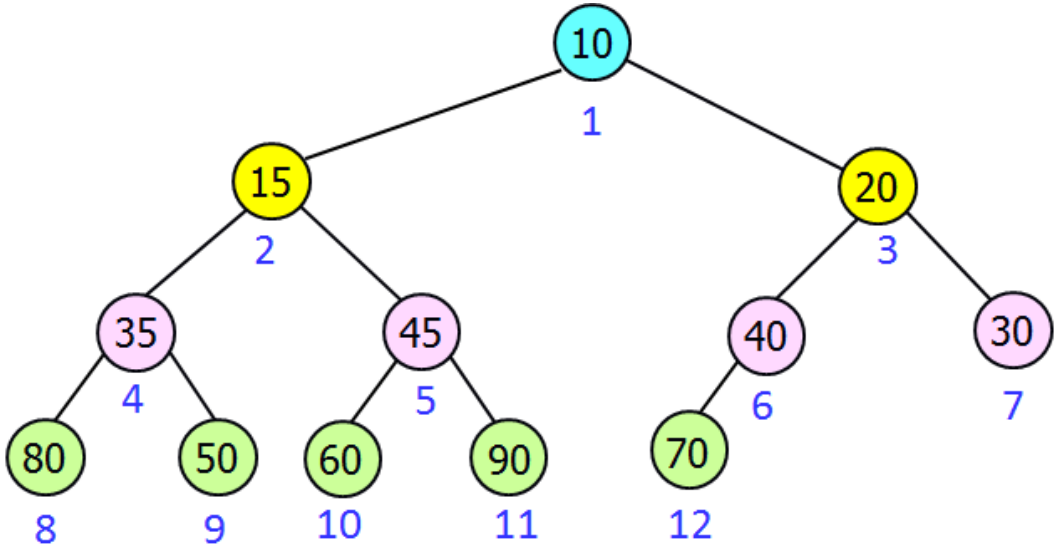
각 노드 아래의 숫자: 노드가 저장된 배열 원소의 인덱스

이진힙의 구현

- **노드들을 배열에 이와 같이 저장했을 때 힙에서 부모와 자식 관계**

  - a[i]의 자식은 a[2i]와 a[2i+1]에 있고,
  - a[j]의 부모는 a[j/2]에 있다. 단, j > 1이고, j/2의 정수만을 취한다.

  - 노드 35의 자식은 a[2x4] =a[8]과 a[2x4+1] = a[9], 즉, 80과 50
  - 노드 90의 부모는 a[11/2] = a[5]의 노드 45



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	10	15	20	35	45	40	30	80	50	60	90	70	-	-	-

이진힙의 종류

- **이진힙의 종류:**
  - 최소힙(Minimum Heap): 키 값이 작을수록 높은 우선순위
  - 최대힙(Maximum Heap): 키 값이 클수록 더 높은 우선순위
- **최소힙의 루트노드에는 항상 가장 작은 키가 저장됨**
  - 부모 노드에 저장된 키가 자식 노드의 키보다 작다는 규칙 때문
  - 루트는 a[1]에 있으므로, O(1) 시간에 min 키를 가진 노드 접근

Entry 클래스

```
01 public class Entry <Key extends Comparable<Key>, Value>{
02     private Key   ky;
03     private Value val;
04     public Entry (Key newKey, Value newValue){ //생성자
05         ky  = newKey;
06         val = newValue;
07     }
08     // get,set 메소드들
09     public Key   getKey()   { return ky;}
10     public Value getValue() { return val;}
11     public void  setKey(Key newKey)       { ky  = newKey;}
12     public void  setValue(Value newValue) { val = newValue;}
13 }
```

- **Entry 객체는 키와 키의 관련 정보를 가지며**
  - Line 01: 2 개의 키를 compareTo() 메소드를 통해 비교하기 위해 Comparable 인터페이스를 사용

BHeap 클래스

```
01 public class BHeap<Key extends Comparable<Key>, Value> {
02     private Entry[] a;  // a[0]은 사용 안함
03     private int N;      // 힙의 크기
04     public BHeap(Entry[] harray, int initialSize) { // 생성자
05         a = harray;
06         N = initialSize;
07     }
08     public int size() { return N; }   // 힙의 크기 리턴
09     private boolean greater(int i, int j) { // 키 비교
10         return a[j].getKey().compareTo(a[i].getKey()) < 0;  }
11     private void swap(int i, int j) { // a[i]와 a[j]를 교환
12         Entry temp = a[i];   a[i] = a[j];   a[j] = temp; }

       // createheap, 삽입, 최솟값 삭제, downheap, upheap
       // 메소드들 선언
   }
```

- Line 04 ~ 07: Bheap 객체 생성자
- BHeap 객체는 Entry 타입의 1차원 배열을 가지며 배열에는 N개의 항목 저장
- Line 08: size() 메소드는 힙의 크기를 리턴
- Line 09 ~ 10: a[i]가 a[j]보다 크면 true를 리턴
- Line 11 ~ 12: a[i]와 a[j]를 서로 교환

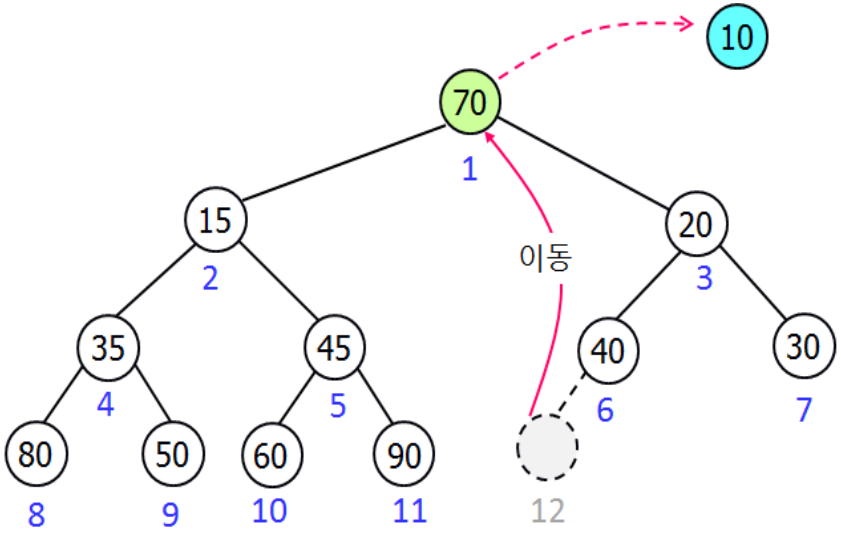
최솟값 삭제(delete_min)

- **루트의 키를 삭제**

[1] 힙의 가장 마지막 노드, 즉, 배열의 가장 마지막 원소를 루트로 옮기고,
[2] 힙 크기를 1 감소시킨다.
[3] 루트로부터 자식들 중에서 작은 값을 가진 자식 (두 자식 사이의 승자) 과 키를 비교하여 힙속성이 만족될 때까지 키를 교환하며 이파리 방향으로 진행

  - [3]의 과정은 루트노드로부터 아래로 내려가며 진행되므로 downheap이라 부름

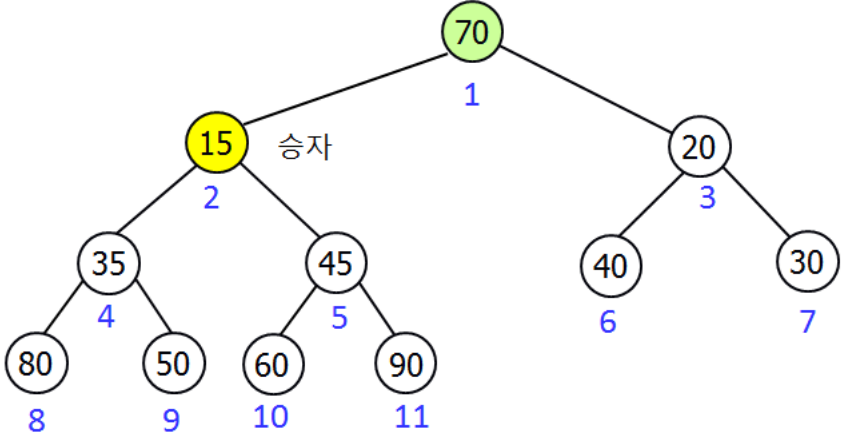
최솟값 삭제 과정



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	70	15	20	35	45	40	30	80	50	60	90	-	-	-	-

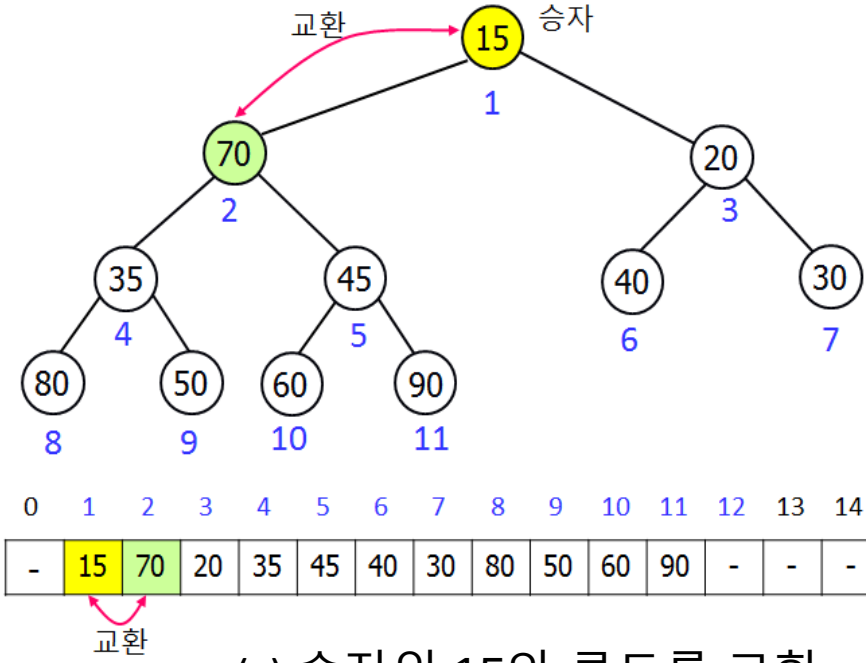
이동

(a) 마지막 노드를 루트로 이동



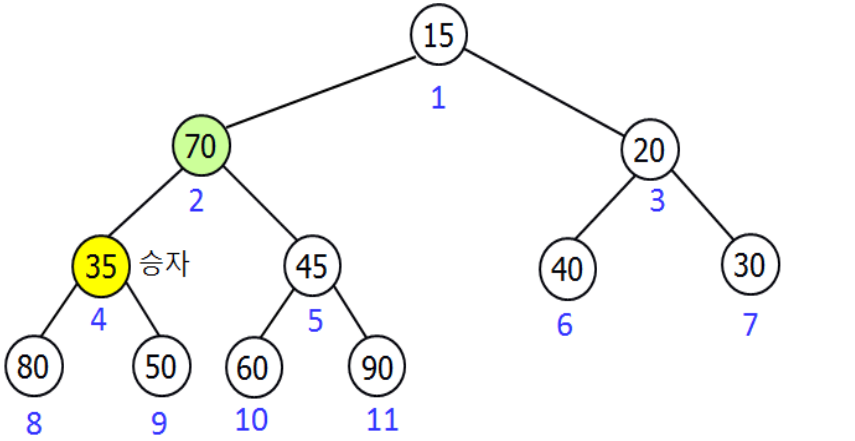
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	70	15	20	35	45	40	30	80	50	60	90	-	-	-	-

(b) 15와 20 중에15가 승자



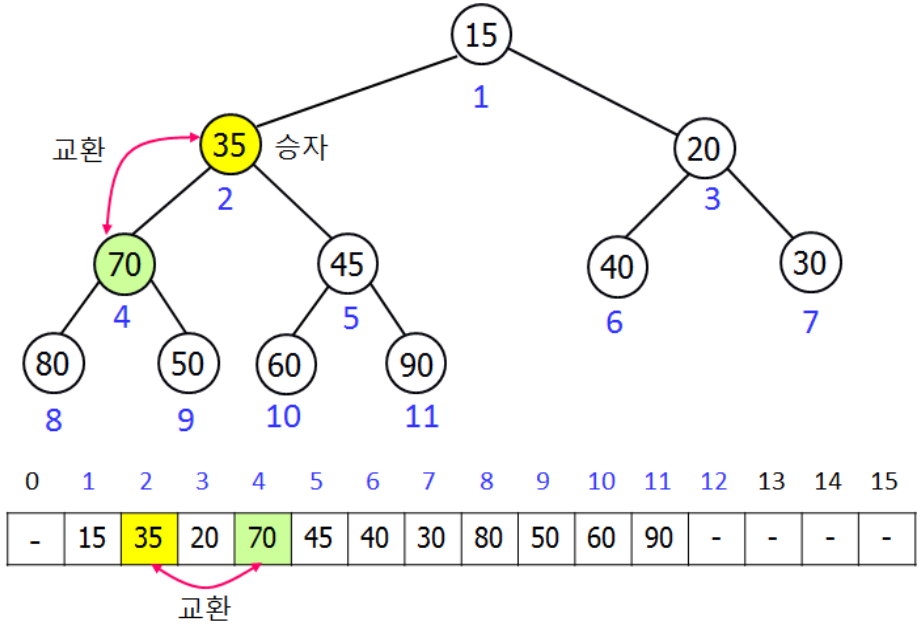
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	15	70	20	35	45	40	30	80	50	60	90	-	-	-	-

(c) 승자인 15와 루트를 교환



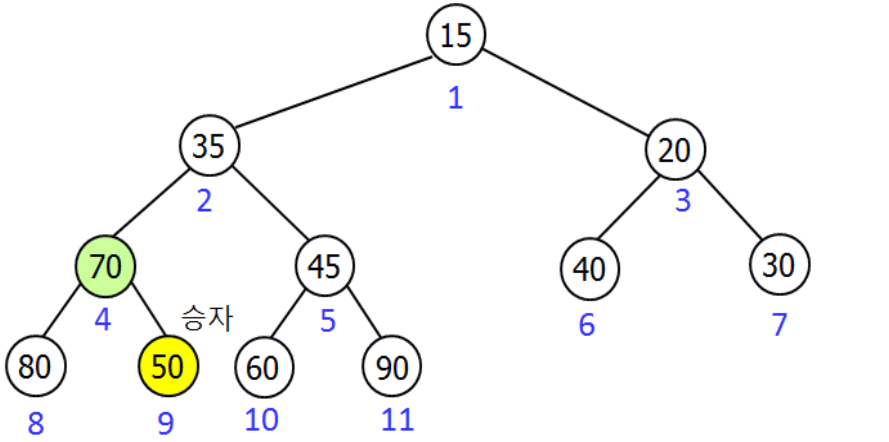
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	15	70	20	35	45	40	30	80	50	60	90	-	-	-	-

(d) 35와 45 중에 35가 승자



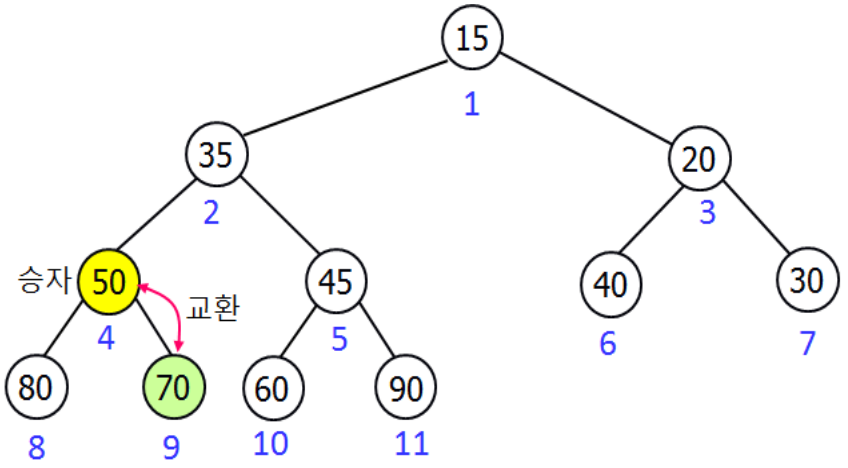
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	15	35	20	70	45	40	30	80	50	60	90	-	-	-	-

(e) 승자인 35와 70를 교환



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	15	35	20	70	45	40	30	80	50	60	90	-	-	-	-

(f) 80과 50 중에 50이 승자

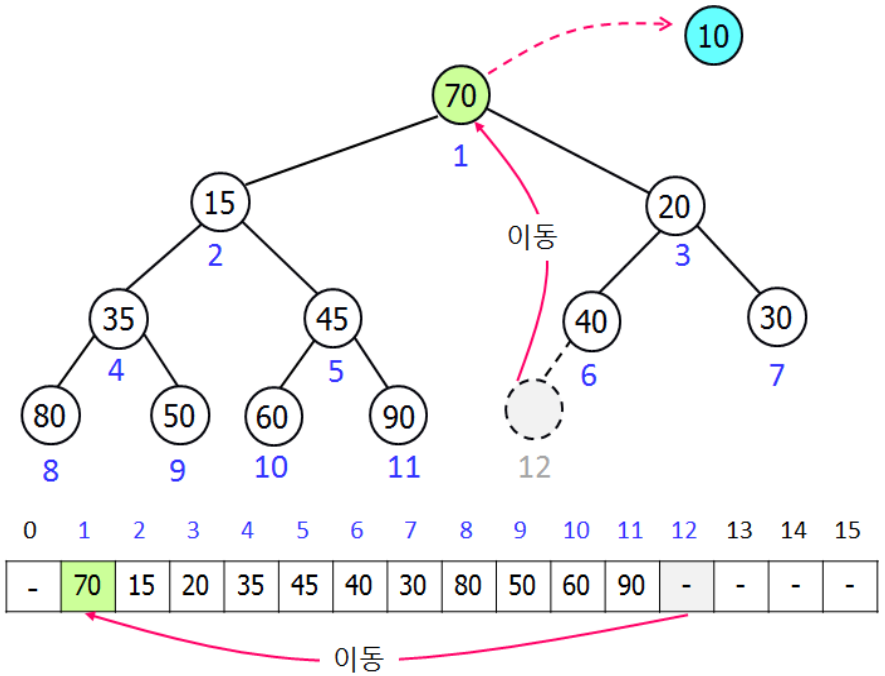


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	15	35	20	50	45	40	30	80	70	60	90	-	-	-	-

(g) 승자인 50과 70을 교환

deleteMin() 메소드

```
01   public Entry deleteMin() {
02       Entry min = a[1];
03       swap(1, N--);
04       a[N+1] = null;
05       downheap(1);
06       return min;
07   }


01  //  최솟값 삭제
02  //  a[1]의 최솟값을 min으로 저장하여 리턴
03  //  힙의 마지막 항목과 교환하고 힙 크기 1 감소
04  //  마지막 항목을 null로 처리
05  //  힙속성을 회복시키기 위해
```

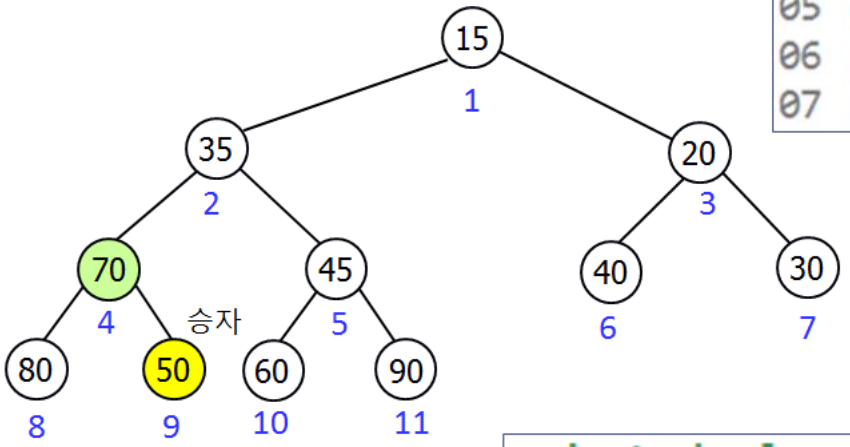


- Line 02: a[1]을 min에 저장한 뒤, 최종적으로 line 06에서 min을 리턴
- Line 03: 힙의 마지막 항목 a[N]과 a[1]을 교환한 뒤에 N을 1 감소
- Line 04: 가비지 컬렉션을 위해 삭제된 객체를 참조하던 배열 원소를 null로 만든다.
- Line 05: downheap(1)을 호출하여 힙속성을 회복시킴

downheap() 메소드

```
01 private void downheap(int i) {
02     while (2*i <= N) {
03         int k = 2*i;
04         if (k < N && greater(k, k+1)) k++;
05         if (!greater(i, k)) break;
06         swap(i, k);
07         i = k;
08     }
09 }
```

```
01 // i는 현재 노드의 인덱스
02 // i의 왼쪽 자식노드가 힙에 있으면
03 // k는 왼쪽 자식노드의 인덱스
04 // 왼쪽과 오른쪽 자식의 승자를 결정하여 k가 승자의 인덱스가됨
05 // 현재 노드가 자식 승자보다 작으면, 루프를 중단하고
06 // 현재 노드가 자식 승자보다 크면 현재 노드와 자식 승자와 교환
07 // 자식 승자가 현재 노드가 되어 다시 반복하기 위해
```




```
private boolean greater(int i, int j) { // 키 비교
    return a[j].getKey().compareTo(a[i].getKey()) < 0;  }
```

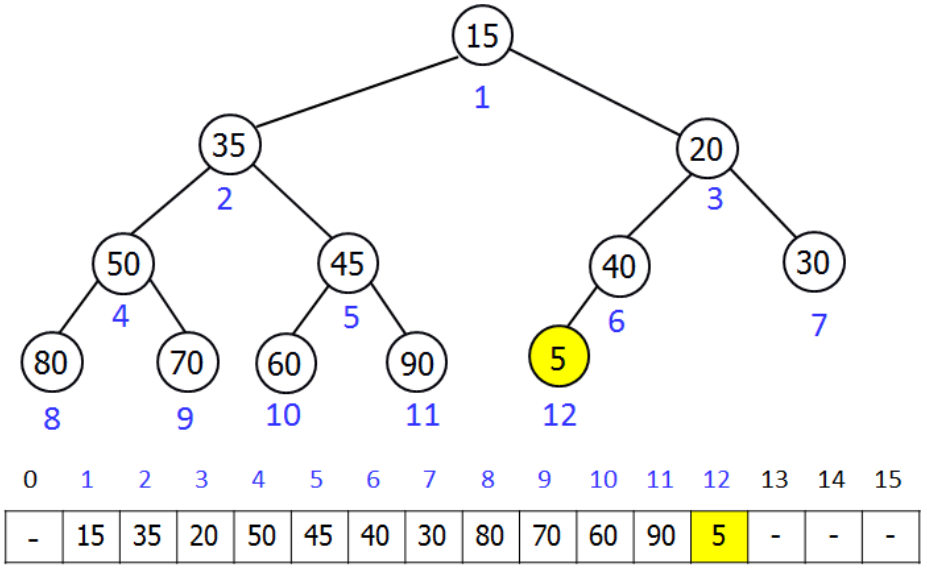
삽입 연산(insert)

[1] 힙의 마지막 노드(즉, 배열의 마지막 원소)의 바로 다음 빈 원소에 새로운 항목을 저장

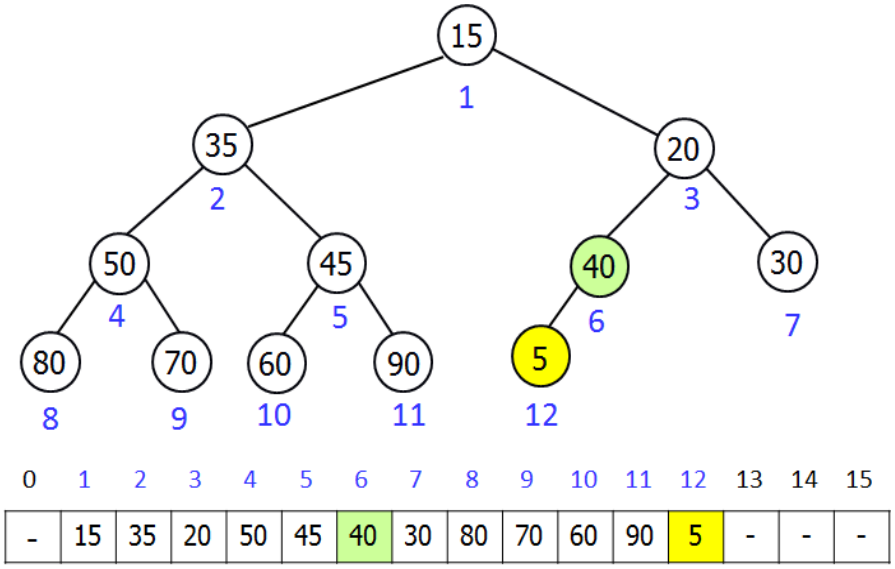
[2] 루트 방향으로 올라가면서 부모노드의 키값과 비교하여 힙속성이 만족될 때까지 노드를 교환

- [2]의 과정은 이파리노드로부터 위로 올라가며 진행되므로 upheap

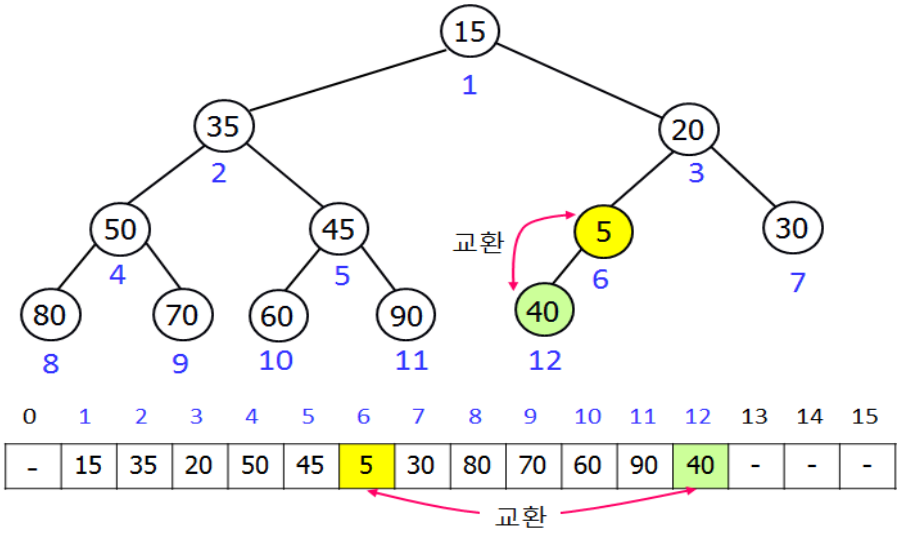
최소힙에 5를 삽입하는 과정



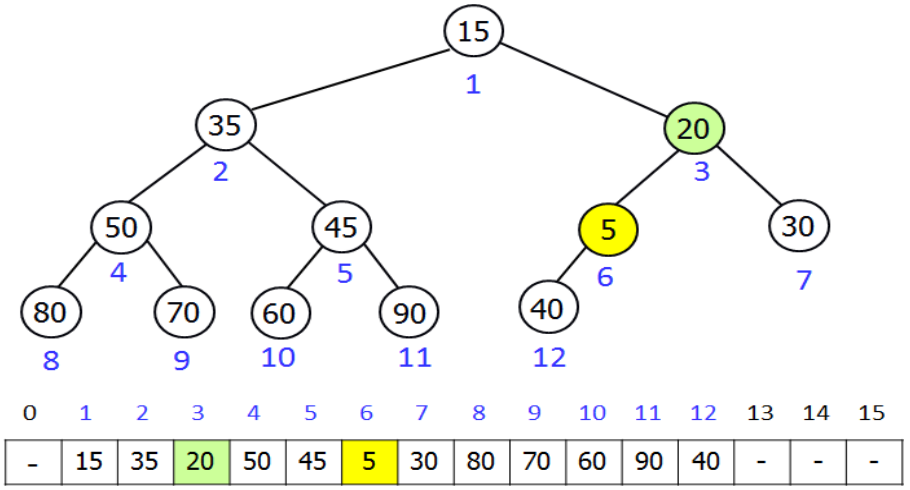
(a) 5를 배열의 마지막 원소(90) 다음에 저장



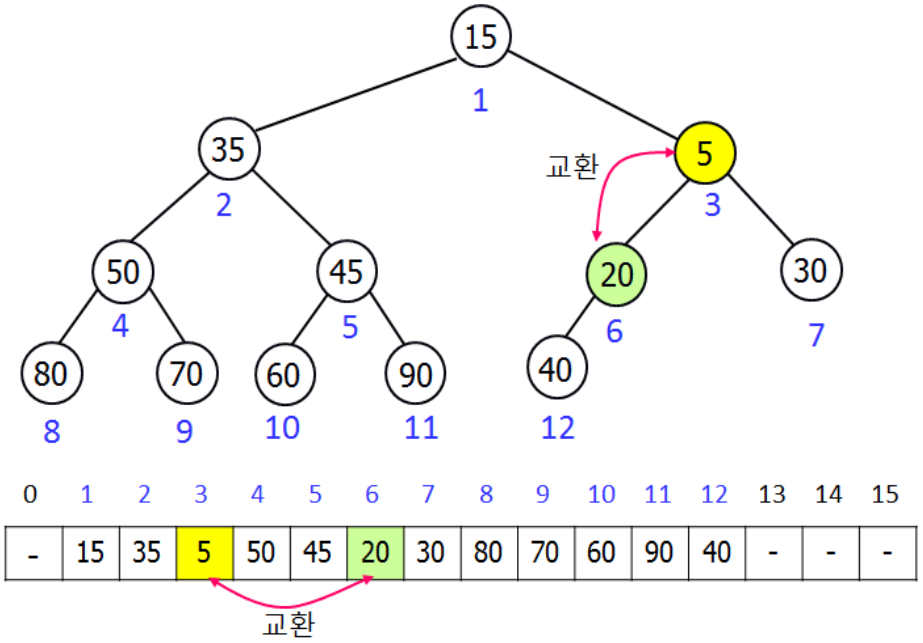
(b) a[12]의 5와 부모노드 a[6]의 40 비교



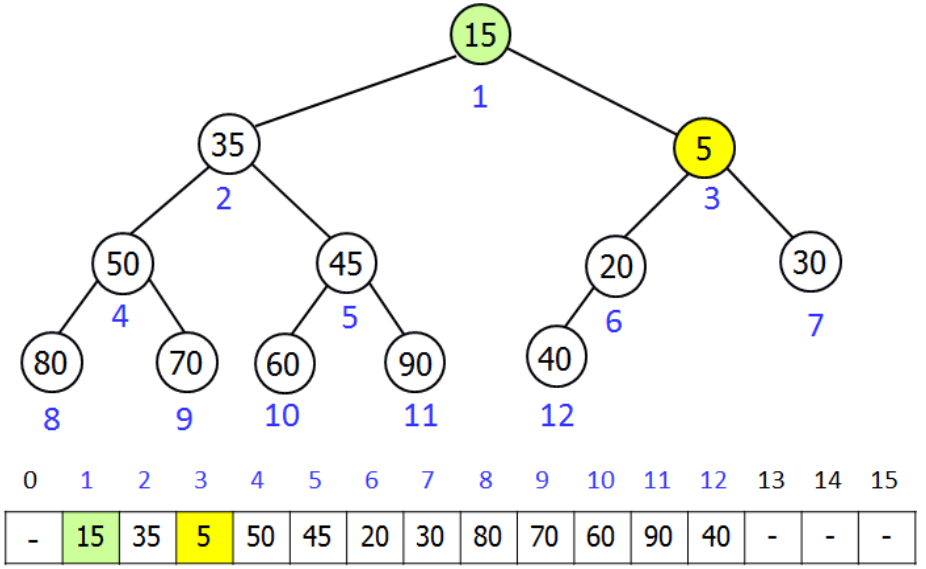
(c) 5와 40을 교환



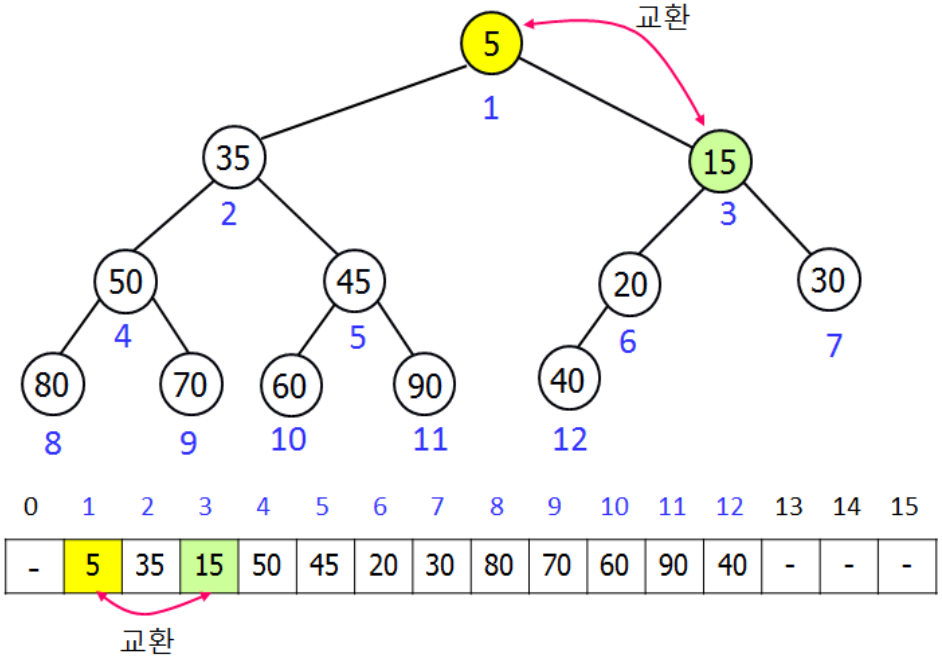
(d) a[6]의 5와 부모노드 a[3]의 20 비교



(e) 5와 20을 교환



(f) a[3]의 5와 부모노드 a[1]의 15 비교



(g) 5와 15를 교환

insert() 메소드

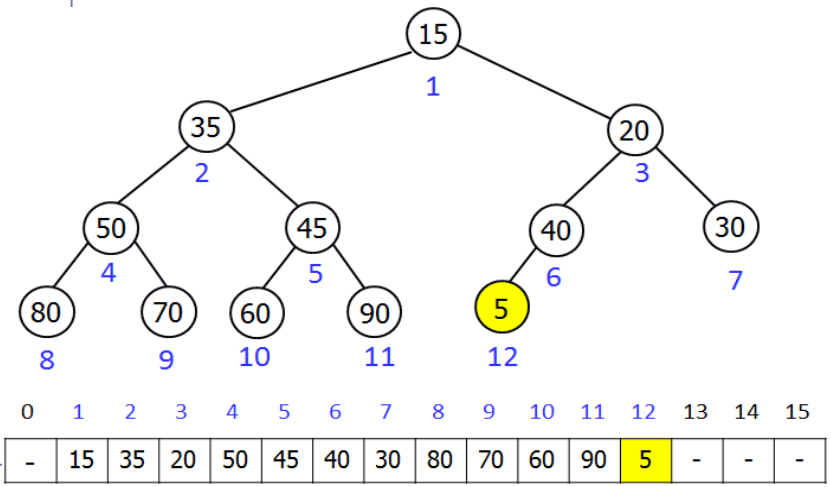
```
01    public void insert(Key newKey, Value newValue) {
02        Entry temp = new Entry(newKey, newValue);
03        a[++N] = temp;
04        upheap(N);
05    }

01 // 새로운 항목 삽입
02 // Entry 생성
03 // 새로운 키를 배열 마지막 원소 다음에 저장
04 // 위로 올라가며 힙속성 회복시키기 위해
```

- Line 02: Entry 객체를 생성
- Line 03: N을 1증가시켜, 즉, 힙 크기를 1 증가시켜, 힙의 마지막 노드 다음의 빈 원소에 새로 생성한 Entry 객체를 저장
- Line 04의 upheap() 메소드를 호출하여 루트노드 방향으로 거슬러 올라가며 힙속성이 어긋나는 경우 부모와 자식을 교환
- [참고] 실제로 Entry 객체를 저장하는 것이 아니라, Entry 객체의 레퍼런스를 배열 원소에 저장

upheap() 메소드

```
01 private void upheap(int j) {
02     while (j > 1 && greater(j/2, j)) {
03         swap(j/2, j);
04         j = j/2;
05     }
06 }
```

```
01 // j는 현재 노드의 인덱스
02 // 현재 노드가 루트가 아니고 동시에 부모가 크면
03 // 부모와 현재 노드 교환
04 // 부모가 현재 노드가 되어 다시 반복하기 위해
```



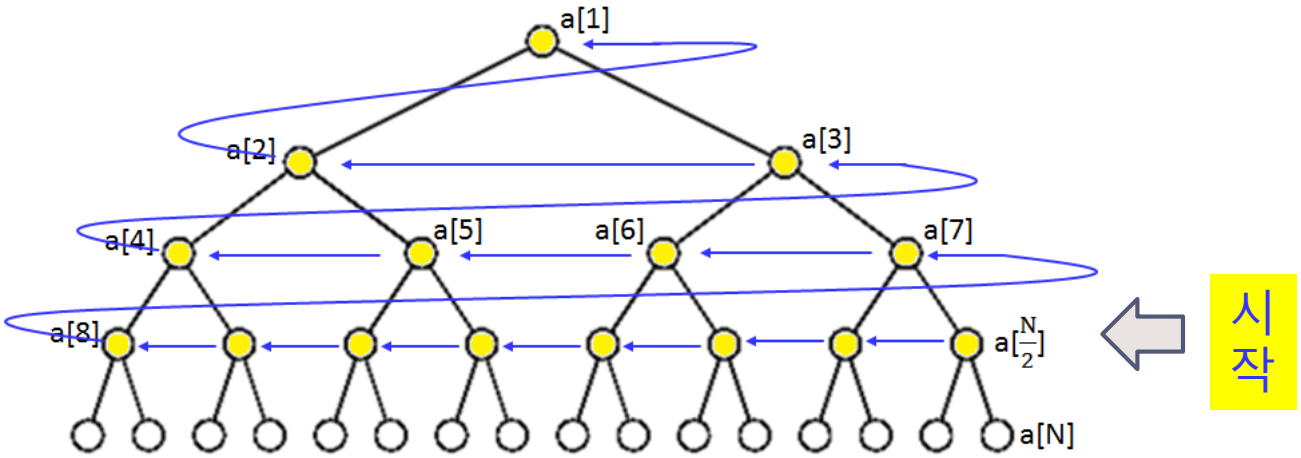
- Line 02: while-루프 조건에서 현재 노드의 인덱스인 j를 2로 나누어 부모노드를 찾고, 부모노드가 현재 노드보다 크면
- Line 03: 부모와 현재 노드를 교환
- Line 04: 현재 노드의 인덱스를 부모노드 인덱스인 j/2로 만들어 계속해서 루트노드 방향으로 올라가며 while-루프를 수행

상향식 힙만들기(Bottom-up Heap Construction)

[핵심 아이디어]

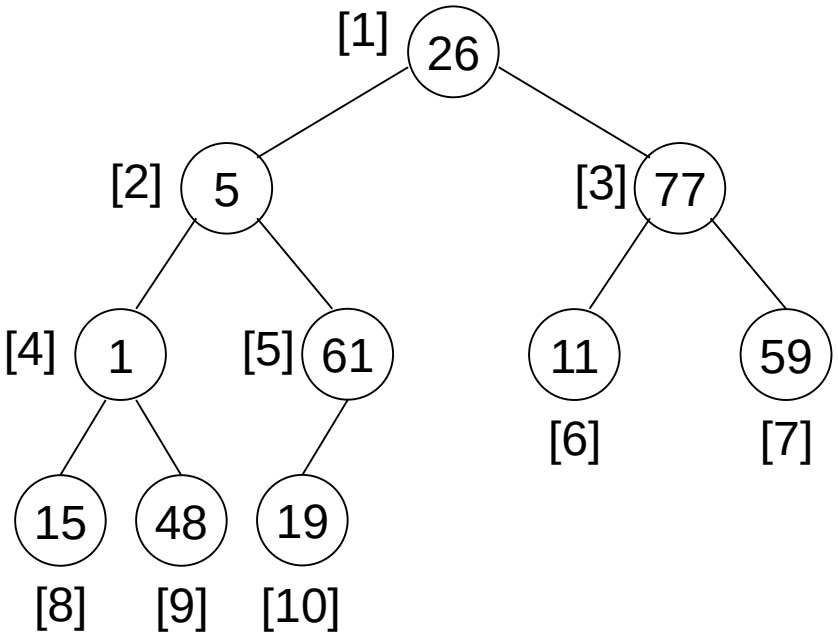
- **상향식(Bottom-up) 방식으로 각 노드에 대해 힙속성을 만족하도록 부모와 자식노드를 서로 교환**
- **N개의 항목이 배열에 임의의 순서로 저장되어 있을 때, 힙을 만들기 위해선 a[N/2]부터 a[1]까지 차례로 downheap을 각각 수행하여 힙속성을 충족시킨다.**

- a[N/2+1]~a[N]에 대하여 downheap을 수행하지 않는 이유:
  - 이 노드들 각각은 이파리노드이므로, 각 노드 스스로가 힙의 크기가 1인 최소힙이기 때문



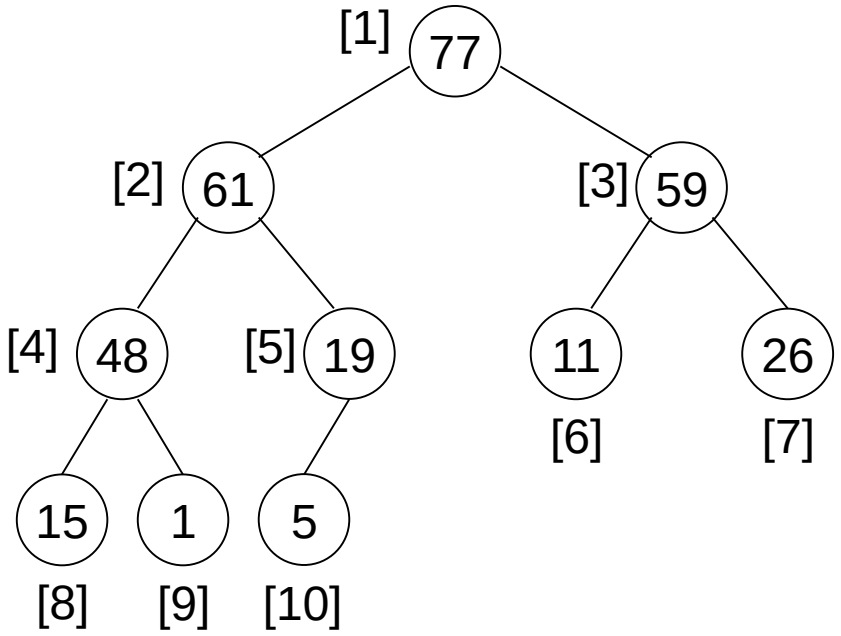
createHeap() 메소드 - MaxHeap의 생성 예

- 입력 리스트 : (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)



26	5	77	1	61	11	59	15	48	19

(a) 입력 리스트를 이진 트리로 변환한 모습



77	61	59	48	19	11	26	15	1	5

(b) 초기 히프 형태로 구성한 것

createHeap() 메소드

- **초기에 임의의 순서로 키가 저장되어 있는 배열 a[1]~a[N]의 항목들을 최소힙으로 만드는 메소드**
  - Line 02: for-루프는 i가 N/2부터 시작하는데,
    이는 a[N/2]부터 downheap을 수행해 나가,
    끝으로 a[1]을 downheap하여 최소힙을 만든다.

```
01  public void createHeap() {      // 초기 힙 만들기
02      for(int i = N/2; i > 0; i--){
03          downheap(i);
04      }
05  }
```

```java
public class main {
    public static void main(String[] args)  {
        Entry[] a = new Entry[16];      // a[0]은 사용 안함
        a[1]  = new Entry(90, "watermelon");a[2]  = new Entry(80, "pear");
        a[3]  = new Entry(70, "melon");     a[4]  = new Entry(50, "lime");
        a[5]  = new Entry(60, "mango");     a[6]  = new Entry(20, "cherry");
        a[7]  = new Entry(30, "grape");     a[8]  = new Entry(35, "orange");
        a[9]  = new Entry(10, "apricot");   a[10] = new Entry(15, "banana");
        a[11] = new Entry(45, "lemon");     a[12] = new Entry(40, "kiwi");
        BHeap h = new BHeap(a,12);    // 힙 객체 생성
        System.out.println("힙 만들기 전:"); h.print();
        h.createHeap();  // 힙 만들기
        System.out.println("최소힙:"); h.print();
        System.out.println("min 삭제 후"); System.out.println(h.deleteMin().getValue());
        h.print();
        h.insert(5,"apple"); System.out.println("5 삽입 후"); h.print();
    }
}
```

Console

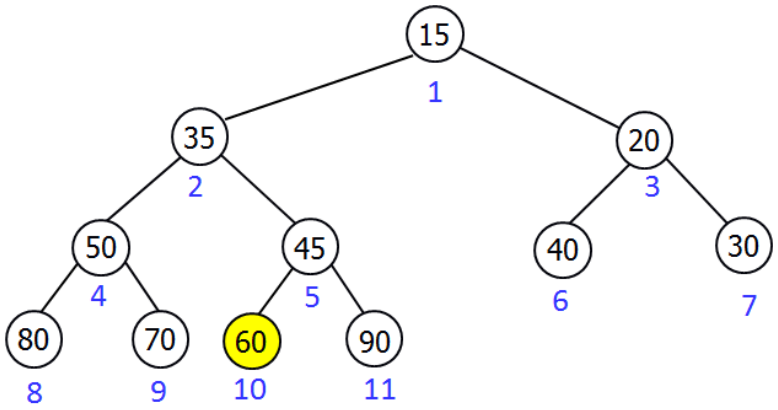
<terminated> main (38) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe

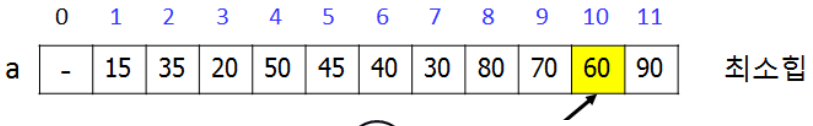
```
힙 만들기 전:
[90 watermelon] [80 pear] [70 melon] [50 lime] [60 mango] [20 cherry] [30 grape] [35 orange] [10 apricot] [15 banana] [45 lemon] [40 kiwi]
힙 크기 = 12

최소힙:
[10 apricot] [15 banana] [20 cherry] [35 orange] [45 lemon] [40 kiwi] [30 grape] [80 pear] [50 lime] [60 mango] [90 watermelon] [70 melon]
힙 크기 = 12

min 삭제 후
apricot
[15 banana] [35 orange] [20 cherry] [50 lime] [45 lemon] [40 kiwi] [30 grape] [80 pear] [70 melon] [60 mango] [90 watermelon]
힙 크기 = 11

5 삽입 후
[5 apple] [35 orange] [15 banana] [50 lime] [45 lemon] [20 cherry] [30 grape] [80 pear] [70 melon] [60 mango] [90 watermelon] [40 kiwi]
힙 크기 = 12
```

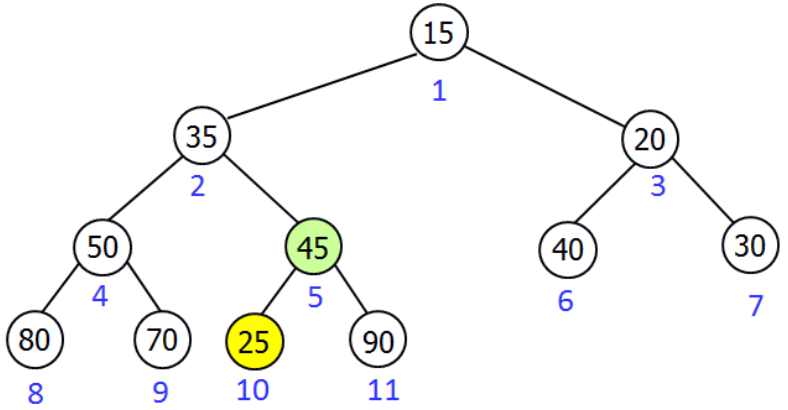
이진힙의 추가 연산을 위한 자료구조

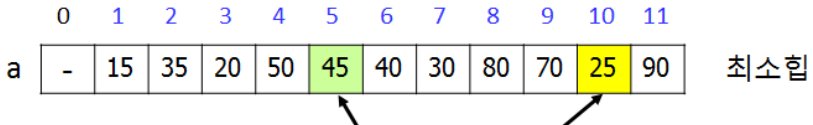
- **임의의 노드를 삭제하는 delete 연산**
- **임의의 노드의 키 값을 감소시키는 decrease_key 연산**
  - 키값을 감소시킨 후, upheap을 수행하면서 힙속성이 어긋나는 경우 부모 자식노드의 교환을 통해 힙속성을 복원
- **이 연산들을 수행하려면 힙에서의 각 키의 위치를 알아야**
  - 이를 위해 두 개의 1차원 배열을 이용하여, 힙에 대해 연산이 수행될 때마다 노드의 위치 변화를 갱신
  - 교재에서는 position 배열을 쓰는 방법을 소개

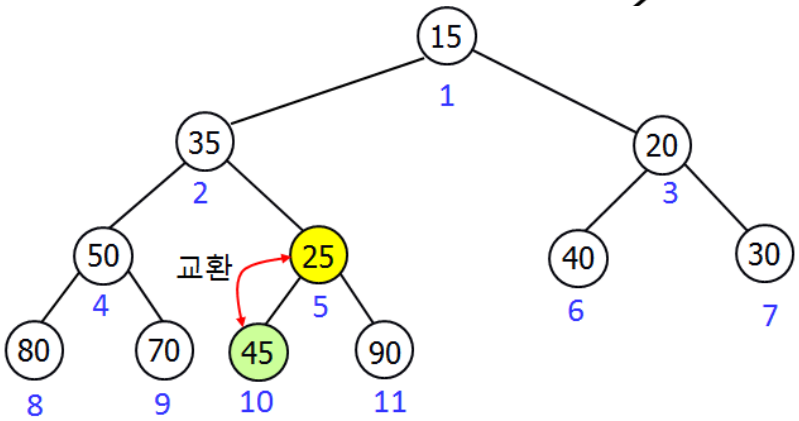
60을 35만큼 감소

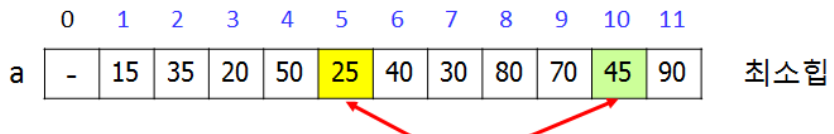

15
1
35
2
20
3
50
4
45
5
40
6
30
7
80
8
70
9
60
10
90
11

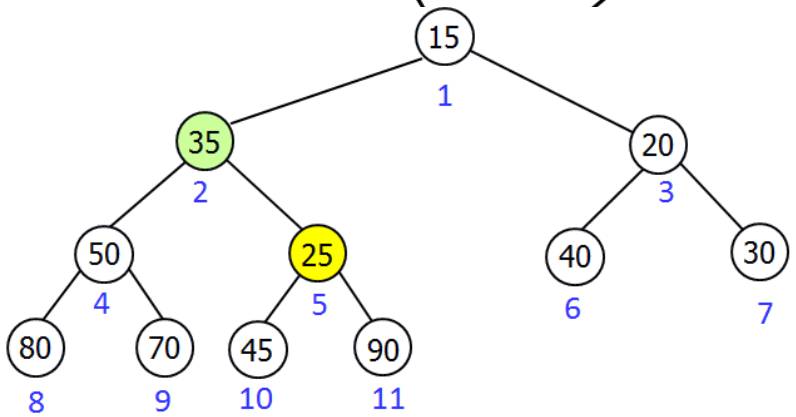

0 1 2 3 4 5 6 7 8 9 10 11
a - 15 35 20 50 45 40 30 80 70 60 90
최소힙

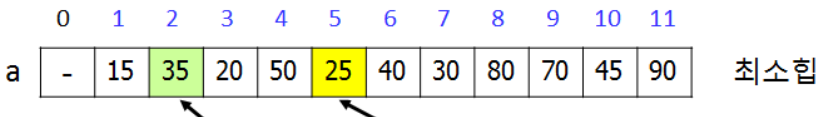

15
1
35
2
20
3
50
4
45
5
40
6
30
7
80
8
70
9
25
10
90
11

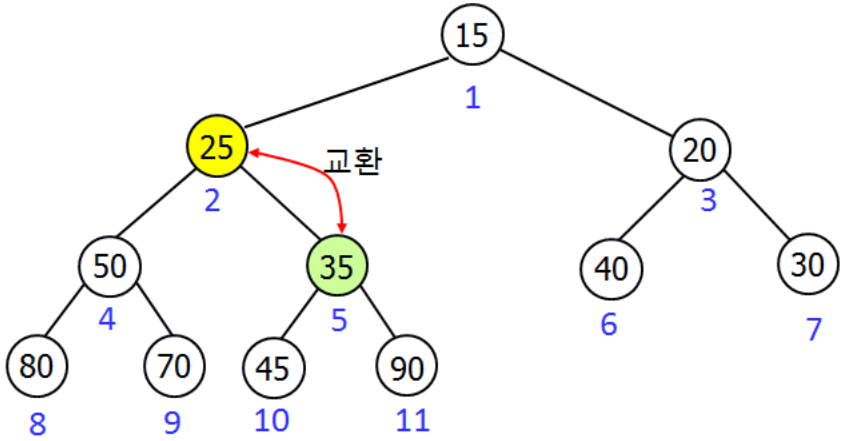

0 1 2 3 4 5 6 7 8 9 10 11
a - 15 35 20 50 45 40 30 80 70 25 90
최소힙

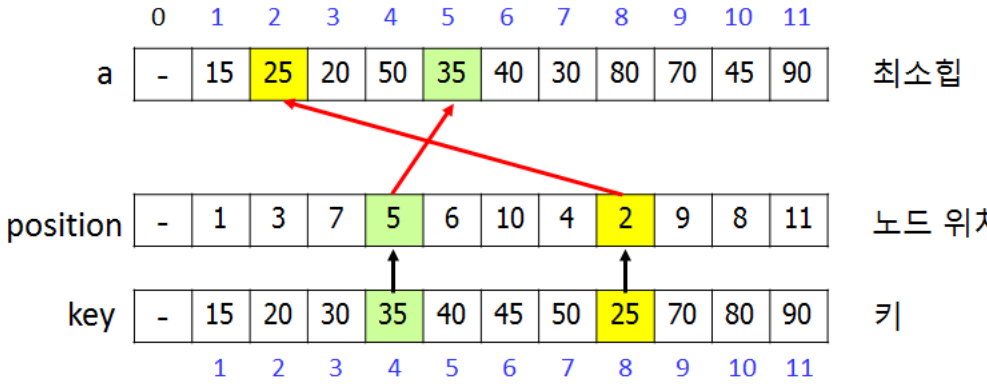

15
1
35
2
20
3
50
4
교환
25
5
40
6
30
7
80
8
70
9
45
10
90
11


0 1 2 3 4 5 6 7 8 9 10 11
a - 15 35 20 50 25 40 30 80 70 45 90
최소힙


15
1
35
2
20
3
50
4
25
5
40
6
30
7
80
8
70
9
45
10
90
11


0 1 2 3 4 5 6 7 8 9 10 11
a - 15 35 20 50 25 40 30 80 70 45 90
최소힙





(e) 25와 부모노드 35의 교환

수행시간

- Insert, decrease_key, delete 연산을 위한 upheap은
  삽입된 노드나 키값이 감소된 노드로부터 최대 루트노드까지 올라가며
  부모와 자식노드를 교환
- delete_min 연산에서는 힙의 마지막 노드를 루트노드로 이동한 후,
  downheap을 최하위 층의 노드까지 교환해야 하는 경우가 발생
- 힙에서 각 연산의 수행시간은 힙의 높이에 비례
- 힙은 완전이진트리이므로 힙에 N개의 노드가 있으면 그 높이는
  $\lceil \log(N+1) \rceil$
- 각 힙 연산의 수행시간은 O(logN)

상향식 힙만들기의 수행시간 분석

- **노드 수가 N인 힙의 각 층에 있는 노드 수를 살펴보자. 단, 간단한 계산을 위하여 $N = 2^k-1$로 가정, k는 양의 상수**
  - 최하위층 (h = 0)의 노드 수 = $\frac{N}{2}$
  - (h=1)의 노드 수 = $\frac{N}{2^2}$
  - (h=2)의 노드 수 = $\frac{N}{2^3}$
  - h층의 노드 수 = $\frac{N}{2^{h+1}}$
- **힙만들기는 h=1인 경우부터 시작하여 최상위층의 루트노드까지 각 노드에 대해 downheap을 수행**

$$T(N) = 1\cdot\frac{N}{2^2} + 2\cdot\frac{N}{2^3} + 3\cdot\frac{N}{2^4} + \cdots + (\log N-1)\cdot\frac{N}{2^{\log N}}$$

$$\leq \sum_{h=1}^{\log N} h\cdot\frac{N}{2^{h+1}} = \frac{N}{2}\sum_{h=1}^{\log N}\frac{h}{2^h} \leq \frac{N}{2}\cdot 2, \quad \sum_{x=0}^{\infty}\frac{x}{2^x} = 2 \text{ 이므로}$$

$$= O(N)$$

응용

- 관공서, 은행, 병원, 우체국, 대형 마켓, 공항 등에서 이루어지는 업무와 관련된 이벤트 처리
- 컴퓨터 운영체제의 프로세스 처리
- 네트워크 라우터에서의 패킷 처리 등
- 실시간 급상승 검색어(데이터 스트림에서 Top k 항목 유지) 제공
- 7.2 절의 허프만 코딩
- 8.6 절의 힙정렬
- 9 장의 Prim의 최소신장트리 알고리즘과 Dijkstra의 최단경로 알고리즘에도 활용

7.2 허프만 코딩

- 대표적인 비손실 압축 방식
- 입력 파일의 문자 빈도 수로 최소힙을 이용하여 허프만 코드를 만들어 파일을 압축하고 나중에 복원하는 알고리즘
- [응용] 허프만 코드는 Unix의 파일압축 명령어인 compress에 사용되고, JPEG 이미지 파일과 MP3 음악 파일을 압축하기 위한 서브루틴으로도 활용

> [핵심 아이디어]
> 빈도 수가 높은 문자에는 짧은 코드를 부여하고,
> 빈도 수가 낮은 문자에는 긴 코드를 부여하여 압축 효율을 높인다.

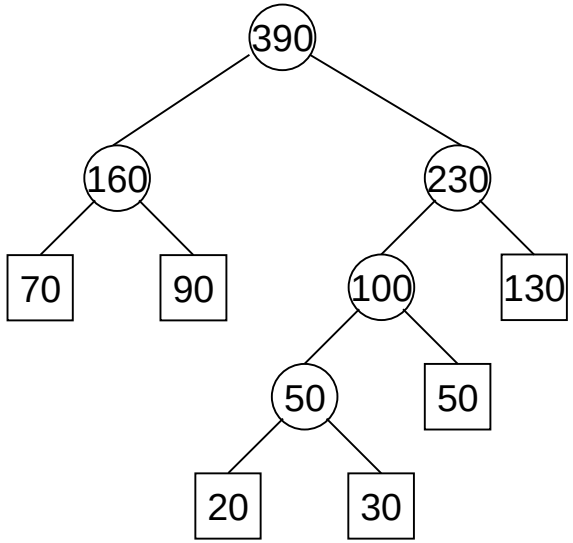
7.2 허프만 코딩

- 예) 빈도 : q_1=20, q_2=30, q_3=50, q_4=70, q_5=90, q_6=130
  - 고정길이 표현. 6개의 자료를 표현하기 위해 3bit를 사용하는 방법

    M_1=000, M_2=001, M_3=010, M_4=011, M_5=100, M_6=101
    - 이 경우 전체 자료를 표현하는데 필요한 bit수는 (20+30+50+70+90+130)*3 = 1080

  - 허프만 코드. 자주 쓰이는 자료에 짧은 길이의 코드를 부여하는 방법

    M_1=1000, M_2=1001, M_3=101, M_5=01, M_6=00, M_7=11
    - 이 경우 필요한 bit수는 20*4 + 30*4 + 50*3 + 70*2 + 90*2 + 130*2 = 930



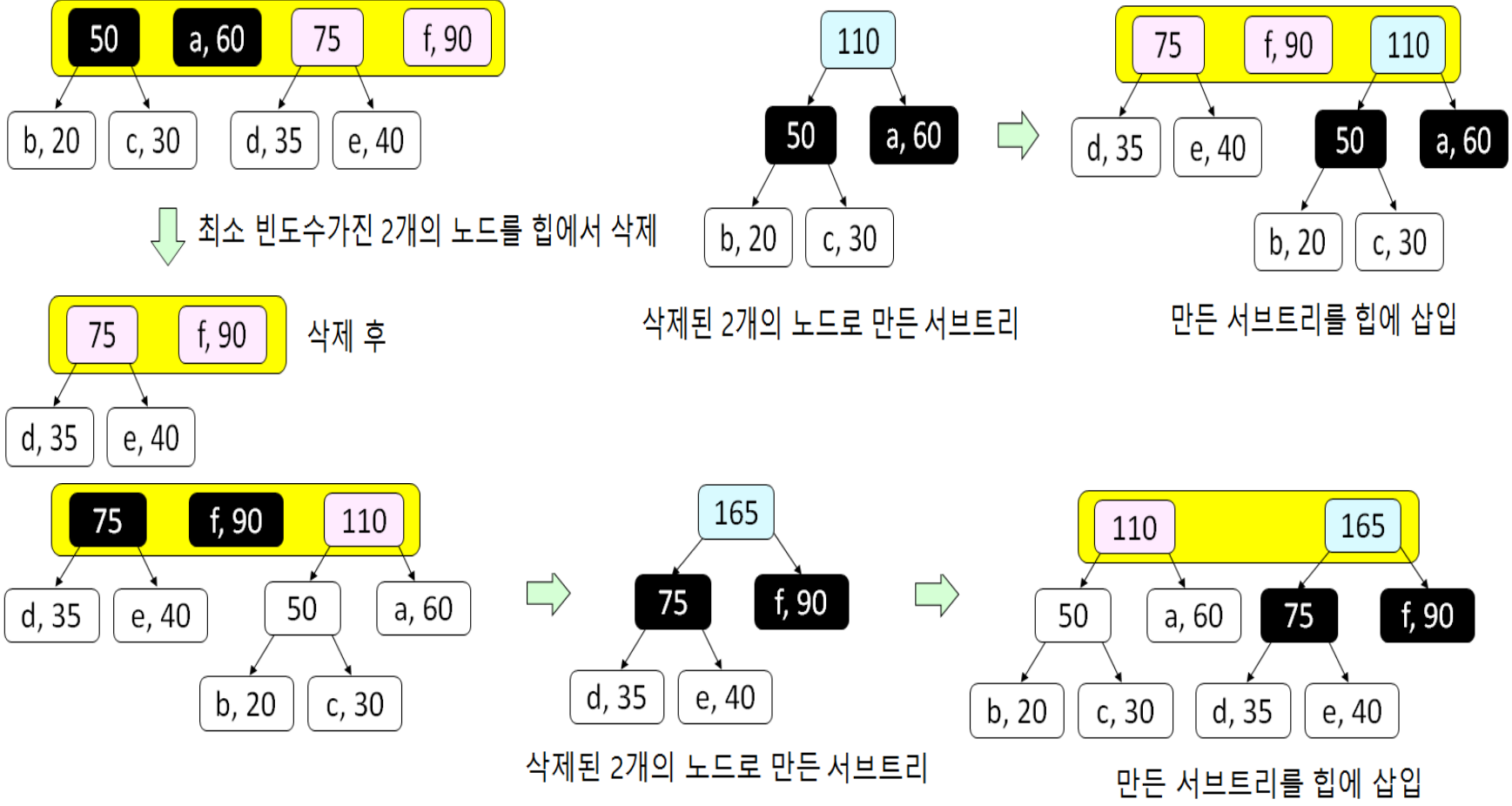
7.2 허프만 코딩

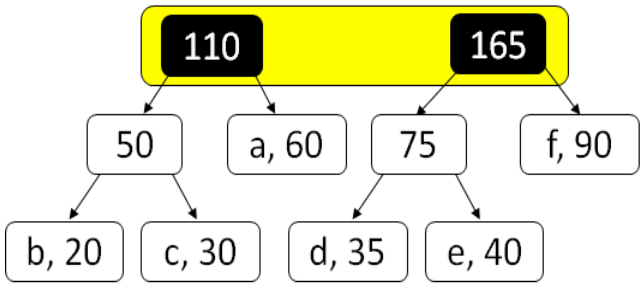
- **허프만 압축 알고리즘은 2단계로 수행**
  - [1] 입력 파일을 스캔하여 각 문자의 빈도 수를 계산하고, 이 빈도 수로 허프만 트리를 생성한다. 끝으로 생성된 트리로부터 각 문자에 대응하는 허프만 코드를 추출
  - [2] 파일을 스캔하며 각 문자를 허프만 코드로 변환

```
== 허프만 트리 생성 ===
[1] 입력 파일을 스캔하여 각 문자의 빈도 수 계산
[2] 빈도 수를 우선순위로 최소힙 h를 구성
[3] while (힙의 크기 > 1)
        e1 = h.delete_min();
        e2 = h.delete_min();
        t    = new 항목(e1의 빈도 수+e2의 빈도 수,
                                left = e1, right =e2);
        h.insert(t);  // 힙에 새로 만든 항목 삽입
[4] return h.delete_min();
```

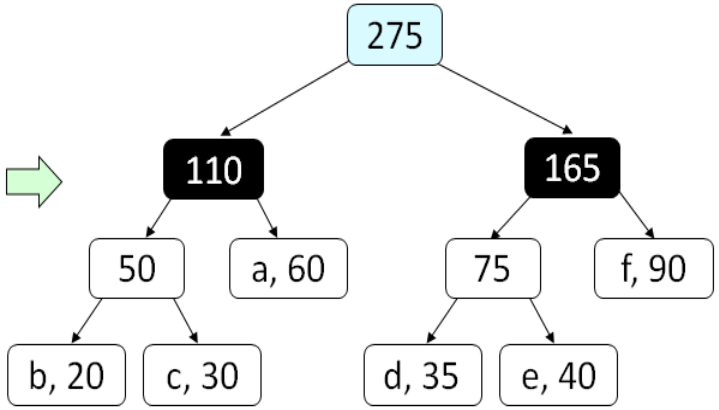
- 허프만 트리 만들기는
  최소의 빈도 수를 가진 2 개의 노드를 합하여
  새로운 노드를 만들어 힙에 삽입하는 과정을
  반복하며, 힙에 1 개의 노드만 남으면
  이 노드를 리턴
- 이 리턴된 노드가 허프만 트리의 루트노드

[예제] 입력 파일이 6개의 문자, a, b, c, d, e, f로 구성되어 있고, 문자의 빈도 수가 각각 60, 20, 30, 35, 40, 90


50
a, 60
75
f, 90
b, 20
c, 30
d, 35
e, 40
최소 빈도수가진 2개의 노드를 힙에서 삭제
110
50
a, 60
b, 20
c, 30
삭제된 2개의 노드로 만든 서브트리
75
f, 90
110
d, 35
e, 40
50
a, 60
b, 20
c, 30
만든 서브트리를 힙에 삽입
75
f, 90
삭제 후
d, 35
e, 40
75
f, 90
110
d, 35
e, 40
50
a, 60
b, 20
c, 30
165
75
f, 90
d, 35
e, 40
삭제된 2개의 노드로 만든 서브트리
110
165
50
a, 60
75
f, 90
b, 20
c, 30
d, 35
e, 40
만든 서브트리를 힙에 삽입

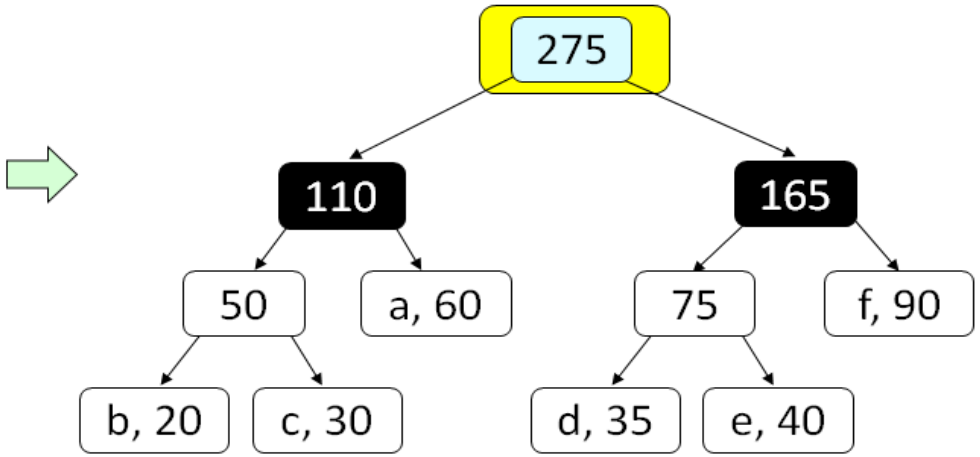


삭제 후



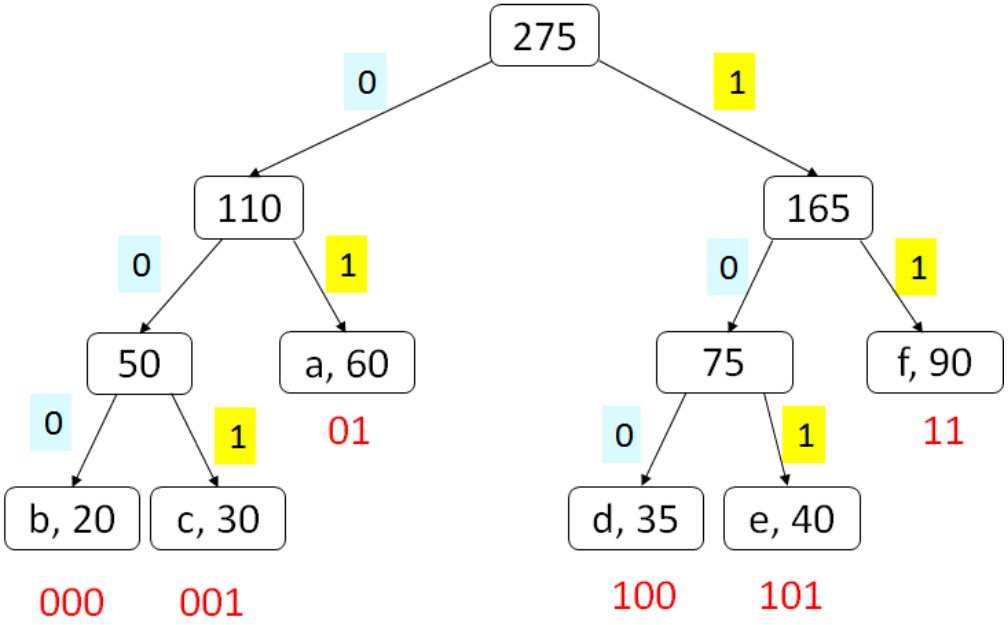
최소 빈도수가진 2개의 노드 힙에서 삭제

삭제된 2개의 노드로 만든 서브트리



만든 서브트리를 힙에 삽입

[허프만 코드] 루트노드로부터 각 이파리노드로 내려가며 **왼쪽으로 내려갈 때엔 0, 오른쪽으로 내려갈 때엔 1**을 추가하여 이파리노드에 있는 문자의 허프만 코드를 계산



허프만 코드

a: 01 b: 000 c: 001

d: 100 e: 101 f: 11

허프만 코드의 속성 : 접두어 속성(Prefix Property)

- **어떤 문자의 코드도 다른 문자의 코드의 접두어(Prefix)가 되지 않게**
  - Prefix Property가 지켜지지 안는 예 : a의 코드 = 00101, b의 코드= 001
  - 허프만 트리에서 루트노드로부터 a를 가진 이파리노드까지 내려가는 경로의 앞부분이 루트로부터 b를 가진 노드로 가는 경로와 일치함
  - 허프만 트리에서는 내부노드가 문자를 가질 수 없으므로 b가 001이라는 코드를 가질 수 없음
  - 접두어 속성 덕분에 문자를 코드로 바꾸는 과정에서 코드와 코드 사이를 분리시키기 위해 특별한 문자를 삽입할 필요가 없음
  - 허프만 코드를 이용해서 abcd를 변환(압축)시키면 010000011100이 되는데, 01#000#001#100과 같은 방식으로 연속된 2 개의 코드 사이에 특수 문자를 넣어 구분할 필요 없음

허프만 코드		
a: 01	b: 000	c: 001
d: 100	e: 101	f: 11

```
01 public class Entry {
02     private int    frequency;  // 빈도 수
03     private String word;       // 이파리 노드의 문자 또는 내부노드의 합성된 문자열
04     private Entry  left;       // 왼쪽 자식
05     private Entry  right;      // 오른쪽 자식
06     private String code;       // 허프만 코드
07     public Entry (int newFreq, String newValue, Entry l, Entry r, String s){
08         frequency = newFreq;
09         word  = newValue;
10         left  = l;
11         right = r;
12         code  = s;
13     }
14     public int    getKey()   { return frequency; }
15     public String getValue() { return word; }
16     public String getCode()  { return code; }
17     public Entry  getLeft()  { return left; }
18     public Entry  getRight() { return right; }
19     public void   setCode(String newCode) { code = newCode; }
20 }
```

- **Entry 클래스는 허프만 트리에 쓰일 노드 객체를 생성**
  - Line 02 ~ 06:
    - 빈도 수를 저장하는 frequency, 문자 또는 합쳐진 스트링을 저장하는 word, 노드의 왼쪽과 오른쪽 자식 레퍼런스인 left와 right, 허프만 코드를 저장할 code
  - Line 07 ~ 13: 객체 생성자
  - Line 14 ~ 19: get, set 메소드들

Huffman 클래스

```
01 public class Huffman {
02     private Entry[] a;  // a[0]은 사용 안함
03     private int N;      // 힙의 크기
04     public Huffman(Entry[] harray, int initialSize) { // 생성자
05         a = harray;
06         N = initialSize;
07     }
08     private boolean greater(int i, int j) {
09         return a[i].getKey()> a[j].getKey(); }
      // size(), swap(), createheap(), insert(), deleteMin(),
      // upheap(), downheap() 메소드들은 BHeap클래스의 메소드들과 동일하다.
   }
```

- **Huffman 클래스는 7.1절의 BHeap 클래스와 거의 동일**
  - createTree()가 추가되고, line 08~09에서 2 개의 빈도 수를 비교하는 greater() 메소드를 int형 단순 비교로 수정

```
01  public Entry createTree(){
02      while (size() > 1){
03          Entry e1 = deleteMin();
04          Entry e2 = deleteMin();
05          Entry temp = new Entry(e1.getKey()+e2.getKey(),
06                                 e1.getValue()+e2.getValue(),
07                                 e1, e2, " ");
08          insert(temp);
09      }
10      return deleteMin();
11  }

02  // 힙에 1개의 노드만 남을 때까지
03  // 힙에서 최소 빈도수 가진 노드 제거하여 e1이 참조
04  // 힙에서 최소 빈도수 가진 노드 제거하여 e2가 참조
05  // e1과 e2의 빈도수를 합산
06  // string 이어붙이기
07  // e1,e2가 각각 새노드의 왼쪽,오른쪽 자식
08  // 새 노드를 힙에 삽입
09
10  // 1개 남은 노드(루트 노드)를 힙에서 제거하며 리턴
```

- **createTree() 메소드는 허프만 트리를 생성**
  - 힙에 1개의 노드가 남을 때까지 line 03~04에서 2번의 deleteMin()을 호출
  - Line 05~07: 빈도 수와 스트링을 각각 합하며 line 08에서 힙에 삽입
  - Line 10: 힙에 남은 1 개의 노드(허프만 트리의 루트노드)를 삭제하는 동시에 리턴하여 트리 생성 과정을 종료

```java
public class main {
    public static void main(String[] args)  {
        Entry[] a;      // a[0]은 사용 안함
        a = new Entry[7];

        a[1] = new Entry(60, "a", null, null, null); a[2] = new Entry(20, "b", null, null, null);
        a[3] = new Entry(30, "c", null, null, null); a[4] = new Entry(35, "d", null, null, null);
        a[5] = new Entry(40, "e", null, null, null); a[6] = new Entry(90, "f", null, null, null);

        Huffman h = new Huffman(a,6);
        System.out.println("최소힙 만들기 전");
        h.print();

        h.createHeap();
        System.out.println("최소힙:");
        h.print();

        System.out.println("허프만 코드");
        Entry root = h.createTree();
        h.preorder(root);
        System.out.println();
    }
}
```

Console

```
<terminated> main (39) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
최소힙 만들기 전
[60 a] [20 b] [30 c] [35 d] [40 e] [90 f]
최소힙:
[20 b] [35 d] [30 c] [60 a] [40 e] [90 f]
허프만 코드
b: 000  c: 001  a: 01  d: 100  e: 101  f: 11
```

입력을 압축하는 과정

- **각 문자에 대응되는 허프만 코드로 바꾸면 된다.**
  - 예를 들어, a c e f f e …를의 각 문자의 허프만 코드로 변환시키면

허프만 코드

a: 01 b: 000 c: 001

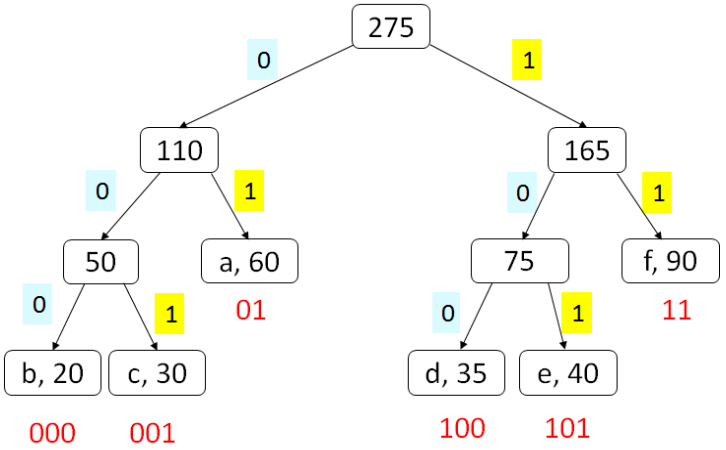
d: 100 e: 101 f: 11

  - 01, 001, 101, 11, 11, 101, …이 된다.
  - 이렇게 얻은 코드들을 붙여 쓴 0100110111111101…이 압축 결과

복원(Decoding) 알고리즘

- **긴 비트 스트링을 어떻게, 어디에서 끊어서 원래의 문자들로 복원할 수 있을까?**
  - [방법 1] 압축된 비트 스트링의 첫 번째 비트부터 읽어나가며 허프만 트리 상에서 루트로부터 0이면 왼쪽 자식노드로 1이면 오른쪽 자식노드로 내려가서 이파리노드에 도달하면 그 이파리노드가 가진 문자로 변환하고, 그 다음부터 동일한 방법으로 복원
    - [단점] 문자 1 개를 복원하는데 트리의 루트노드부터 이파리노드까지 내려가는 시간, 즉, 최대 허프만 트리의 높이에 비례하는 시간이 소요
  - 허프만 트리의 높이: 문자들의 빈도수 분포에 따라 최저 logN에서 최대 N-1

01000001100



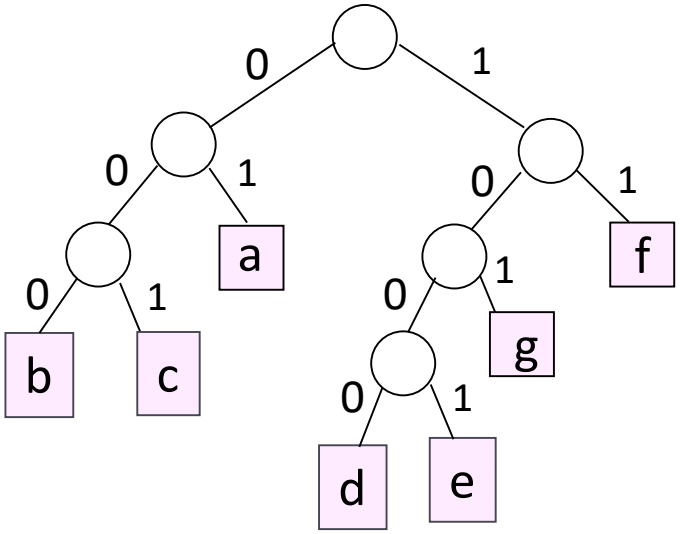
룩업테이블(Lookup Table) 방법

[1] 가장 긴 코드의 길이(비트 수) L을 찾는다.

[2] 각 문자 c_i의 테이블 주소의 길이를 L이 되도록 2^{L-l_i}개의 중복된 테이블 항목들을 다음과 같이 만든다.
단, 문자 c_i의 허프만 코드는 h_i이고, 그 길이는 l_i이다.

문자 c_i의 테이블 주소를 h_i뒤에 $L - l_i$ 비트의 모든 0과 1의 조합을 추가하여 중복된 항목들을 만든다. 그리고 각 테이블 항목에 문자 c_i와 허프만 코드 길이 l_i를 함께 저장한다.

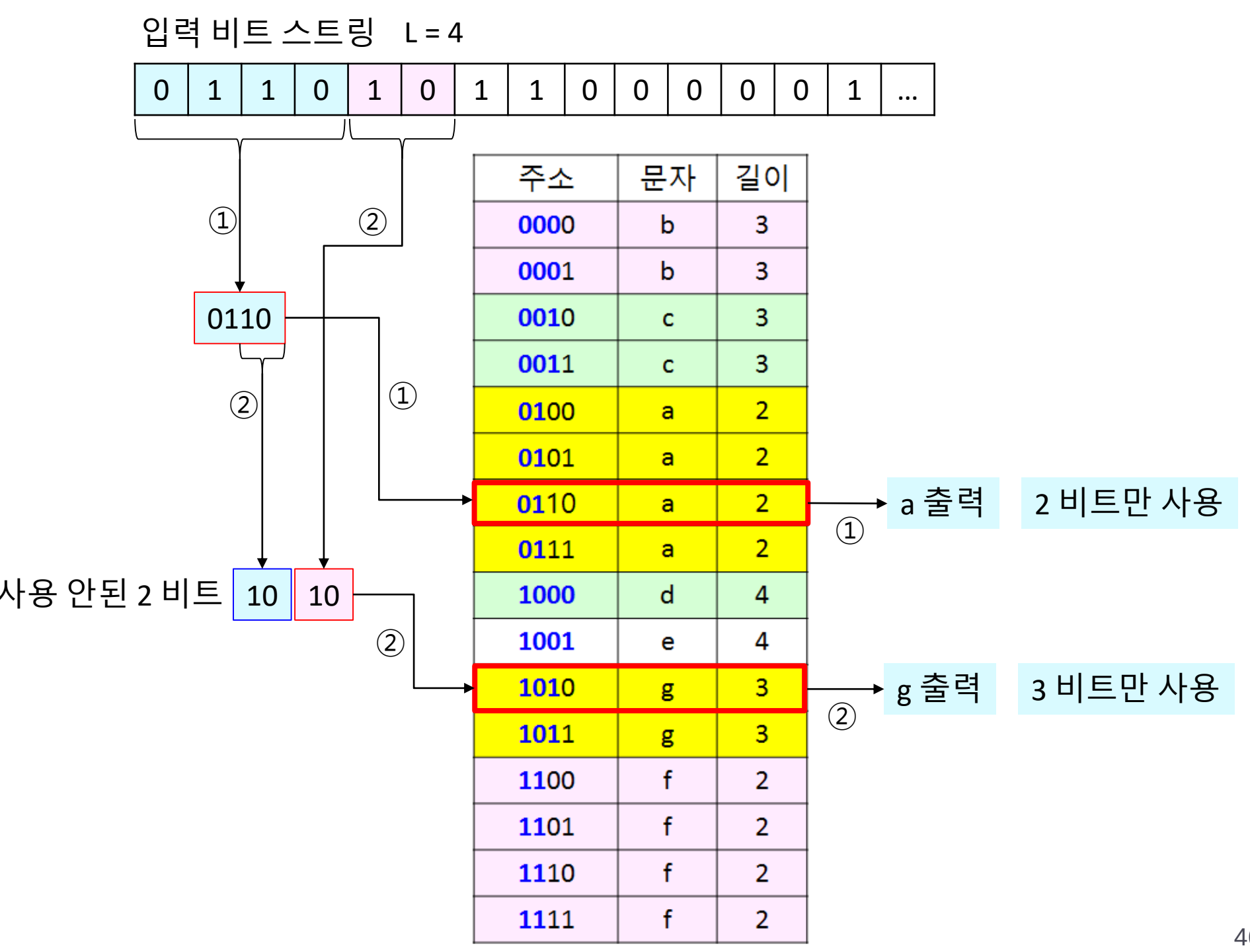
[예제]



주소	문자	길이
0000	b	3
0001	b	3
0010	c	3
0011	c	3
0100	a	2
0101	a	2
0110	a	2
0111	a	2
1000	d	4
1001	e	4
1010	g	3
1011	g	3
1100	f	2
1101	f	2
1110	f	2
1111	f	2

복원 방법

- 복원할 때에는 입력 비트 스트링에서 첫 L비트를 읽어와서 L 비트에 해당되는 주소의 테이블 항목에 있는 문자 c_i를 첫 복원한 문자로 출력
- 그리고 나머지 L-l_i 비트는 아직 복원을 위해 사용되지 않은 부분이므로 입력으로부터 그 다음 l_i 비트만큼을 읽어와 L 비트를 만들어 이에 대응되는 테이블 항목을 찾음
- 이러한 방식으로 반복하여 전체 입력 비트 스트링을 압축 이전의 상태로 복원


입력 비트 스트링 L = 4
0 1 1 0 1 0 1 1 0 0 0 0 0 1 ...
①
②
0110
②
①
사용 안된 2 비트
10
10
②
주소 문자 길이
0000 b 3
0001 b 3
0010 c 3
0011 c 3
0100 a 2
0101 a 2
0110 a 2
0111 a 2
1000 d 4
1001 e 4
1010 g 3
1011 g 3
1100 f 2
1101 f 2
1110 f 2
1111 f 2
a 출력
2 비트만 사용
①
g 출력
3 비트만 사용
②

복원 방법

- **01101011000001…을 복원하는 과정**
  - 먼저 가장 긴 허프만 코드의 길이L = 4이므로, 입력의 처음 4비트인 '0110'을 가져옴
  - 룩업 테이블에서 '0110'에 해당되는 테이블 항목 [0110, a, 2]를 찾아 'a'를 출력
  - a의 코드 길이가 2이므로, 뒷부분의 2비트인 '10'과 입력에서 그 다음 2비트인 '10'을 가져와 합쳐진 '1010'에 해당되는 테이블 항목을 찾으면 [1010, g, 3]가 되어, 'g'를 출력
  - g의 코드길이가 3이므로, 나머지 1비트인 '0'과 입력에서 그 다음 3비트인 '110'을 가져와 합쳐진 '0110'에 해당되는 항목을 찾으면 [0110, a, 2]이므로, 'a'를 출력
  - 이러한 방법으로 디코딩을 계속하면 d, c, …로 출력되고, 최종 복원된 문자열은 a g a d c…이다.

수행시간

- **허프만 트리 만들기는 먼저 최소힙을 구성하는데 O(N) 시간 소요. 단, N은 문자의 수**
- **최소힙에서 N-1회의 while-루프가 수행되는데, 루프 내에서는 2번의 delete_min과 1번의 insert가 각각 수행**
  - delete_min이나 insert는 downheap 과 upheap을 수행하므로 각각 O(logN) 시간 소요
- **허프만 트리를 만드는 시간: O(N) + (N-1)O(logN) = O(NlogN)**
- **허프만 코드 계산: 트리에서 전위순회를 수행하므로 트리의 노드 수에 비례 O(N)**

수행시간

- **압축하는 시간:**
  - 입력 파일의 모든 문자를 스캔하여 빈도 수를 계산하는 시간 = O(N)
  - 각 문자를 허프만 코드로 변환하는 시간 = O(N)
- **룩업테이블을 사용하여 복원하는 시간도 비트 스트링에서 스트링을 읽어올 때마다 한 개의 문자를 출력하므로 O(N)**

압축 성능

- **허프만 알고리즘은 입력에 민감**
- **최악의 경우는 입력 파일의 문자들이 모두 같은 빈도 수를 갖는 경우**
  - 허프만 트리를 만들면 완전이진트리와 유사한 형태를 가지게 되고 모든 문자가 거의 같은 길이의 코드를 가짐
- **파일에 문자들의 빈도 수가 고르지 않게 분포할 때에 우수한 압축 성능을 보임**

7.3 기타 우선순위큐

- Leftist 힙
- Skew 힙
- 이항힙(Binomial Heap)
- 피보나치힙(Fibonacci Heap)

요약

- 우선순위큐는 가장 높은 우선순위를 가진 항목을 접근 또는 삭제하는 연산과 삽입 연산을 지원
- 이진힙은 완전이진트리로서 부모의 우선순위가 자식의 우선순위보다 높은 자료구조
- 허프만 코딩은 빈도 수가 높은 문자에 짧은 이진코드를 부여하고, 빈도 수가 낮은 문자에 긴 이진코드를 부여하여 압축 효율을 높인다.
- 허프만 알고리즘은 최악의 경우는 입력 파일의 문자들이 모두 같은 빈도 수를 갖는 경우이고, 파일에 문자들의 빈도 수가 고르지 않게 분포할 때 우수한 압축 성능보임

- Leftist 힙은 왼쪽 부분으로 치우진 구조를 가지며, 그 이유는 rightmost 경로의 길이를 logN보다 크지 않게 하기 위함
- Skew 힙은 구조적 제약조건이 없으며 무조건 현재 노드의 좌우 서브트리를 교환
- 이항힙은 노드들이 힙의 속성을 만족하는 이항트리들로 구성
- 피보나치힙은 힙속성을 만족하는 트리들의 집합이며, 트리의 각 노드x에 대해 x의 자식 수가 k이면 x는 적어도Fk+2개의 자손(자신 포함)을 가짐
- 피보나치힙에서는 delete_min연산을 수행할 때까지 다른 연산들을 수행할 때 힙의 구조적인 일 처리를 미루다가 delete_min 연산을 수행하면서 힙의 구조를 정상화

힙 자료구조들의 연산의 수행시간 비교

	insert	delete_min	combine	decrease_key	delete
이진힙	O(logN)	O(logN)	O(N)	O(logN)	O(logN)
Leftist힙	O(logN)	O(logN)	O(logN)	-	-
Skew힙	O(logN)†	O(logN)†	O(logN)†	-	-
이항힙	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)
피보나치힙	O(1)	O(logN)†	O(1)	O(1)†	O(logN)†

† 상각(amortized)시간