제 7 장 정렬(Sorting)

7.1 동기: 탐색

- 리스트: 하나 이상의 필드를 가진 레코드들의 집합
  (예: 성적레코드 – 학번, 이름, 학기, 과목코드, 학점 등)
- 키: 레코드를 식별할 수 있는 필드 (예: 학번)
- 순차탐색과 이진탐색이 주로 사용됨
- 순차탐색: 레코드 리스트를 왼편에서 오른편 또는 오른편에서 왼편으로 레코드를 검사하는 것
- 이진탐색: 레코드리스트의 중간을 반복적으로 검사하는 것

(1) 순차 탐색(sequential search)

- 키들은 순서가 없음
- 예: [0] [1] [2] [3] [4]
  8 5 15 7 12

k=5: 1을 반환

k=11: -1을 반환 (존재하지 않을 경우)

순차 탐색 알고리즘

```
Template <class T>
int SeqSearch(T*a, const int n, const T& k)
// a[0:n-1]을 왼쪽에서 오른쪽으로 탐색한다.
// a[i]의 키 값이 k 와 같은 가장 작은 i를 반환한다.
// 그런 i가 없으면 -1을 반환한다
{
   int i;
   for (i=0; i<n && a[i]!= k; i++) ;
   if (i >= n) return -1;
   return i;
}
```

- 프로그램 분석:
  최선의 경우: 1, 최악의 경우: n, 평균의 경우: (n+1)/2
  시간복잡도: O(n)

(2) 이진 탐색(binary search)

- 순서에 따라 정렬된 리스트
- 존재하지 않는 경우에는 -1을 반환
- 예:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	left	right	middle
	2	5	6	9	16	20	25			
k=5:				(0+6)/2 = 3				0	6	3
			(0+2)/2 = 1					0	2	1
	*결과: 첨자 1을 반환									
k=18:				(0+6)/2 = 3				0	6	3
					(4+6)/2 = 5			4	6	5
				(4+4)/2 = 4				4	4	4
								5	4	
	*결과: left > right 이므로 -1을 반환									

이진 탐색 알고리즘

```
    template <class T>
1   int BinarySearch(T *a, const T& k, const int n)
2    // 정렬된 배열 a[0], ..., a[n-1]에서 x를 탐색한다.
3    {
4        for (int left = 0, right = n-1; left <= right;){  // 원소가 더 있을 때까지
5            int middle = (left + right) / 2;
6            switch (comare(k, a[middle]) {
7                case '>' : left = middle + 1; break;      // k > a[middle]
8                case '<' : right = middle - 1; break;     // k < a[middle]
9                case '=' : return middle;                 // k == a[middle]
10           }   // switch의 끝
11       }       // for의 끝
12       return -1; // 발견되지 않음
13   }  // BinarySearch의 끝
```

- 분석: 최선의 경우: 1, 최악의 경우: log n
  시간복잡도: O(log n)

정렬의 정의

- 정렬: 리스트를 키의 크기 순(오름차순, 내림차순)으로 재배치하는 것
- 두 가지 방법:

  내부 정렬: 정렬할 리스트 전체를 주기억장치에 저장한 후 정렬하는 방법

  외부 정렬: " 저장할 수 없을 때 정렬하는 방법

7.2 삽입정렬(insertion sort)

- 기본 단계: 삽입 원소를 기존의 정렬된 부분 리스트 속에 끼워 넣기

예:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	
−∞	1	2	4	6	8	③	3을 끼워넣기

↑ (3 < 8 true)

8

↑ (3 < 6 true)

6

↑ (3 < 4 true)

4

↑ (3 < 2 false)

3

결과: −∞ 1 2 3 4 6 8

기본 단계 insert 함수

```
template <class T>
void Insert(const T& e, T *a, int i)
{// e를 정렬된 리스트 a[1:i]에 삽입하여 결과
  // 리스트 a[1: i+1]도 정렬되게 한다.
  // 배열 a는 적어도 i+2 원소를 포함할 공간을 가지고 있어야 한다.
  //  a[0] = -∞; 가 저장되어 있음.
    while(e <a[i])
    {
        a[i+1] = a[i];
        i--;
    }
    a[i+1] = e;
}
```

- 최악의 경우 i+1 번 비교

삽입정렬

- 삽입 원소의 위치를 옮기면서 기본 단계 insert를 반복적으로 적용한다.

예:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	
–	4	8	1	2	6	3	
		↑					(삽입 원소의 위치: 2)
–∞	4	(8)					
			↑				(삽입원소의 위치: 3)
–∞	(1)	4	8				
				↑			(삽입원소의 위치: 4)
–∞	1	(2)	4	8			
					↑		(삽입원소의 위치: 5)
–∞	1	2	4	(6)	8		
						↑	(삽입원소의 위치: 6)
–∞	1	2	(3)	4	6	8	

삽입정렬 함수

```
Template <class T>
void InsertionSort(T *a, const int n) // a[1:n]을  오름차순으로 정렬
{
   for(int j=2; j<=n; j++) {
      T temp = a[j];
      insert(temp, a, j-1);      // a[j]가 삽입 원소이다.
   }
}
```

- 분석: $\sum_{i=1}^{n-1} (i + 1) = n^2$, 따라서 $O(n^2)$
- 최선의 경우: 오름차순으로 정렬된 상태
- 최악의 경우: 역순(내림차순)으로 정렬된 상대

7.3 퀵 정렬(quick sort)

- 평균 수행시간이 가장 짧다.

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] left right

(26 5 37 1 61 11 59 15 48 19) 1 10

중추키 ↑ ↑

i j

19 37

15 61

↑ ↑

j i

(11 5 19 1 15) 26 (59 61 48 37) 1 5

반복 적용

(1 5) 11 (19 15) 26 1 2

1 5

15 19 4 5

(59 61 48 37) 7 10

(48 37) 59 (61) 7 8

37 48

61 10 10

1 5 11 15 19 26 37 48 59 61 ➔ 결과

퀵 정렬의 예제

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

그림 7.1 : 함수 QuickSort 예제

퀵 정렬 함수

```
void QuickSort(T* a, const int left, const int right) {
// 중추키(pivot)는 a[left]로 선정한다.
// a[left] ≤ a[right+1] 라고 가정한다.
  if(left < right)  {
        int i = left,
         j = right + 1,
         pivot = a[left];
      do{
          do { i++; } while( a[i] < pivot);
          do { j--; } while( a[j] > pivot);
          if (i<j) swap(a[i], a[j]);
      }while (i<j);
      swap(a[left], a[j]);

      QuickSort(a, left, j-1);
      QuickSort(a, j+1, right);
      }
}
```

퀵 정렬의 분석

- 처음 함수 호출: QuickSort(a, 1, n)
- 최악의 경우 – 리스트가 이미 정렬된 경우, $n + (n-1) + \ldots 1$,
  따라서 $O(n^2)$.

  최선의 경우 – 리스트가 항상 반으로 나누어 질 때

  $T(n) \le cn + 2T(n/2)$ (어떤 상수 c에 대해)

  $\le cn + 2(cn/2 + 2T(n/4))$

  $\le 2cn + 4T(n/4)$

  …

  $\le cn \log n + nT(1) = O(n \log n)$

  평균의 경우 – $O(n \log n)$
- 변형: 중추 키를 선택할 때 (첫번째, 가운데, 마지막) 위치에 있는 값들 중에서 중간 크기의 값을 사용하면 약간 개선시킬 수 있다.

  → 중간값 규칙