11장 정렬

순서

선택 정렬 (1)

- 기본 개념
  - 잘못된 위치에 들어가 있는 원소를 찾아 그것을 올바른 위치에 집어 넣는 원소 교환 방식으로 정렬
- 방법
  - 가장 작은 원소를 찾아 첫 번째 위치의 원소와 교환
  - 두 번째로 작은 원소를 찾아 두 번째 위치의 원소와 교환
  - 나머지 a[i], …, a[n-1] 원소 중 가장 작은 원소를 선택해서 a[i] 원소와 계속 교환
- **SelectionSort 알고리즘**

```
selectionSort(a[])
    for (i ← 0; i < a.length; i ← i + 1) do {
        a[i], … , a[n-1] 중에서 가장 작은 원소 a[k]를 선택;
        a[k]를 a[i]와 교환;
    }
```

  - 전체 비교 연산 수 = n(n-1)/2, 시간 복잡도 = $O(n^2)$

선택 정렬 (2)

- **SelectionSort의 실행 예**

	a [] :	[0]	[1]	[2]	[3]	[4]	
	초기 :	5	2	8	3	1	
for 루프	i = 0 :	1	2	8	3	5	a [0]과 a [4] 교환
	i = 1 :	1	2	8	3	5	a [1]과 a [1] 교환
	i = 2 :	1	2	3	8	5	a [2]와 a [3] 교환
	i = 3 :	1	2	3	5	8	a [3]과 a [4] 교환

파란색 : 서로 교환된 원소

선택 정렬 (3)

- **Sorting 클래스 메소드 멤버 구현**

```
public class Sorting  {
   public static void selectionSort(int[] a)  {
      int i, j, min;
      for (i = 0; i < a.length – 1; i++)  {
           for (j = i+1, min = i; j < a.length; j++)  {     // a[j] … a[a.length] 사이에
                                                            // 가장 작은 원소의 인덱스(min)을 찾음
              if (a[j] < a[min]) min = j;
       }
       swap(a, min, i);     // a[i]와 a[min]을 교환
   }

   public static void swap(int[] a, int j, int k)  {     // a[j]와 a[k]를 교환
      int temp = a[j];   a[j] = a[k];   a[k] = temp;
   }

   // • • • • • •
   // bubleSort(), insertionSort(), quickSort() 등 기타 다른 메소드들을 추가로 포함
   // • • • • • •
}
```

선택 정렬 (4)

- **SortMain 클래스 (샘플 프로그램)**

```
public class SortMain  {
    public static void main(String[] args)  {
        int[] a = { 5, 2, 8, 3, 1};
        System.out.println("정렬전 배열 원소 : ");
        int i;
        for (i = 0; i < a.length; i++)
            System.out.print(a[i] + "  ");
        System.out.println();
        Sorting.selectionSort(a);     // 원하는 정렬 메소드를 선택
        System.out.println("정렬된 배열 원소 : ");
        for (i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();
    }
}
```

<실행결과>
정렬전 배열 원소 :
5 2 8 3 1
정렬된 배열 원소 :
1 2 3 5 8

버블 정렬 (1)

- 기본 개념
  - 배열을 검사하여 두 인접한 원소가 오름차순 정렬 순서에 맞지 않으면 이들을 서로 교환
- 방법
  - a[0]과 a[1]을 비교하여 정렬순서에 맞도록 교환
  - a[1]과 a[2]을 비교하여 정렬순서에 맞도록 교환
  - a[n-2]와 a[n-1]을 비교하여 정렬순서에 맞도록 교환
  - 제일 큰 원소가 배열의 n-1 위치로 이동 (→패스 1)
  - 배열 처음부터 다시 비교 및 정렬

    …
  - 마지막 패스로 a[0]과 a[1]을 비교하여 정렬 (→패스 n-1)

버블 정렬 (2)

- **BubbleSort 알고리즘**
  - 의사코드

```
bubbleSort(a[])
   for (i ← a.length - 1;  i ≥ 0;  i ← i - 1) do  {
        for (j ← 0;  j < i;  j ← j+1) do  {
             if (a[j] > a[j+1]) then a[j]와 a[j+1]을 교환;
        }
   }
```

  - Java 코드

```
public static void bubbleSort(int[] a)  {
   int i, j;
   for (i = a.length – 1;  i ≥ 0;  --i)
        for (j = 0;  j < i; j++)
             if (a[j] > a[j+1])
                  swap(a, j, j+1);
   }
```

  - 전체 비교 연산 수 = n(n-1)/2, 시간 복잡도 = $O(n^2)$

버블 정렬 (3)

- **BubbleSort의 실행 예**

	a [] :	[0]	[1]	[2]	[3]	[4]	
패스 1 :	초기 :	5	2	8	3	1	
		2	5	8	3	1	
		2	5	3	8	1	
		2	5	3	1	8	a [4] 확정
패스 2 :		2	3	5	1	8	
		2	3	1	5	8	a [3] 확정
패스 3 :		2	1	3	5	8	a [2] 확정
패스 4 :		1	2	3	5	8	a [1] 확정, a [0] 자동 확정

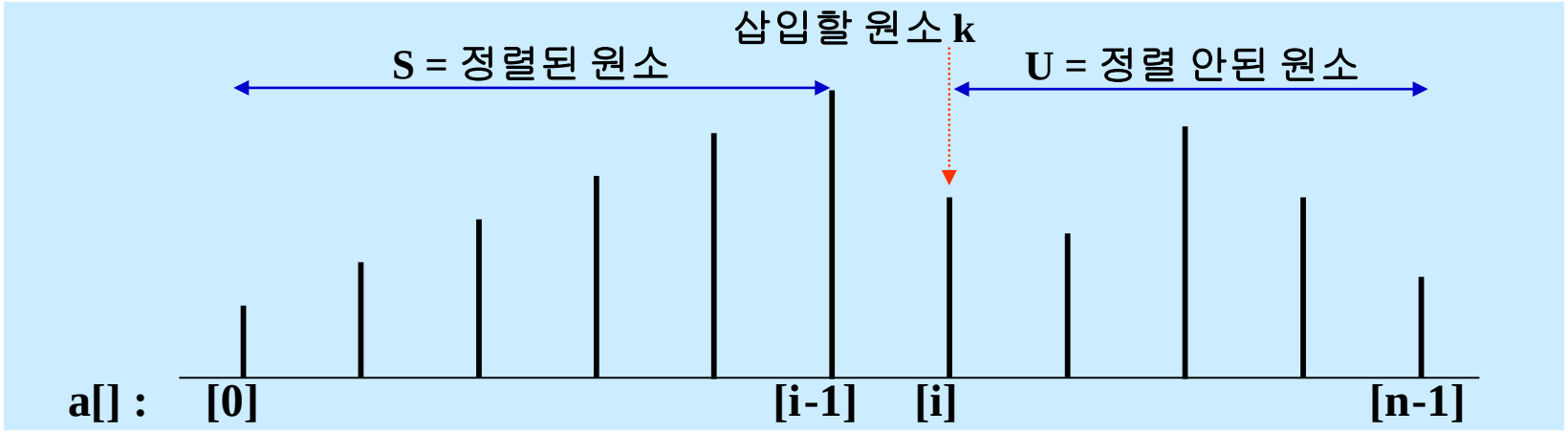
파란색 : 서로 교환되는 원소

삽입 정렬 (1)

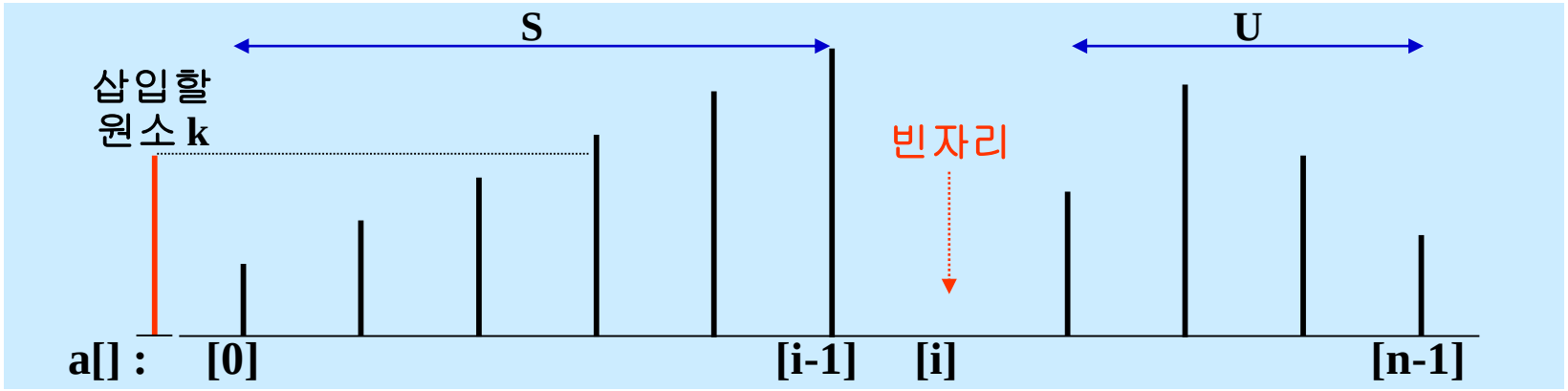
- 기본 개념
  - 가정사항
    - S : 정렬되어 있는 배열의 왼쪽 부분
    - U : 정렬되어 있지 않은 배열의 오른쪽 부분
  - 정렬되어 있지 않은 U의 왼쪽 끝에서 삽입할 원소를 찾아 정렬되어 있는 S의 적절한 위치에 삽입
- 방법
  - U의 왼쪽에서 삽입할 원소 k를 선택
  - k를 삭제 (빈자리)
  - S에 있는 k보다 큰 원소들을 오른쪽으로 이동
  - k를 S에 만들어진 빈자리에 삽입
  - U의 모든 원소들이 S에 삽입될 때까지 반복
  - 시간 복잡도 = $O(n^2)$

삽입 정렬 (2)

- **InsertionSort에서 원소 k(=a[i])의 삽입과정(1)**



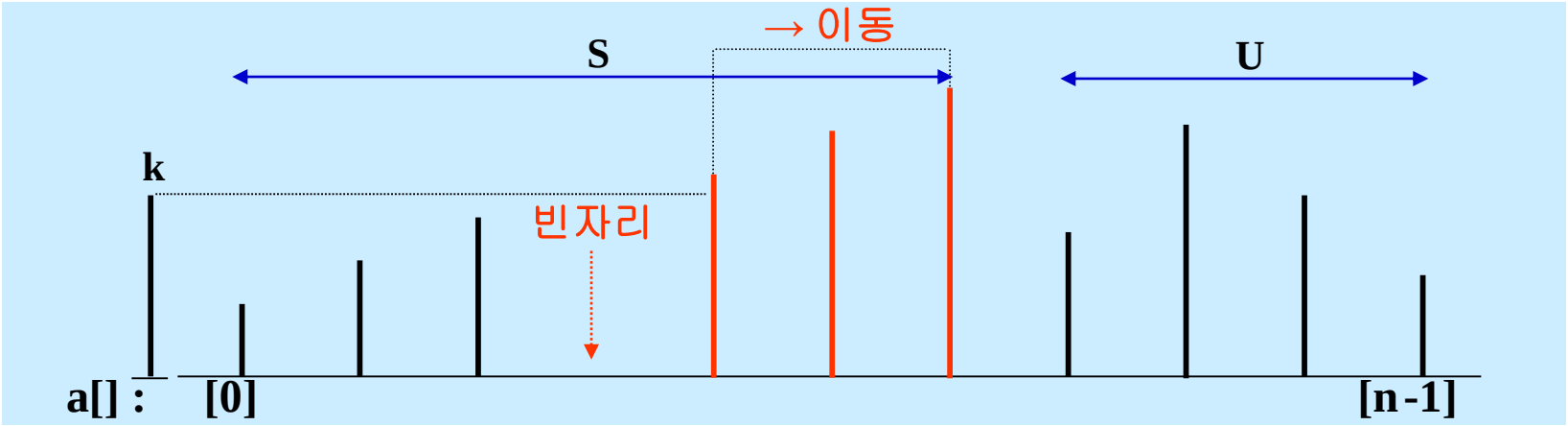
(a) 초기 단계



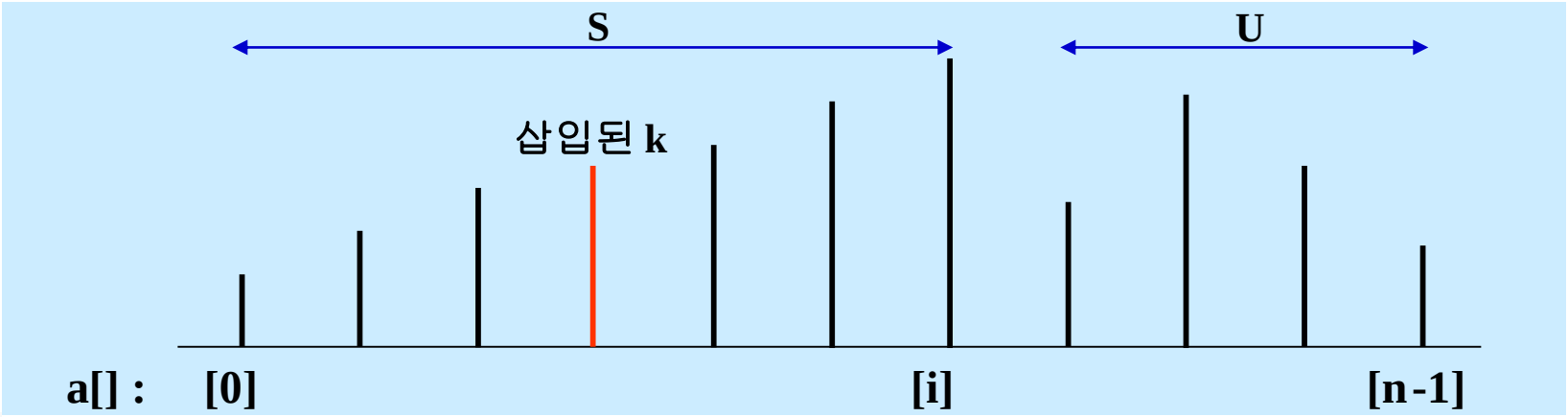
(b) k[i]를 제거하여 빈자리를 만듦(k ← k[i])

삽입 정렬 (3)

- **InsertionSort에서 원소 k(=a[i])의 삽입과정(2)**



(c) K보다 큰 원소는 오른편으로 한 자리씩 이동



(d) K를 빈자리에 삽입

삽입 정렬 (4)

- **InsertionSort 알고리즘**

```
insertionSort(a[])
        // 원소 ak는 a[i], 0 < i < n,
        // 원소 k(=a[i], 0 < i < n)를 부분 배열 a[0 : i-1]에 오름차순으로 삽입
    for (i = 1;  i < n;  i ← i + 1)  {   // 두 번째 원소 a[1]부터
        k ← a[i];     // k는 임시 저장소
        j ← i;
        if (a[j-1] > k) then move ← true
        else move ← false;
        while (move) do  {
                a[j] ← a[j-1];   // a[j-1]을 오른쪽으로 한자리 이동시켜 빈자리를 만듦
                j ← j-1;
                if (j > 0 and a[j-1] > k)
                then move ← true
                else move ← false;
        }
        a[j] ← k;   //k를 빈자리에 삽입
    }   // for
end InsertionSort()
```

삽입 정렬 (5)

- **InsertionSort 실행 예**
  - 배열 a = [3, 1, 9, 8, 4]

	[0]	[1]	[2]	[3]	[4]
a [] :	3	1	9	8	4
i = 1 : k = 1	[3	]	9	8	4
	[1	3]	9	8	4
i = 2 : k = 9	[1	3	]	8	4
	[1	3	9]	8	4
i = 3 : k = 8	[1	3	9	]	4
	[1	3	8	9]	4
i = 4 : k = 4	[1	3	8	9	]
	[1	3	4	8	9]

[ ] : 정렬된 원소의 배열

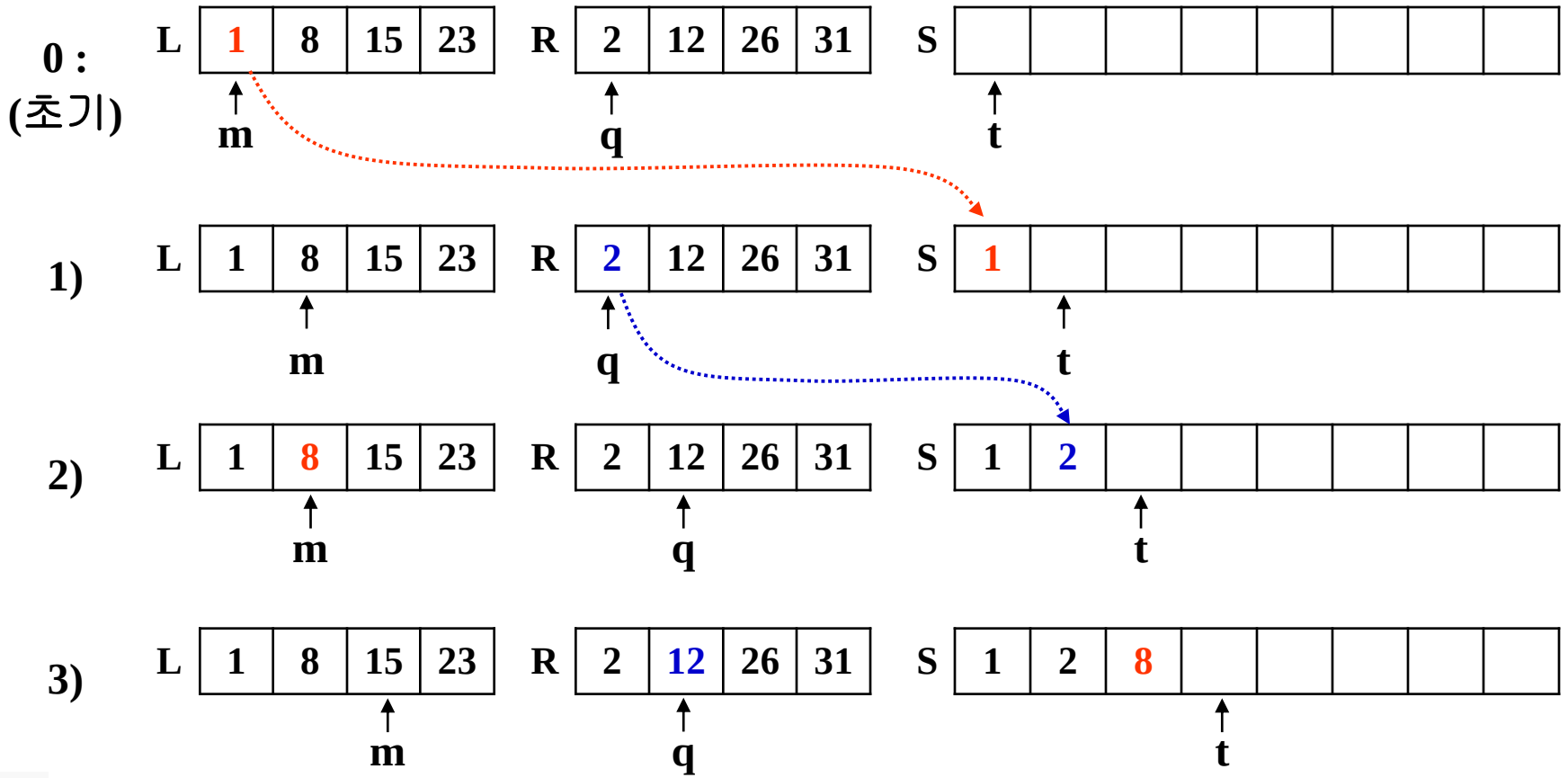
합병 정렬 (1)

- 기본 개념
  - 배열을 이등분하여 각각을 정렬한 후 합병
- 방법
  - 배열 a를 L과 R로 이등분한 후 배열 L과 R을 각각 정렬
  - 정렬된 배열 L과 R에서 작은 원소를 삭제하여 새로운 임시 공백 배열 S에 차례대로 삽입
  - 원래의 배열 a에 복사
  - 시간 복잡도 : $O(n\log n)$
- **MergeSort 알고리즘**

```
mergeSort(a[], m, n)
    if (a[m : n]의 원소수 > 1) then  {
          divide a[m : n] into a[m : middle] and a[middle+1 : n];
          mergeSort(a[], m, middle);
          mergeSort(a[], middle+1, n);
          merge(a[m : middle], a[middle+1 : n]);
    }
end MergeSort()
```

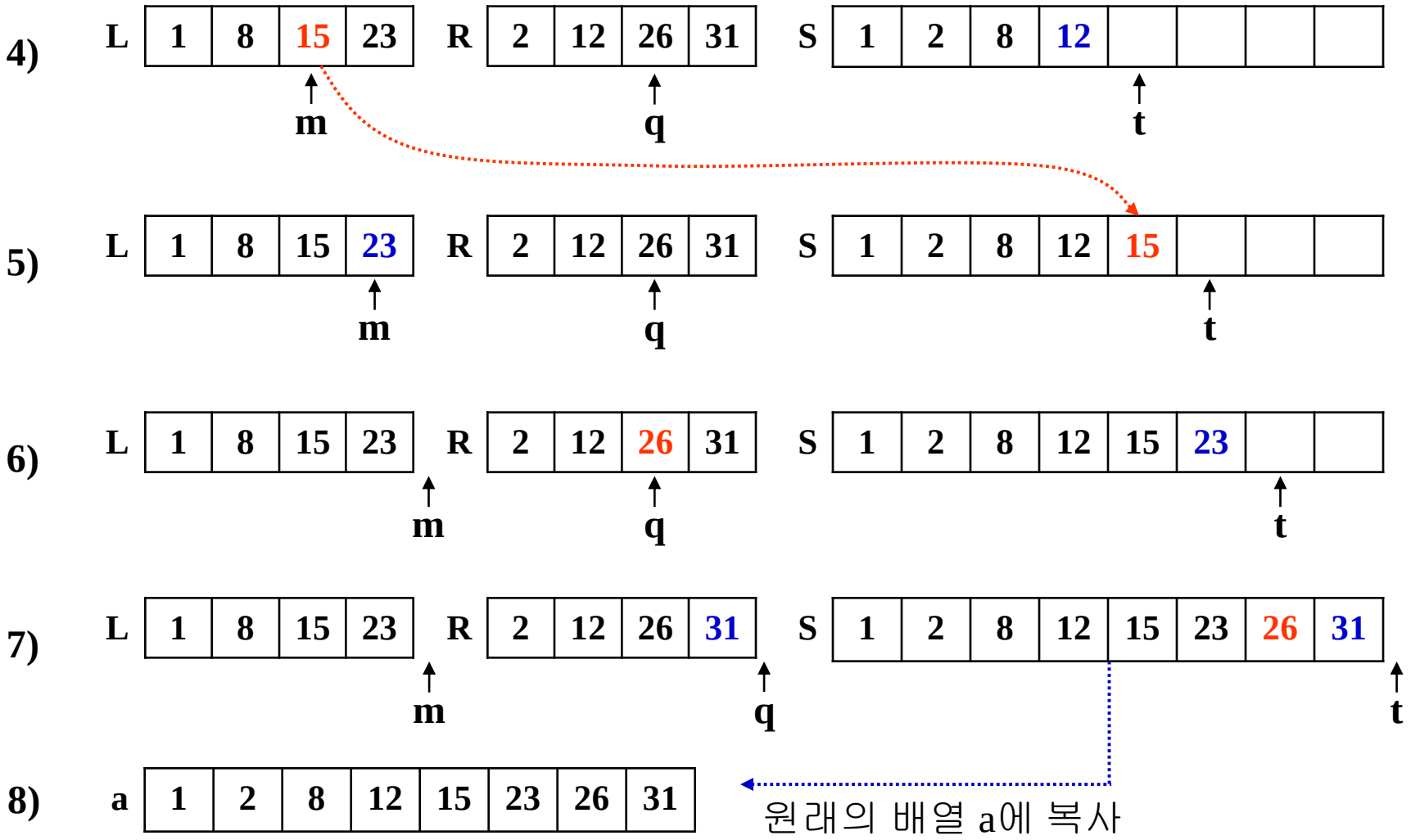
합병 정렬 (2)

- **Merge 알고리즘의 수행 과정(1)**
  - a = [8, 1, 23, 15, 31, 2, 26, 12]



합병 정렬 (3)

- **Merge 알고리즘의 수행 과정(2)**



합병 정렬 (4)

- **mergeSort 프로그램(1)**

```
public static void mergeSort(int[] a)  {
        // 합병 정렬의 메인 메소드
   int[] temp[a.length];
   internalMergeSort(a, temp, 0, a.length-1);
}

private static void internalMergeSort(int[] a, int[] temp, int m, int n)  {
        // 순환 호출을 하는 mergeSort()의 보조 메소드
   if (m < n)  {    // 정렬할 원소가 2개 이상인 경우
      int middle = (m+n) / 2;
      internalMergeSort(a, temp, m, middle);
      internalMergeSort(a, temp, middle+1, n);
      merge(a, temp, m, middle, middle+1, n);
   }
}
```

합병 정렬 (5)

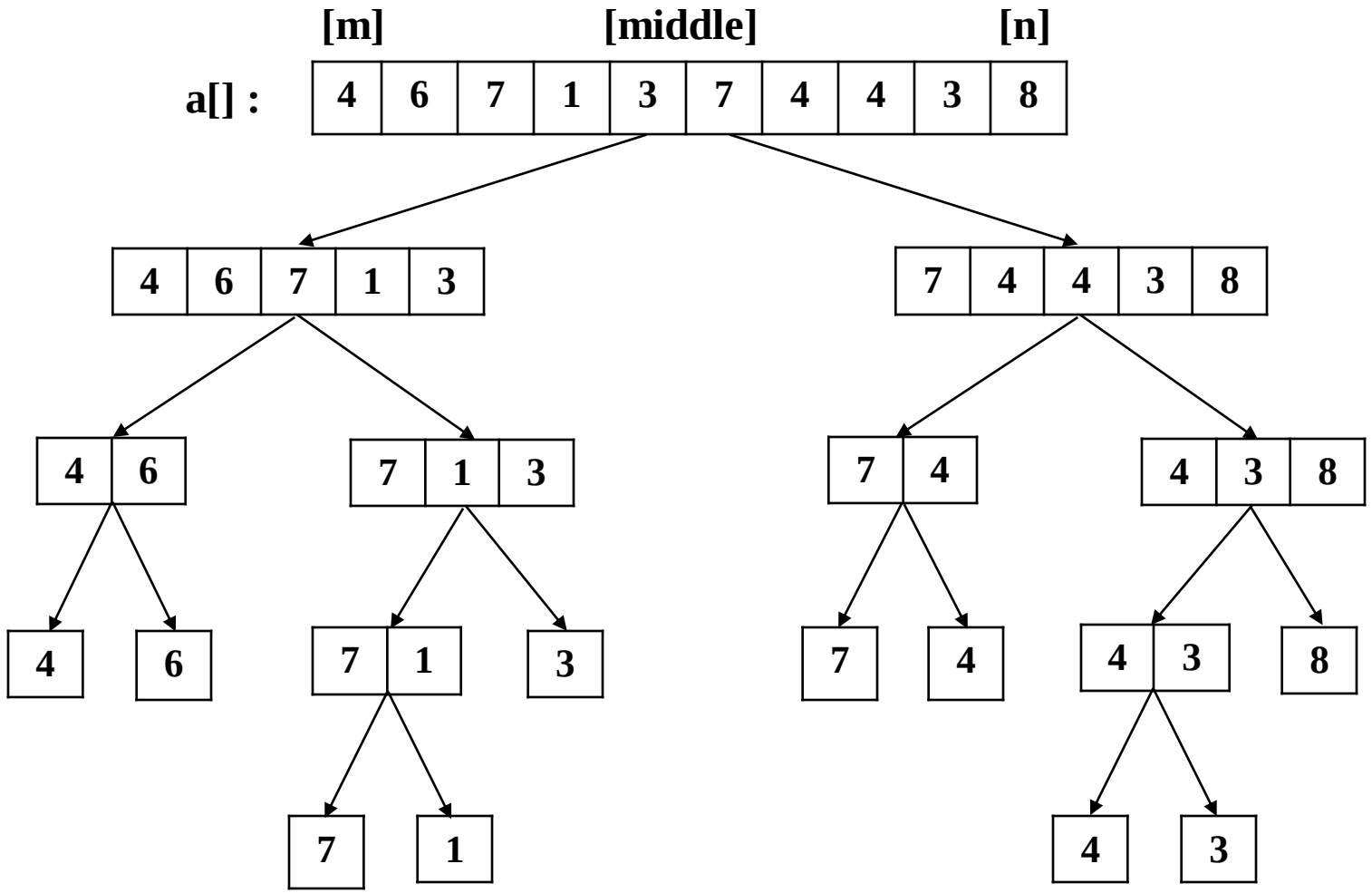
- **mergeSort 프로그램(2)**

```
private static void merge(int[] a, int[] temp, int m, int p, int q, int n)  {
        // internalMergeSort()의 보조 메소드
   int t = m;
   int numElements = n – m + 1;
   while (m <= p && q <= n)  {
        if (a[m] < a[q])
            temp[t++] = a[m++];
        else temp[t++] = a[q++];
   }
   while (m <= p)     // 왼쪽 부분 배열에 원소가 남아 있는 경우
        temp[t++] = a[m++];
   while (q <= n)     // 오른쪽 부분 배열에 원소가 남아 있는 경우
        temp[t++] = a[q++];
   for (int i = 0,  i < numElements; i++, n--)     // 배열 temp[]를 a[]로 복사
       a[n] = temp[n];
}
```

- 단점 : 주어진 배열과 동일한 크기의 임시배열 **temp[]**가 필요

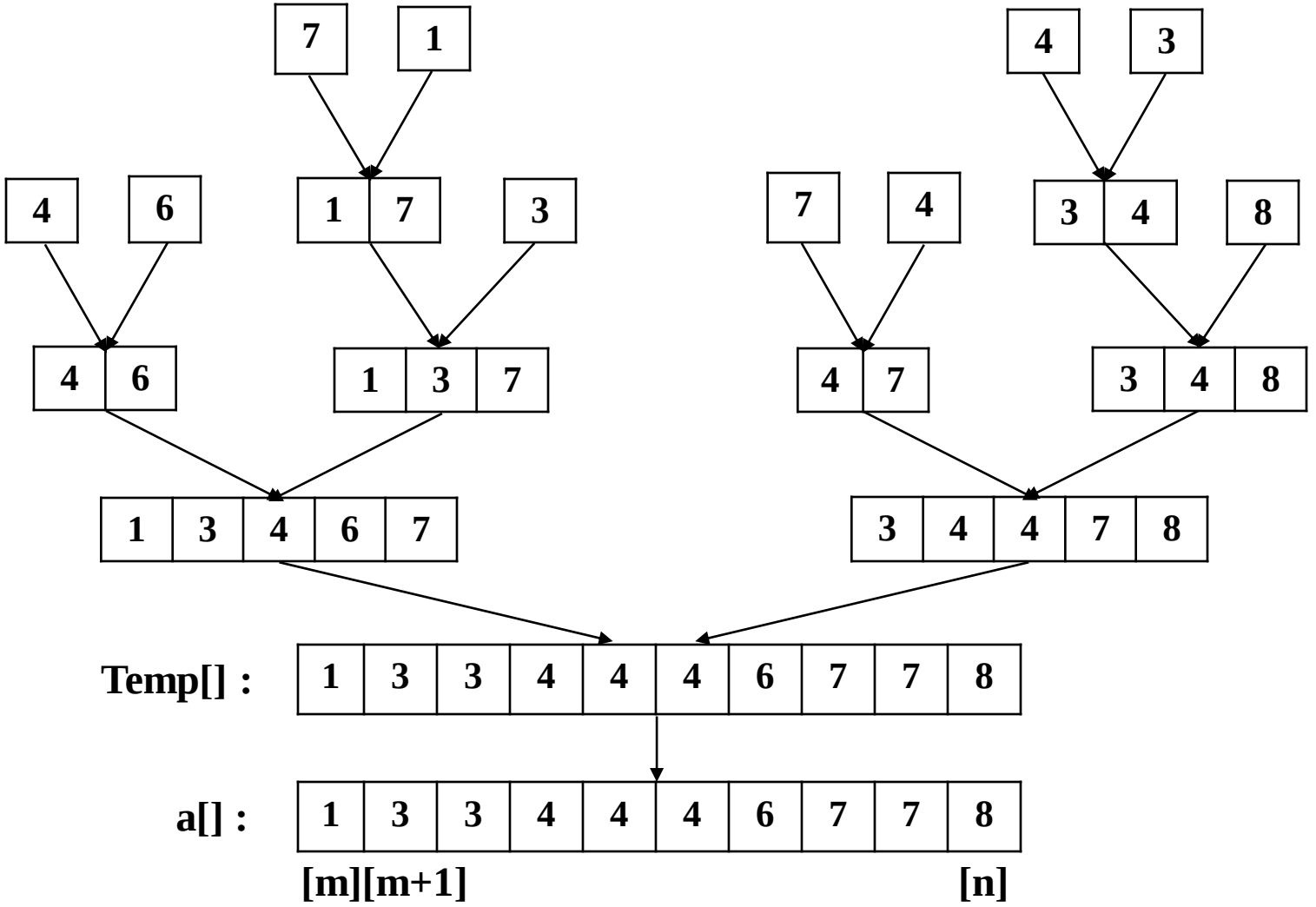
합병 정렬 (6)

- **MergeSort 알고리즘의 분해 과정**



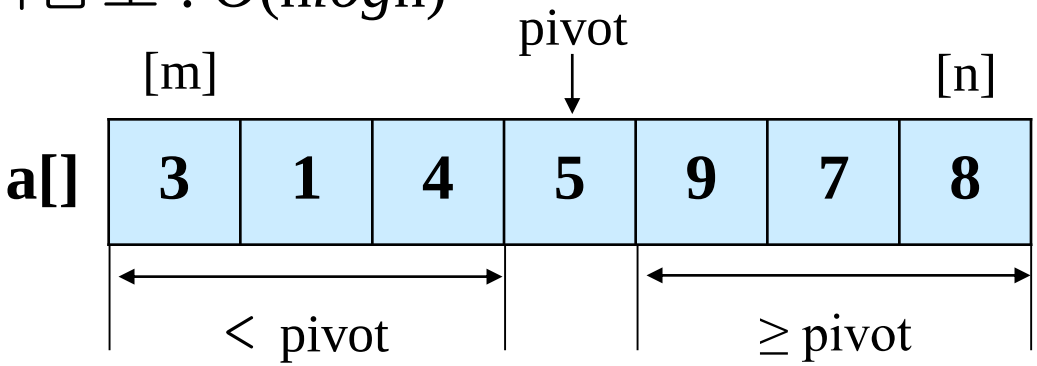
합병 정렬 (7)

- **MergeSort 알고리즘의 합병 과정**



퀵 정렬 (1)

- 기본 개념
  - 분할 정복(divide and conquer) 정렬 방법의 하나
- 방법
  - 배열 a[m:n]의 한 원소를 pivot으로 선정
  - pivot 을 기준으로 두 개의 파티션으로 분할
    - 왼쪽 파티션 : pivot보다 작은 값의 원소들로 구성
    - 오른쪽 파티션 : pivot 보다 크거나 같은 값의 원소들로 구성
  - 각각의 파티션에 대해 다시 퀵 정렬을 순환 적용
    - pivot 선정 및 파티션 분할
  - 시간 복잡도 : $O(n \log n)$



퀵 정렬 (2)

- **QuickSort 알고리즘**
  - 기본 골격

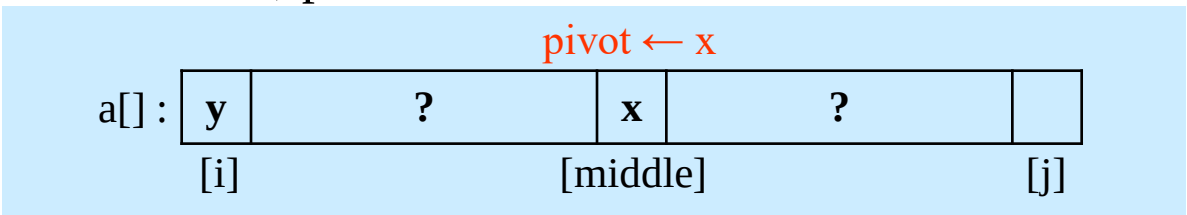
```
if (a[m : n]의 원소 ≤ 1) then return;
(a[]의 원소 하나를 pivot으로 선정하여 a[m : n]을
   leftPartition과 rightPartition으로 분할);
(QuickSort(leftPartition));
(QuickSort(rightPartition));
```

  - QuickSort 알고리즘

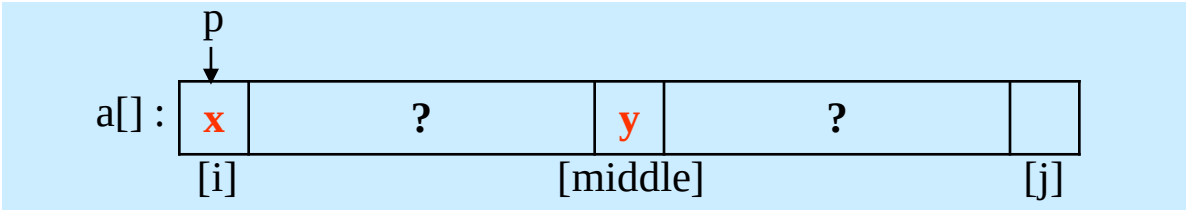
```
quickSort(a[], m, n)
            // 배열 a[]의 부분 배열 a[m : n]을 오름차순으로 정렬
   if (m ≥ n) then return;     // 정렬할 원소 수가 0이거나 1일 때는 복귀
   p ← partition(a, m, n);     // p는 파티션이 끝난 뒤에 사용된 pivot의 인덱스
   quickSort(a[], m, p-1);
   quickSort(a[], p+1, n);
end quickSort()
```

퀵 정렬 (3)

- **partition 알고리즘**
  - 목적
    - 부분 배열 a[i : j]의 중앙 원소를 pivot 값으로 정하고 왼쪽과 오른쪽 파티션으로 분할
  - 단계별 분할 과정
    - a[i : j]의 중앙 인덱스 값 middle을 선정하여 a[middle]의 원소 x를 분할의 기준값, pivot으로 정한다.

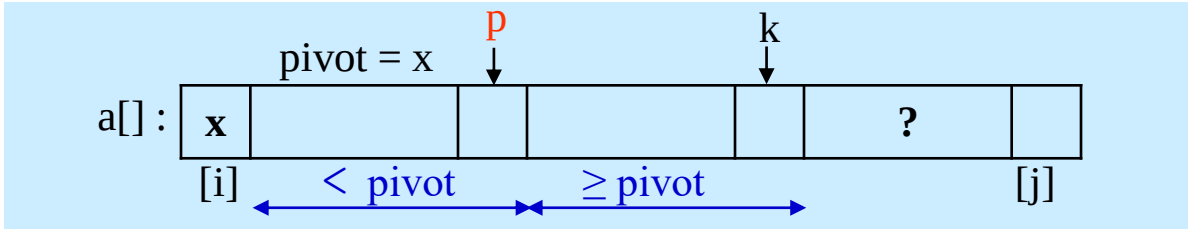


    - a[middle]의 원소 x와 a[]의 제일 왼쪽 끝에 있는 a[i]의 원소 y를 서로 교환 (pivot 인덱스 p는 i를 지시)

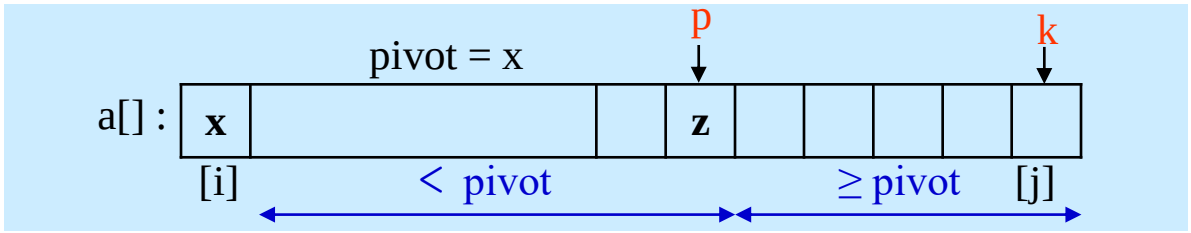


퀵 정렬 (4)

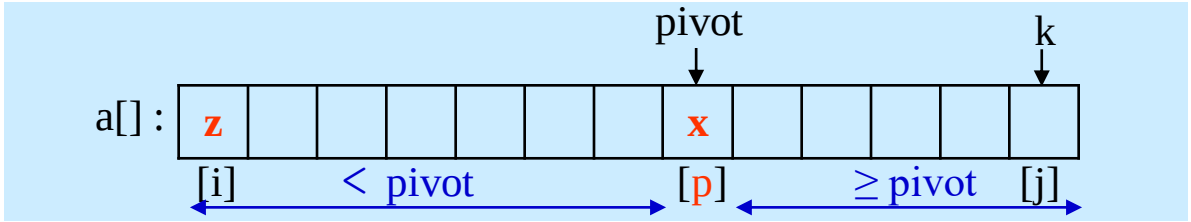
- a[i+1 : j]의 모든 원소 a[k]를 검사하여
  - pivot 보다 작을 경우 : a[i+1 : p]에 삽입 (p를 1 확장)
  - pivot 보다 크거나 같을 경우 : a[p+1 : j]에 삽입
  - 항상 (**a[i+1 : p] < pivot, a[p+1 : k] ≥ pivot**)을 유지



- 모든 원소의 검사 완료 후 pivot 인덱스가 확정되면 a[i]와 a[p]를 교환하여 pivot 값을 제자리에 저장



- 분할 완료시 확정된 pivot 인덱스 p를 반환



퀵 정렬 (5)

- **partition 알고리즘**

```
partiton(a[], i, j)
    middle ← (i + j) / 2;     // middle은 a[]의 중앙 인덱스 값
    pivot ← a[middle];     // a[]의 중앙 원소값을 pivot으로 설정
    a[middle] ← a[i];     // a[i]와 a[middle]을 서로 교환
    a[i] ← pivot;     //a[i]는 pivot 값을 보관
    p ← i;     // p는 두 파티션의 경계를 지시하는 인덱스
    for (k ←i+1;  k ≤ j;  k ← k+1) do  {
                        // a[i]를 제외한 a[i+1 : j]에 있는 모든 원소 a[k]들을 검사하여
        if (a[k] < pivot) then  {     // a[k]가 pivot보다 작으면
            p ← p+1;                    // p를 1 증가시켜 a[k]를 p 인덱스 범위
            temp ← a[p];               // 안으로 포함되게 함
            a[p] ← a[k];
            a[k] ← temp;
        }
    }
    temp ← a[i];     // a[i]와 a[p]를 교환
    a[i] ← a[p];
    a[p] ← temp;
    return p;
end partition()
```

퀵 정렬 (6)

- 퀵 정렬 수행 과정 (1)
  - 배열 a = [3 1 4 5 9 8 7]에서
    - 초기 배열 a[i : j], i = 0, j = 6

      a : [0] [1] [2] [3] [4] [5] [6]

      [3 1 4 5 9 8 7]
    - pivot을 중앙 원소값(a[3]) 5로 선정하고 a[i] 원소와 교환

      p k

      [5 1 4 3 9 8 7]
    - k를 1씩 증가, a[i+1 : p]에 있는 원소가 pivot보다 작게 교환

      k

      p

      [5 1 4 3 9 8 7]
    - k가 j까지 증가되었을 때, p를 고정시키고 a[p]와 a[i] 원소 교환

      pivot, a[p]를 중심으로 2개의 파티션이 생성

      [3 1 4] 5* [9 8 7]

_는 교환될 원소, *는 각 단계에서 확정된 pivot

퀵 정렬 (7)

퀵 정렬 수행 과정 (2)

- 배열 원소 수의 pivot이 결정될 때까지 정렬을 수행

  파티션 i = 0, j = 2에 대해 순환적으로 퀵 정렬을 수행

  a : [0] [1] [2] [3] [4] [5] [6]

  [3 1 4] 5 [9 8 7]

  p k

  [1 3 4] 5 [9 8 7]

- P를 확정시키고 공백 파티션과 파티션 i = 1, j = 2를 생성

  1* [3 4] 5 [9 8 7]

- 파티션 i = 1, j = 2에 대해 다시 퀵 정렬을 수행

  p k

  1 [3 4] 5 [9 8 7]

  1 3* [4] 5 [9 8 7]

  1 3 4* 5 [9 8 7]

퀵 정렬 (8)

퀵 정렬 수행 과정 (3)

- 파티션 i = 4, j = 6에 대해 순환적으로 퀵 정렬을 수행

a : [0] [1] [2] [3] [4] [5] [6]

1	3	4	5	[8	9	7]
1	3	4	5	[8	7	9]
1	3	4	5	[7]	8*	[9]
1	3	4	5	7*	8	[9]
1	3	4	5	7	8	9*

- 제자리 알고리즘(in-place algorithm)
  - 별도의 배열을 사용하지 않고 원래의 배열 위에서만 정렬을 수행

퀵 정렬 (9)

- **Sorting 클래스의 메소드 멤버 구현**

```
public static void quickSort(int[] a)  {
            // 퀵 정렬의 메인 메소드
   internalQuickSort(a, 0, a.length-1);
}

private static void internalQuickSort(int[] a, int m, int n)  {
            // quickSort()의 보조 메소드
   int p;
   if (m > n) then return;
   p = partition(a, m, n);
   internalQuickSort(a, m, p-1);
   internalQuickSort(a, p+1, n);
}

private static int partition(int[] a, int m, int n)  {
            // internalQuicksort()의 보조 메소드
   • • • • •    // partition 알고리즘의 Java 코드
   return p;
}
```

히프 정렬 (1)

- 기본 개념
  - 최대 히프를 이용한 정렬 기법
- 방법
  - 정렬할 원소를 모두 공백 히프에 삽입
  - 루트 노드(가장 큰 원소)를 삭제하여 리스트 뒤에 삽입
  - 삽입된 원소를 제외한 나머지 원소들에 대해 반복 수행
  - 시간 복잡도 : $O(n\log n)$

```
for (i ← 0;  i < n;  i ← i+1) do  {
      insertHeap(Heap, a[i]);
}
for  (i ← n-1;  i ≥ 0,  i ← i-1) do  {
      a[i] ← deleteHeap(Heap);
}
```

히프 정렬 (2)

- **HeapSort 알고리즘**

```
heapSort(a[])
    n ← a.length-1;   // n은 히프 크기(원소의 수)
                      // a[0]은 사용하지 않고 a[1 : n]의 원소를 오름차순으로 정렬
    for (i ← n/2; i ≥ 1; i ← i-1) do {     // 배열 a를 히프로 변환
        heapify(a, i, n);                  // i는 내부 노드
    for (i ← n-1; i ≥ 1; i ← i-1) do {     // 배열 a[]를 오름차순으로 정렬
        temp ← a[1];     // a[1]은 제일 큰 원소
        a[1] ← a[i+1];     // a[1]과 a[i+1]을 교환
        a[i+1] ← temp;
        heapify(a, 1, i);
    }
end HeapSort()
```

- Heapify()를 호출하여 배열 a[1 : n]을 히프 구조로 변환
- 원소를 교환하여 최대 원소 저장
- Heapify()를 호출하여 나머지 원소를 히프로 재구성

히프 정렬 (3)

- **Heapify 알고리즘**

```
heapify(a[], h, m)
        // 루트 h를 제외한 h의 왼쪽 서브트리와 오른쪽 서브트리는 히프
        // 현재 시점으로 노드의 최대 레벨 순서 번호는 m
  ah = a[h];
  for (j ← 2*h;  j ≤ m;  j ← 2*j) do  {
    if (j < m) then
      if (a[j] < a[j+1]) then j ← j+1;   // j는 값이 큰 왼쪽 또는 오른쪽 자식 노드
    if (ah ≥ a[j]) then exit
    else a[j/2] ← a[j];    // a[j]를 부모 노드로 이동
  }
  a[j/2] ← ah;
end heapify()
```

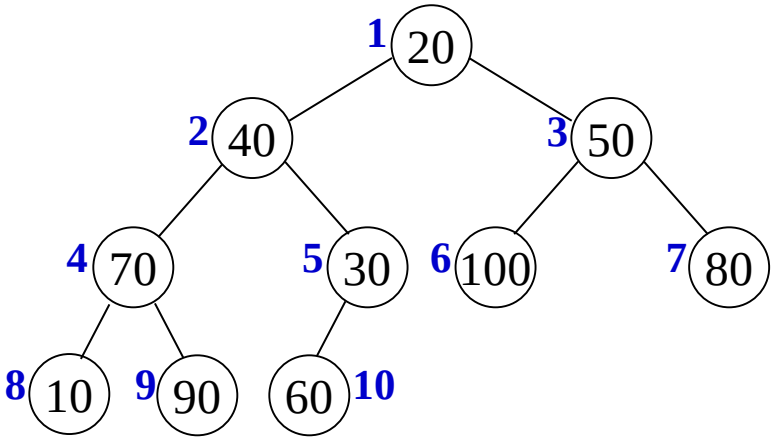
  - 완전 2진 트리를 히프로 변환

히프 정렬 (4)

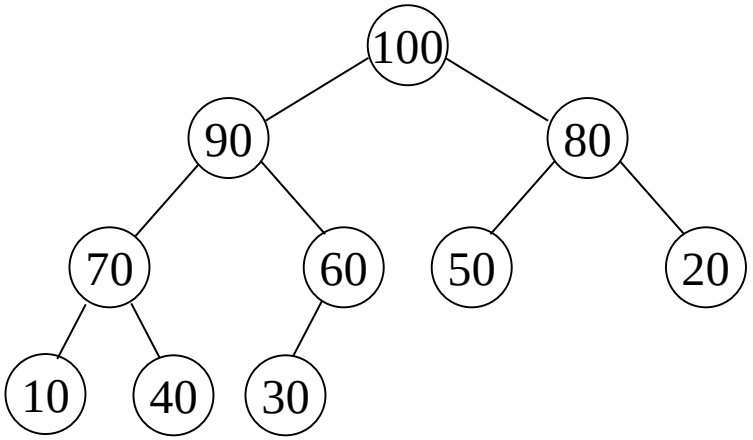
- **HeapSort(a[], 10) 실행 예(1)**
  - a = [20 40 50 70 30 100 80 10 90 60]

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
a[] :	-	**20**	**40**	**50**	**70**	**30**	**100**	**80**	**10**	**90**	**60**

완전 2진 트리 표현



히프 구조 변환



히프 정렬 (5)

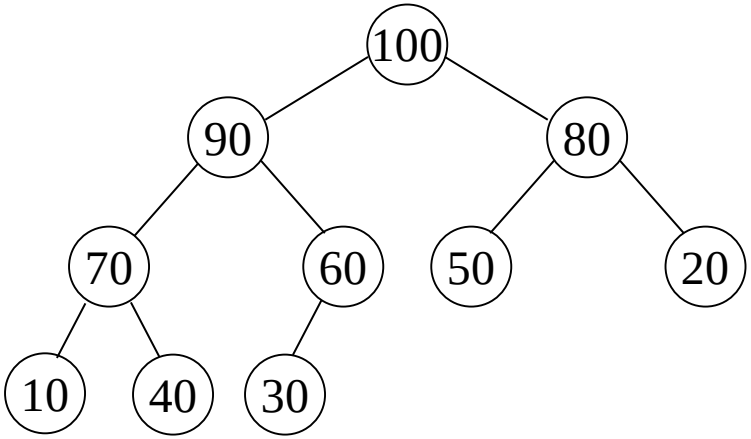
- **heapSort(a[], 10) 실행 예(2)**

a[] :	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
초기 :	20	40	50	70	30	100	80	10	90	60
히프(크기 10)로 변환 :	100	90	80	70	60	50	20	10	40	30
(두번째 for 루프) i = 9 :	90	70	80	40	60	50	20	10	30	**100**
(i = 히프 크기) i = 8 :	80	70	50	40	60	30	20	10	**90**	
i = 7 :	70	60	50	40	10	30	20	**80**		
i = 6 :	60	40	50	20	10	30	**70**			
i = 5 :	50	40	30	20	10	**60**				
i = 4 :	40	20	30	10	**50**					
i = 3 :	30	20	10	**40**						
i = 2 :	20	10	**30**							
i = 1 :	**10**	**20**								

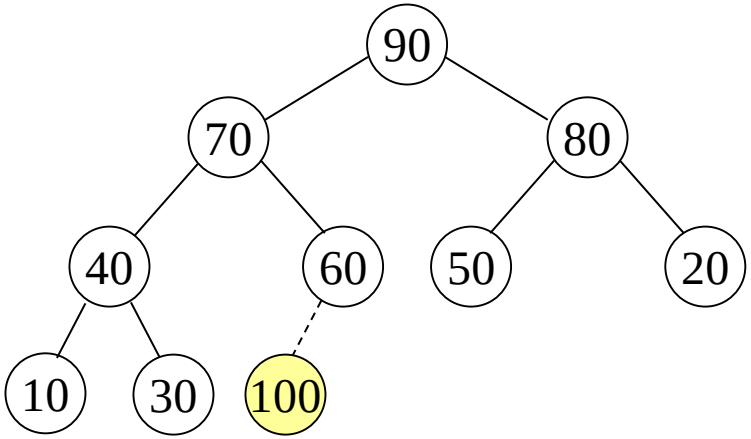
파란색 : 정렬 완료된 원소값

히프 정렬 (6)

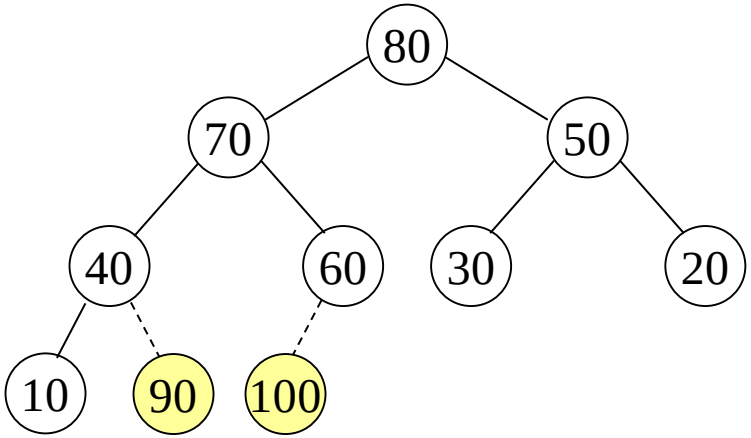
- 히프 정렬 과정에서의 히프 변화(1)



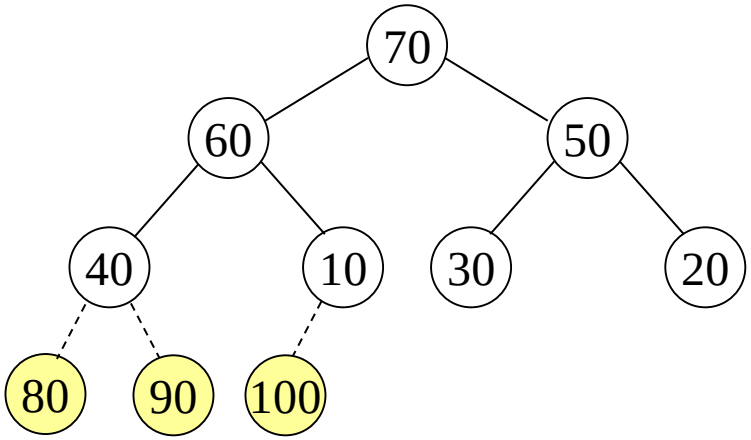
(a) 히프 크기 = **10**



(b) 히프 크기 = **9**



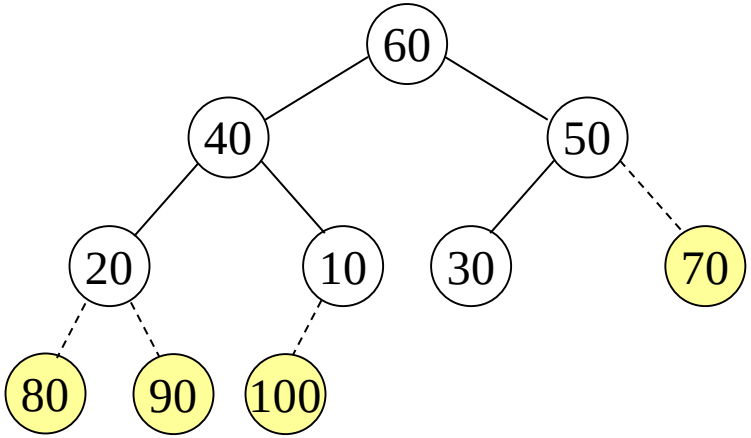
(c) 히프 크기 = **8**



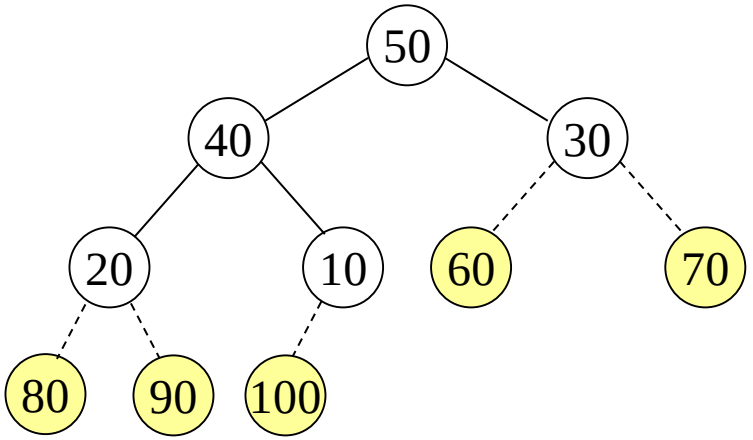
(d) 히프 크기 = **7**

히프 정렬 (7)

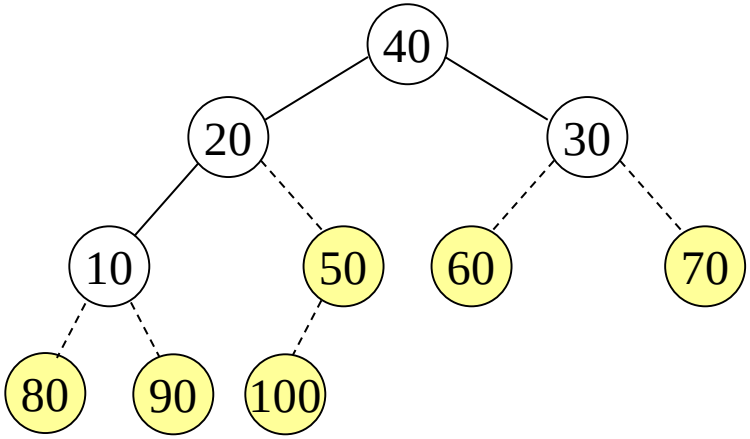
- 히프 정렬 과정에서의 히프 변화(2)



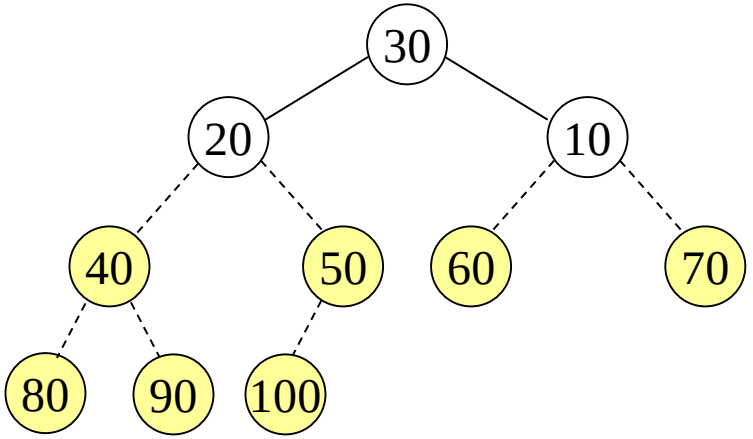
(e) 히프 크기 = 6



(f) 히프 크기 = 5



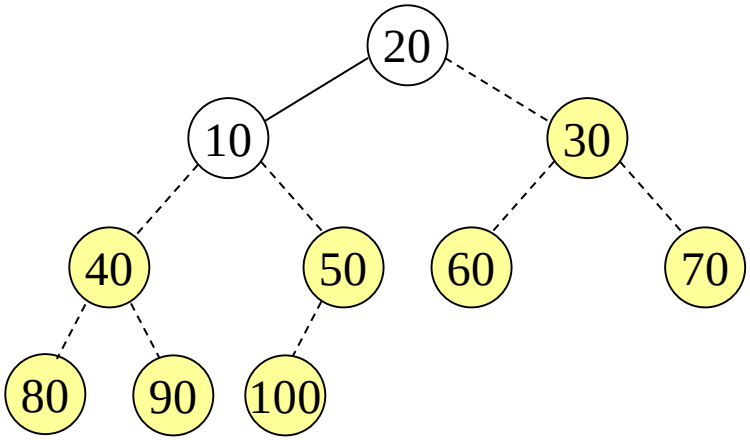
(g) 히프 크기 = 4



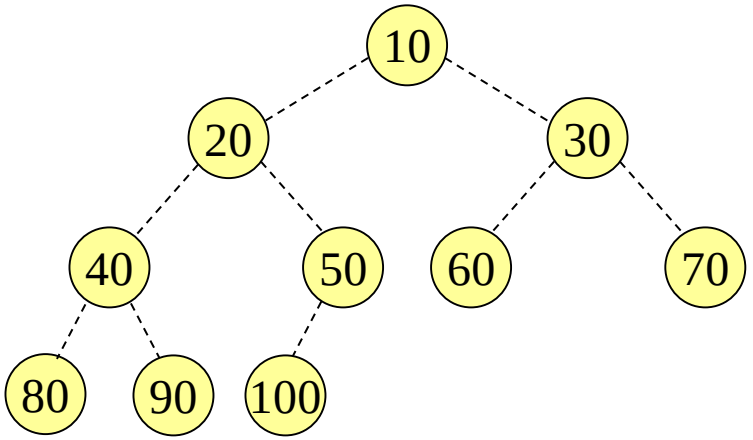
(h) 히프 크기 = 3

히프 정렬 (8)

- 히프 정렬 과정에서의 히프 변화(3)



(i) 히프 크기 = **2**



(j) 히프 크기 = **1 :** 정렬 종료

히프 정렬 (9)

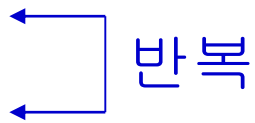
- **Sorting 클래스의 메소드 멤버 구현**

```
public static void heapSort(int[] a)  {
            // 히프 정렬의 메인 메소드
   • • • • •    // HeapSort 알고리즘의 Java 코드
   heapify(a, 1, i);
}

private static void heapify(int[] a, int h, int m)  {
            // heapSort()의 보조 메소드
   • • • • •    // heapify 알고리즘의 Java 코드
}
```

쉘 정렬 (1)

- 기본 개념
  - 원소의 비교 연산과 먼 거리의 이동을 줄이기 위해 몇 개의 서브리스트로 나누어 삽입 정렬을 반복 수행
- 방법
  - 정렬에서 사용할 간격(interval) 결정
    - 전체 원소 수 n의 1/3, 다시 이것의 1/3 …
    - interval이 작아질수록 짧은 거리를 이동하고 원소 이동이 적음
  - 첫 번째 interval에 따라 서브리스트로 분할
  - 각 서브리스트에 대해 삽입 정렬 수행
  - 두 번째 interval에 따라 서브리스트로 분할
  - 각 서브리스트에 대해 삽입 정렬 수행



  - 리스트 전체에 대해 삽입 정렬 수행 (마지막 interval = 1)
  - 시간 복잡도 : $O(n^{1.25})$

쉘 정렬 (2)

- 쉘 정렬 예 (1) : interval = 5

i :	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
a[i] :	3	14	12	4	10	13	1	5	2	7	9	6	8	11

(a) 배열 : a[0 :13]

i :	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
a[i] :	3	14	12	4	10	13	1	5	2	7	9	6	8	11

[3 13 9]
[14 1 6]
[12 5 8]
[4 2 11]
[10 7]

(b) 5개의 interval-5 서브리스트

i :	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
a[i] :	3	1	5	2	7	9	6	8	4	10	13	14	12	11

(c) 정렬된 interval-5 서브리스트

쉘 정렬 (3)

- **쉘 정렬 예 (2) : interval = 3**

i :	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
a[i] :	3	1	5	2	7	9	6	8	4	10	13	14	12	11

[3 2 6 10 12]

[1 7 8 13 11]

[5 9 4 14]

(d) 3개의 interval-3 서브리스트

i :	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
a[i] :	2	1	4	3	7	5	6	8	9	10	11	14	12	13

(e) 정렬된 interval-3 서브리스트

- **쉘 정렬 예 (3) : interval = 1**

i :	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
a[i] :	1	2	3	4	5	6	7	8	9	10	11	12	13	14

(f) 최종 정렬 결과

쉘 정렬 (4)

- **ShellSort 알고리즘**

```
shellSort(a[])
    interval ← a.length;
    while (interval > 1) do  {
        interval ← 1 + interval / 3;
        for (i ← 0;  i < interval;  i ← i+1) do  {
                intervalSort(a, i, interval);
        }
    }
end ShellSort()
```

쉘 정렬 (5)

- **intervalSort 알고리즘**

```
intervalSort(a, i, interval)
            // 서브리스트를 삽입 정렬로 정렬하는 ShellSort()의 보조 함수
   j ← i + interval;
   while (j < a.length) do  {
      new ← a[j];    // 서브리스트의 새로운 원소
      k ← j;    // new보다 큰 원소는 interval만큼 오른편으로 이동
      move ← true;
      while (move) do  {
         if (a[k – interval] ≤ new) then  {
               move ← false;
         }
         else  {
               a[k] ← a[k – interval];
               k ← k – interval;
               if (k = i) then
                  move ← false;
         }
      }
      a[k] ← new;    // 이동해서 생긴 자리에 삽입
      j ← j + interval;    // 다음 서브리스트 원소의 인덱스
   }
end intervalSort()
```

쉘 정렬 (6)

- **Sorting 클래스의 메소드 멤버 구현**

```
public static void shellSort(int[] a)  {
    int interval = a.length;
    • • • • •    // ShellSort 알고리즘의 Java 코드
    intervalSort(a, i, interval);
    • • • • •    // 기타 Java 코드
}

private static void intervalSort(int[] a, int i, int interval)  {
            // 서브리스트를 삽입 정렬로 정렬하는 shellSort()의 보조 메소드
    • • • • •    // intervalSort 알고리즘의 Java 코드
}
```

기수 정렬 (1)

- 기본 개념
  - 정렬할 원소의 키값을 나타내는 숫자의 자리수(radix)를 기초로 정렬하는 기법(사전식 정렬)
  - 80칼럼 천공 카드를 정렬시키거나 도서관 목록 카드 정렬과 같은 생활 응용에 많이 이용
- 방법
  - 첫번째 패스
    - 정렬할 키의 가장 작은 자리수를 기준으로 분배 정렬
  - 두 번째 패스
    - 두 번째 작은 자리수를 기준으로 분배 정렬
    - 키 값이 같은 경우 이전 정렬에서의 순서를 그대로 유지
  - 키의 가장 큰 자리수까지 반복 수행
  - 시간 복잡도 : $O(n)$

기수 정렬 (2)

- **RadixSort 알고리즘**

```
radixSort(a[])
    n ← a.length;
    for (k ← 1;  k≤ m;  k ← k+1) do  {     // m은 digit 수, k=1은 가장 작은 자리수
        for (i ← 0;  i < n;  i ← i+1) do  {
            kd ← kth-digit(a[i]);
            enqueue(Q[kd], a[i]);     // Q[kd]에 a[i]를 삽입
        }
        p ← -1;
        for (i ← 0;  i ≤ 9;  i ≤ i+1) do  {
            while (Q[i] ≠ ∅) do  {     // Q[i]의 모든 원소를 a[]로 이동
                    p ← p+1;
                    a[p] ← dequeue(Q[i];
            }
        }
    }
end radixSort()
```

기수 정렬 (3)

- **RadixSort 알고리즘 실행 단계**
  - (a) 초기 리스트 : a = [35 81 12 67 93 46 23 26]

Q[0] :	[ ]
Q[1] :	[81]
Q[2] :	[12]
Q[3] :	[93 23]
Q[4] :	[ ]
Q[5] :	[35]
Q[6] :	[46 26]
Q[7] :	[67]
Q[8] :	[ ]
Q[9] :	[ ]

Q[0] :	[ ]
Q[1] :	[12]
Q[2] :	[23 26]
Q[3] :	[35]
Q[4] :	[46]
Q[5] :	[ ]
Q[6] :	[67]
Q[7] :	[ ]
Q[8] :	[81]
Q[9] :	[93]

(b) 첫 번째 자리수를 기초로 정렬

- 결과 : [81 12 93 23 35 46 26 67]

(c) 두 번째 자리수를 기초로 정렬

- 결과 : [12 23 26 35 46 67 81 93]

기수 정렬 (4)

- **Sorting** 클래스의 메소드 멤버 구현

```
public static void radixSort(int[] a)  {
            // 기수 정렬의 메인 메소드
   • • • • •    // RadixSort 알고리즘의 Java 코드
}
```

트리 정렬 (1)

- 기본 개념
  - 이진 정렬 트리와 중위 우선 순회 방법을 사용한 정렬 방법
- 방법
  - 배열을 이진 정렬 트리에 삽입
    - 동일한 키값의 원소 허용
    - 루트와 같은 키값은 오른쪽 서브트리에 삽입
  - 중위 우선 순회 방법으로 원소를 하나씩 검색하여 원래의 배열에 저장
  - 시간 복잡도 : $O(n\log n)$

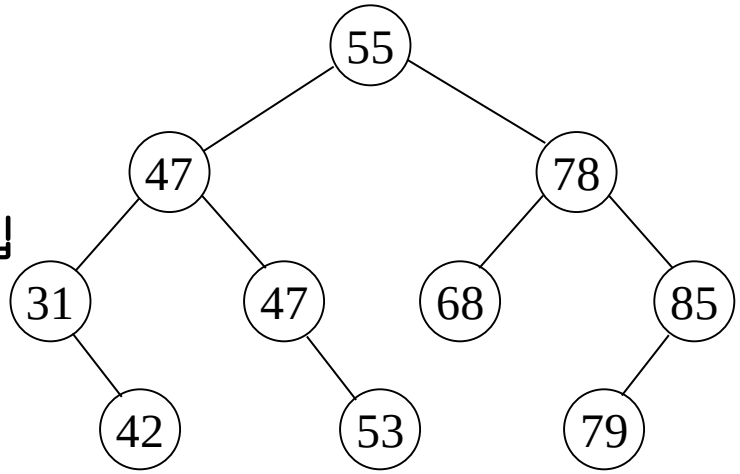
트리 정렬 (2)

- 트리 정렬 수행 예

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
a[] :	**55**	**47**	**31**	**78**	**47**	**53**	**85**	**42**	**68**	**79**

(a) 초기 배열 a[]

(b) 이진 정렬 트리 삽입



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
a[] :	**31**	**42**	**47**	**47**	**53**	**55**	**68**	**78**	**79**	**85**

(c) 정렬된 배열 a[]

트리 정렬 (3)

- **TreeSort 알고리즘**

```
treeSort(a[], n)
    for (i ← 0;  i < n;  i ←  i+1)
          insert(T, a[i]);     // T는 이진 정렬 트리
    inorder(T, a, -1);     // inorder 알고리즘
end treeSort()
```

- **inorder 알고리즘**

```
inoder(T, a[], i)
          // treeSort()의 보조 함수
    if(T = null)   then return;
    inorder(T.left, a, i);     // T의 왼쪽 서브트리
    i = i + 1;
    a[i] ← T.key;
    inorder(T.right, a, i);     // T의 오른쪽 서브트리
end inorder()
```