9 장. 정렬

Jinseog Kim

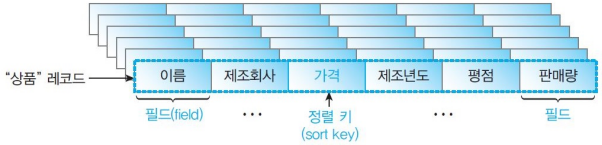
학습 내용

1. 정렬이란?
2. 선택 정렬
3. 삽입 정렬
4. 버블 정렬
5. 함수 포인터를 사용한 정렬
6. 병합 정렬
7. 퀵 정렬
8. 기수 정렬

정렬이란?

- 객체들을 순서대로 나열하는 작업: 가장 기본적이고 중요한 알고리즘의 하나

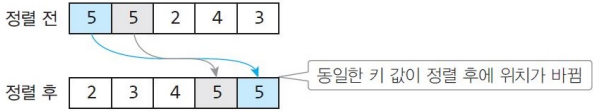
- 레코드와 필드, 키의 개념



- 정렬이란 레코드를 키 (key) 의 순서로 재배열하는 작업

정렬 관련 용어

- 모든 경우에 대해 최적인 정렬 알고리즘은 없음
  - 해당 응용 분야에 적합한 정렬 방법 사용해야 함
  - 레코드의 수 / 레코드의 크기 / key 의 특성 (문자, 정수, 실수 등)
- 정렬의 종류
  - 단순하지만 비효율적인 방법: 삽입, 선택, 버블 정렬 등
  - 복잡하지만 효율적인 방법: 퀵, 힙, 병합, 기수 정렬 등
- 알고리즘의 성질
  - 안정성 (stable sort)



  - 제자리 정렬 (in-place sort)

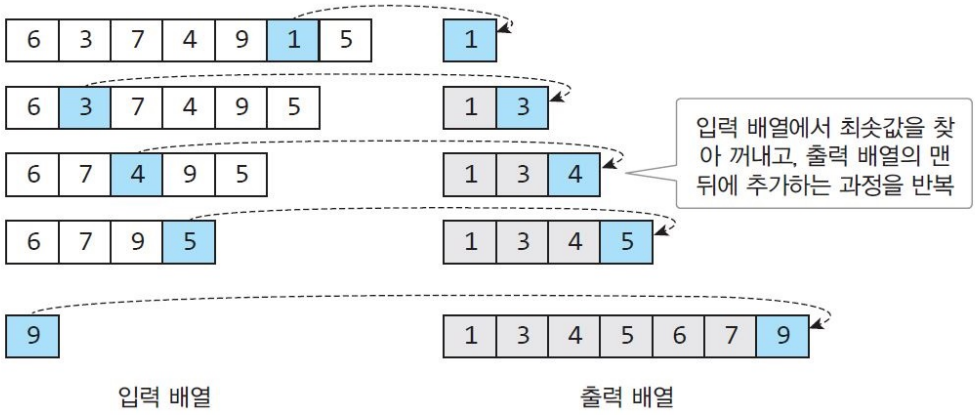
정렬 알고리즘의 종류

1. $O(n^2)$
   - 선택정렬 (selection sort)
   - 삽입정렬 (insertion sort)
   - 버블정렬 (bubble sort)
2. 병합정렬 (Merge Sort)
3. 셸정렬 (Shell sort)
4. 기수정렬 (Radix sort)
5. 퀵정렬 (Quick sort)
6. 힙정렬 (Heap sort)

dongguk UNIVERSITY WISE

선택 정렬

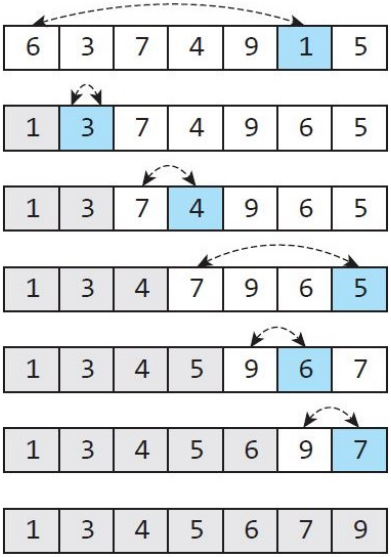
- 가장 작은 숫자부터 하나씩 찾아 순서대로 저장



- 제자리 정렬이 아님

제자리 정렬로 개선?

- 교환을 이용



선택 정렬 알고리즘

```
1  selection_sort(A[], n)
2     for i=0 to n-2 :          // n-1 번 반복
3         least <- i              // 최소 요소의 인덱스
4         for j = i+1 to n-1 :
5           if A[j] < A[least] :
6             least <- j
7         A[i] <-> A[least]     // 교환
```

선택 정렬 알고리즘 구현

```
1  #define SWAP(x,y,t) ((t)=(x),(x)=(y),(y)=(t))
2
3  void selection_sort(int A[], int n)
4  {
5      int tmp;                          // SWAP() 을 위한 임시변수
6      for (int i = 0; i<n - 1; i++) {
7          int least = i;                // 최소 요소의 인덱스
8          for (int j = i + 1; j<n; j++) // A[i+1~n-1] 을 검사함
9              if (A[j]<A[least])        // 최소 요소보다 작으면
10                 least = j;            // 최소 요소 갱신
11         SWAP(A[i], A[least], tmp);    // A[i] 와 A[min] 교환
12
13         print_step(A, n, i+1);        // 중간 과정 출력용 문장
14     }
15 }
```

선택 정렬 알고리즘 복잡도 분석

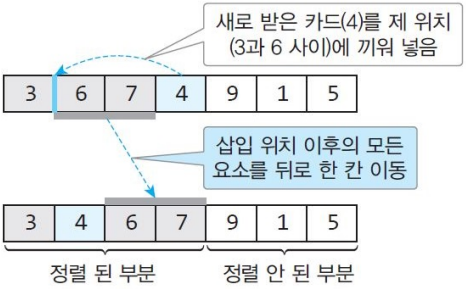
복잡도

- 전체 시간 복잡도: $O(n^2)$

특징

- 알고리즘이 매우 간단
- 효율적인 알고리즘은 아님 ($O(n^2)$)
- 안정성을 만족하지 않음
- 제자리 정렬
- 자료 이동 횟수가 미리 결정됨

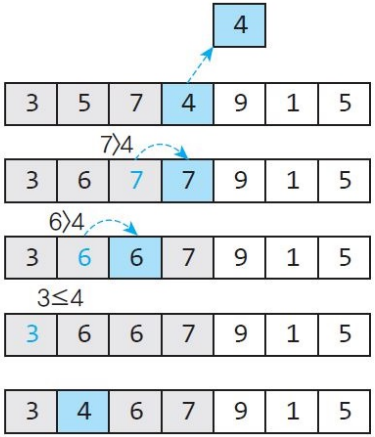
- 카드를 정렬하는 방법과 유사



삽입 정렬

- 삽입 정렬의 전체 처리 과정

삽입 과정


4
3 5 7 4 9 1 5
삽입할 요소를 일단 복사해 둠
7〉4
3 6 7 7 9 1 5
정렬된 부부의 맨 뒤 요소부터 비교
7〉4이므로 7을 한 칸 뒤로 복사
6〉4
3 6 6 7 9 1 5
6〉4이므로 6을 한 칸 뒤로 복사
3≤4
3 6 6 7 9 1 5
3≤4이므로 삽입 위치를 찾음
3 4 6 7 9 1 5
4를 삽입 위치에 복사: 삽입 완료

삽입 정렬 알고리즘

```
1  insertion_sort(A[], n)
2      for i = 1 to n-1 :
3          key <- A[i]                 // 삽입할 요소
4          for j = i-1 to 0 :          // 뒤에서 부터
5              if A[j] > key :         // A[j] 가 더 크면
6                  A[j + 1] = A[j]     // A[j] 를 뒤로 한 칸 이동
7              else : break            // 제 위치를 찾음. A[j] 의 다음 위치
8          A[j + 1] <- key             // 제 위치에 복사
```

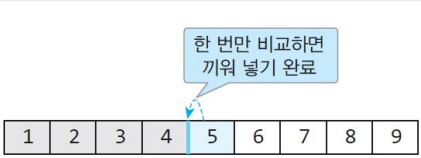
삽입 정렬 알고리즘 구현

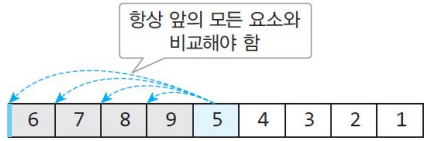
```
1  void insertion_sort(int A[], int n)
2  {
3      for (int i = 1; i < n; i++) {
4          int key = A[i];                  // 미리 A[i] 를 저장해 둠
5          int j;
6          for (j = i - 1; j >= 0; j--) {  // i-1 부터 0 까지 하나씩 줄이면서
7              if (A[j] > key)              // A[j] 가 더 크면
8                  A[j + 1] = A[j];         // A[j] 를 뒤로 한 칸 이동
9              else break;                  // 제 위치를 찾음. A[j] 의 다음 위치
10         }
11         A[j + 1] = key;
12         print_step(A, n, i);
13     }
14 }
```

삽입 정렬 알고리즘 복잡도 분석

복잡도

- 복잡도는 입력의 구성에 영향을 받음: 최선의 경우 $O(n)$, 최악의 경우는 $O(n^2)$



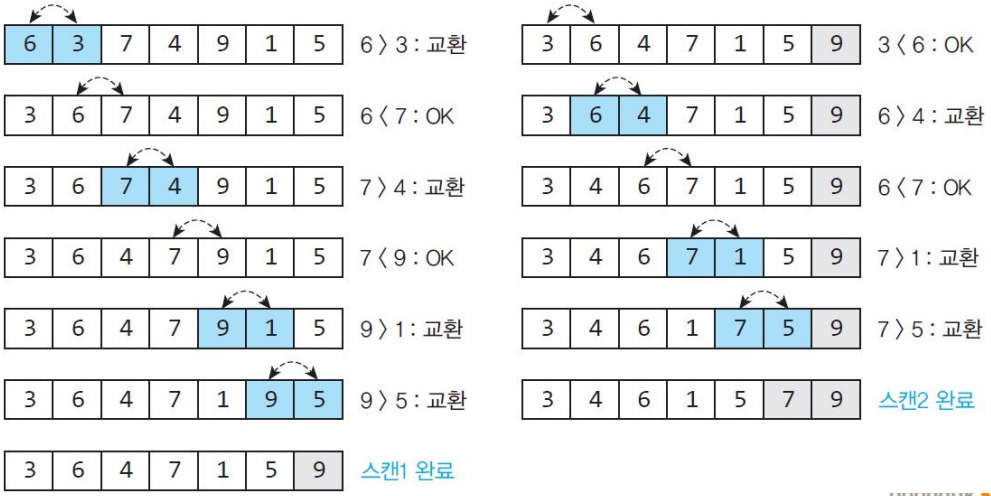


특징

- 많은 레코드의 이동을 포함하므로 레코드 크기가 클수록 불리
- 안정성을 만족
- 제자리 정렬
- 레코드가 대부분 이미 정렬되어 있다면 효과적으로 사용될 수 있음 $\Longrightarrow$ 셸 정렬 등에서 활용

버블 정렬

- 인접한 레코드를 비교해 크기 순서가 맞지 않으면 서로 교환하는 방법



버블 정렬 알고리즘

```
1  void bubble_sort(int A[], int n){
2      int tmp;
3      for (int end = n - 1; end > 0; end--) { // 정렬되지 않은 부분의 마지막 위치
4          int bChanged = 0;
5          for (int j = 0; j < end; j++)       // 한 번의 스캔: 0 부터 end-1 까지 진행
6              if (A[j] > A[j + 1]) {          // 인접한 요소가 역전되어 있으면
7                  SWAP(A[j], A[j + 1], tmp);  // 교환
8                  bChanged = 1;               // 교환이 발생함
9              }
10         if (!bChanged) break;               // 교환이 한 번도 없었으면 종료
11         print_step(A, n, n - end);
12     }
13 }
```

버블 정렬 알고리즘 복잡도 분석

복잡도

- 복잡도는 입력의 구성에 영향을 받음
- 최선: 정렬된 리스트 $O(n)$
- 최악: 역순으로 정렬된 리스트 $O(n^2)$

특징

- 많은 레코드의 이동을 포함하므로 레코드 크기가 클수록 불리
- 안정성을 만족
- 제자리 정렬
- 입력 자료가 어느 정도 정렬되어 있다면 효과적으로 사용됨

dongguk UNIVERSITY WISE

함수 포인터를 사용한 정렬

- 비교 함수를 만들어 매개변수로 전달하는 방법: (예: 삽입 정렬)

```
1  int ascend(int x, int y) { return y - x; }  // 오름차순 비교함수
2  int descend(int x, int y) { return x - y; } // 내림차순 비교함수
3  void insertion_sort_fn(int A[], int n, int(*f)(int, int)){
4      for (int i = 1; i < n; i++) {
5          int key = A[i];                  // 미리 A[i] 를 저장해 둠
6          int j;
7          for (j = i - 1; j >= 0; j--) {  // i-1 부터 0 까지 하나씩 줄이면서
8              if (f(A[j], key) < 0)       // A[j] 가 더 크면
9                  A[j + 1] = A[j];        // A[j] 를 뒤로 한 칸 이동
10             else break;                 // 제자리를 찾음. A[j] 다음 위치.
11         }
12         A[j + 1] = key;                 // A[i] 는 j+1 에 있어야 함. 제 위치에 복사
13         print_step(A, n, i);            // 중간 과정 출력
14     }
15 }
```

Lab: 함수 포인터를 이용한 구조체 정렬

- 2 차원 좌표들로 이루어진 구조체를 다양한 기준으로 정렬
  - 기준 1: x 의 오름차순 정렬
  - 기준 2: y 의 내림차순 정렬
  - 기준 3: 크기 ($\sqrt{x^2 + y^2}$) 의 오름차순

```
1  // 배열요소의 자료형 정의
2  typedef struct {
3    int x, y;
4  }Point2D;
5  # define Element Point2D;
```

Lab: 함수 포인터를 이용한 구조체 정렬

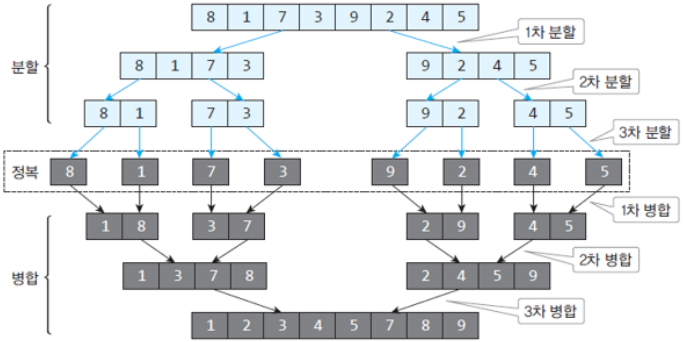
```c
int x_ascend(Element a, Element b) { return (b.x - a.x); }  // x 오름차순
int y_descend(Element a, Element b) { return (a.y - b.y); } // y 내림차순
int z_ascend(Element a, Element b) {    // 크기의 오름차순
  return ((b.x*b.x + b.y*b.y) - (a.x*a.x + a.y*a.y));
}
void insertion_sort_fn(Element A[], int n, int(*f)(Element, Element)){
    for (int i = 1; i < n; i++) {
        Element key = A[i];
        int j;
        for (j = i - 1; j >= 0; j--) {
            if (f(A[j], key) < 0)
                A[j + 1] = A[j];
            else break;
        }
        A[j + 1] = key;
    }
}
```

병합 정렬 (merge sort)

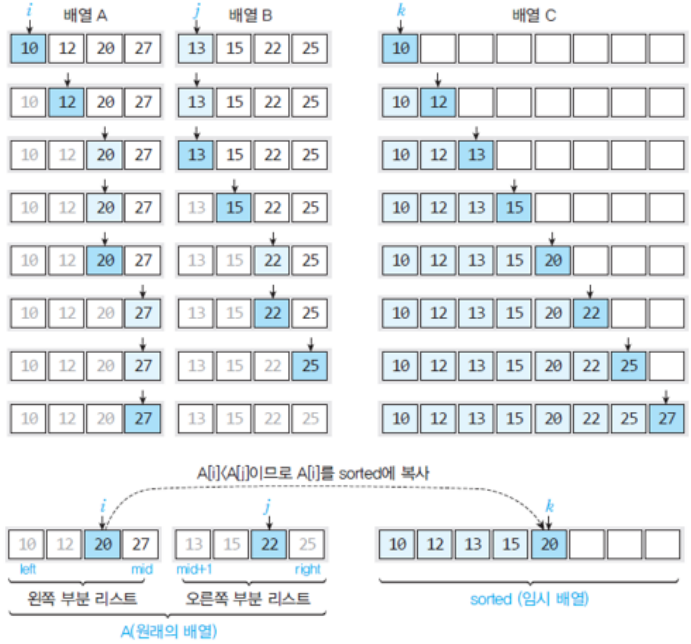
- 각각 정렬된 두 개의 부분 배열을 병합하여 하나의 정렬된 배열을 만드는 방법
- 분할 정복 (divide and conquer) 전략
  - 분할 (Divide)
    입력 배열을 같은 크기의 2 개의 부분 배열로 분할
  - 정복 (Conquer)
    부분 배열을 정렬. 부분 배열의 크기가 충분히 작지 않으면 순환 호출을 이용하여 다시 분할 정복 기법을 적용
  - 병합 (Combine)
    정렬된 부분 배열을 하나의 배열에 병합

병합 정렬 알고리즘

```
1  merge_sort(A[], left, right)
2      if left < right :
3          mid <- (left + right) / 2   // 리스트의 균등 분할
4          merge_sort(A, left, mid)        // 부분 리스트 정렬
5          merge_sort(A, mid + 1, right)   // 부분 리스트 정렬
6          merge(A, left, mid, right)      // 병합
```



병합 정렬 알고리즘


i
배열 A
10 12 20 27
j
배열 B
13 15 22 25
k
배열 C
A[i]<A[j]이므로 A[i]를 sorted에 복사
left
mid
mid+1
right
왼쪽 부분 리스트
오른쪽 부분 리스트
A(원래의 배열)
sorted (임시 배열)

병합 정렬 알고리즘

```
1  merge(A[], left, mid, right)
2      i <- left        // 왼쪽 부분 배열 A[left~mid] 의 시작 위치
3      j <- mid+1       // 오른쪽 부분 배열 A[mid+1~right] 의 시작 위치
4      k <- left        // 임시 배열 B[left~right] 의 시작 위치
5      while i <= mid and j <= right :
6          if A[i] <= A[j] :
7              sorted[k] <- A[i]
8              i <- i + 1
9          else :
10             sorted[k] = A[j]
11             j <- j + 1
12         k <- k + 1
13     if i <= mid :
14        sorted[k~right] <- A[i~min]  // 왼쪽에 남은 레코드 모두 복사
15     else :
16        sorted[k~right] <- A[j~right]
17     A[left ~ right] <- sorted[left ~ right]
```

병합 정렬 알고리즘 복잡도 분석

복잡도

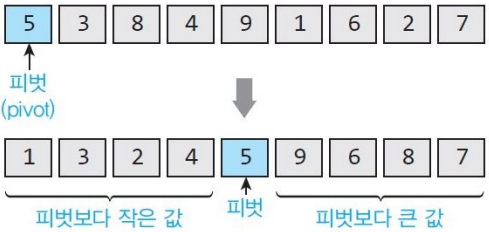
- 비교 연산
  - 크기 n 인 리스트를 균등 분배하므로 $\log(n)$ 개의 패스
  - 각 패스에서 레코드 n 개를 비교: n 번 비교
- 이동 횟수
  - 각 패스에서 $2n$ 번 이동
  - 전체 이동: $2n \log(n)$
- 시간 복잡도: $O(n \log(n))$

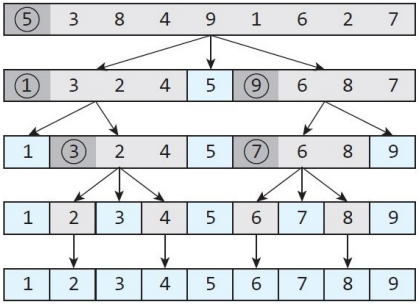
특징

- 효율적이고 입력의 구성과 상관없이 동일한 시간에 정렬
- 안정성을 만족
- 제자리 정렬이 아님

퀵 정렬 (quick sort)

- 병합 정렬과 같은 분할 정복 전략
  - 병합 정렬: **위치**에 따라 나누는 전략
  - 퀵정렬: **값**에 따라 나누는 전략
  - 분할의 기준이 되는 값을 **피벗 (pivot)** 이라고 함

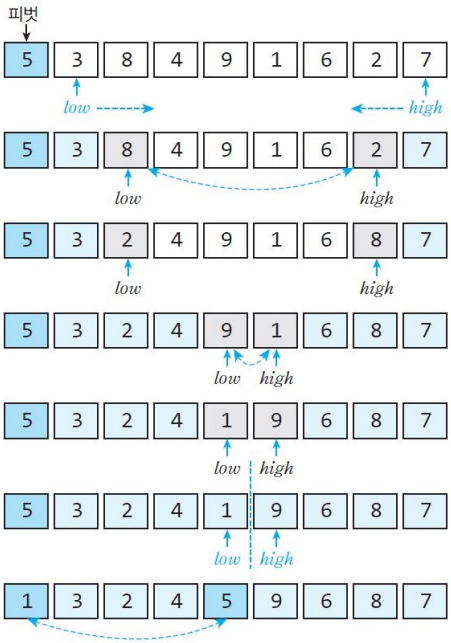




퀵 정렬 알고리즘

```
1  quick_sort(A[], left, right)
2      if left < right  :                  // 요소가 2 개 이상인 경우
3          q <- partition(A, left, right) // 피벗을 중심으로 좌우로 분할
4          quick_sort(A, left, q - 1)     // 왼쪽 부분 정렬
5          quick_sort(A, q + 1, right)    // 오른쪽 부분 정렬
```

퀵 정렬의 분할 알고리즘



5를 피벗으로 선택
low ← left+1
high ← right

low를 피벗보다 큰 요소까지 이동
high를 피벗보다 작은 요소까지 이동

low와 high의 요소 교체

다시 진행
low를 피벗보다 큰 요소까지 이동
high를 피벗보다 작은 요소까지 이동

low와 high의 요소 교체

다시 진행
low와 high가 역전됨 → 종료

피벗과 high위치의 요소 교환

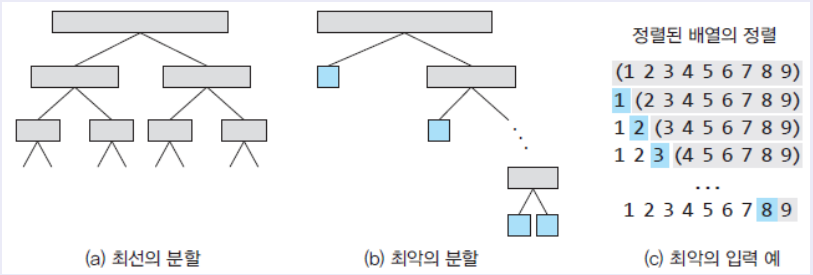
퀵 정렬의 분할 알고리즘

```
1  partition(A[], left, right)
2      pivot <- A[left]                  // 피벗
3      low <- left + 1                   // 왼쪽 배열을 위한 인덱스
4      high <- right                     // 오른쪽 배열을 위한 인덱스
5      while low <= high :
6          while low <= high && A[low] <= pivot : low <- low + 1
7          while low <= high && A[high] > pivot : high <- high - 1
8          if low < high :                  // 인덱스가 역전되지 않았으면
9              A[low] <-> A[high]           // 조건에 맞지 않는 두 요소 교환
10     A[left] <-> A[high]           // high 와 피벗 교환
11     return high                   // 피벗의 인덱스 반환
```

퀵 정렬 알고리즘 복잡도 분석

복잡도

- 최선의 경우: $O(n \log n)$, 최악의 경우: $O(n^2)$



특징

- 효율적인 알고리즘
  - 불필요한 데이터의 이동을 줄이고
  - 먼 거리의 데이터를 교환
  - 한번 결정된 피벗들이 추후 연산에서 제외
- 피벗 선택 개선 방법: median of three

퀵 정렬 라이브러리 함수

- C 언어 기본 라이브러리의 퀵 정렬 함수 qsort()

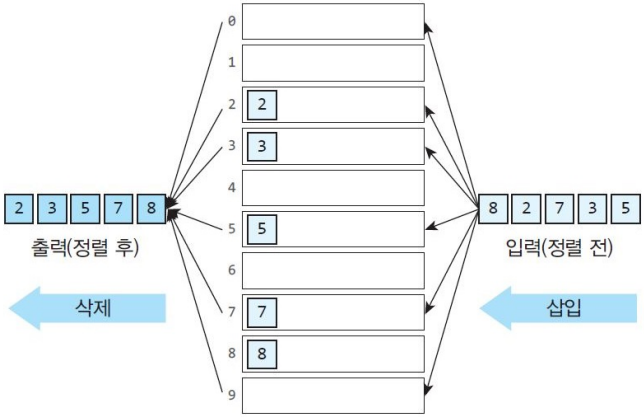
```
void qsort(
    void *base, //배열의 시작주소
    int num, //배열요소의 수
    int width, // 배열요소 하나의 크기 (byte)
    int (*compare)(void *, void *) //비교함수 포인터
);
```

퀵 정렬 라이브러리 함수

```c
# include <stdlib.h>
int compare(const void *arg1, const void *arg2){
    if (*(double *)arg1 > *(double *)arg2) return 1;
    else if (*(double *)arg1 < *(double *)arg2) return -1;
    else return 0;
}
void main() {
    double list[9] = { 2.1, 0.9, 1.6, 3.8, 1.2, 4.4, 6.2, 9.1, 7.7 };
    printf(" 정렬 전: ");
    for (int i=0; i < 9; i++)
        printf("%4.1f ", list[i]);
    qsort((void *)list, 9, sizeof(double), compare);
    printf("\n정렬 후: ");
    for (int i=0; i<9; i++)
        printf("%4.1f ", list[i]);
    printf("\n");
}
```

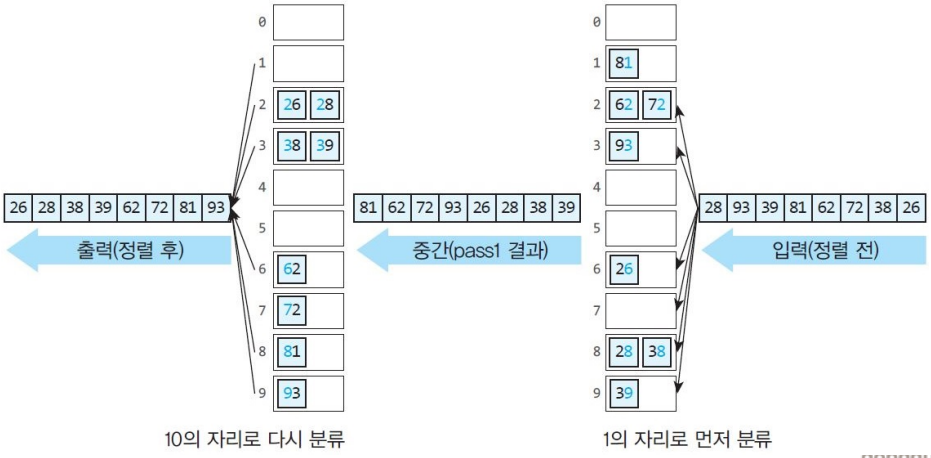
기수 정렬 (radix sort)

- 요소를 비교하지 않고 정렬하는 독특한 기법
  - 비교 기반 정렬의 이론적인 시간 복잡도 하한선 돌파
- 원리: 한 자릿수 숫자의 정렬
  - 10 개의 버킷 (queue) 사용



기수 정렬

- 여러 자릿수 정렬
  - 먼저 낮은 자릿수로 정렬한 다음 높은 자릿수로 정렬



기수 정렬 알고리즘

```
1  radix_sort(A[], n)
2      factor <- 1                    // 1 의 자리부터 시작
3      for d=0 to DIGITS - 1 :        // 모든 자릿수에 대하여
4          for i = 0 to n - 1 :       // 배열의 모든 요소를
5              id <- (A[i] / factor) % BUCKETS // 해당버킷을 찾아
6              Q[id].enqueue (A[i])           // 그 베킷에 삽입
7
8          i <- 0                             // 버킷에서 꺼내는 작업시작
9          for b=0 to BUCKETS -  1 :          // 모든 버킷에서
10             while !Q[b].is_empty(A[i]) : // 모든 요소를 순서데로 꺼내
11                 A[i] <- Q[b].dequeue()   // 입력배열로 다시 저장
12                 i <- i + 1
13
14         factor = factor* BUCKETS           // 다음 자릿수로 올라감
```

기수 정렬 알고리즘 분석

- n 개의 레코드, d 개의 자릿수로 이루어진 키를 기수 정렬
  - 메인 루프는 자릿수 d 번 반복
  - 큐에 n 개 레코드 입력 수행
  - 시간 복잡도: $O(dn)$

    대부분 d<10 이하 –> 거의 선형 시간
- 기수 정렬의 단점
  - 정렬할 수 있는 레코드의 타입 한정

    실수, 한글, 한자 등은 정렬 불가
    키가 동일한 길이를 가지는 숫자나 단순 문자이어야만 함
  - 비교 기반의 정렬 방법들은 모든 종류의 키 형태에 적용 가능

정렬 방법의 성능 비교

알고리즘	Best case	Average	Worst
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix	$O(dn)$	$O(dn)$	$O(dn)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

힙 정렬 (Heap Sort)

- **완전 이진 트리** (Complete Binary Tree) 를 이용한 정렬 알고리즘
- **최대 힙 (Max Heap)** 또는 **최소 힙 (Min Heap)** 을 구성하여 데이터를 정렬

힙 정렬의 주요 단계:

1. **힙 구성:** 주어진 데이터를 힙 구조로 변환
2. **정렬 과정:** 힙에서 루트 노드를 제거하고, 남은 노드들로 다시 힙을 구성 (이 과정을 반복)

힙 정렬 알고리즘 분석

복잡도

- **시간 복잡도:** 평균 및 최악의 경우 모두 $O(n \log n)$
- **공간 복잡도:** $O(1)$

특징

- 힙 정렬은 병합 정렬 (Merge Sort) 과 퀵 정렬 (Quick Sort) 만큼 빠르며
- 특히 **우선순위 큐** (Priority Queue) 와 같은 데이터 구조에서 유용하게 사용

dongguk UNIVERSITY WISE

연습문제