연결 구조

신속한 삽입과 삭제를 허용하는 순서 리스트를 유지해야 할 경우에는 연결 자료 구조를 사용해야 한다.

1. 순서 배열의 유지

정렬된 배열에 새 원소를 삽입하는 것은 어렵다.
새 원소가 올바른 위치에 배치되도록 하기 위해 그 보다 큰 모든 원소들을 모두 이동시켜야 한다.

insert()

```
int[] insert(int[] a, int n, int x){
1       int i = 0;
2       while(i<n && a[i]<=x)
3               i++;
4       System.arraycopy(a, i, a, i+1 ,n-1);
5       a[i]=x;
}
```

배열 a[], 배열 안에 정렬되어 있는 원소의 수 n, 이들 사이에 새로 삽입할 원소 x
선조건 : 배열의 앞에 n원소가 오름차순이고, 배열이 최소한 하나의 추가적인 원소를 위한 공간을 가지고 있음
후조건 : 배열이 아직 오름차순을 유지하고 있고, x가 성공적으로 삽입되었음을 명세

라인 1-3의 코드는 배열에서 x가 삽입될 올바른 위치 탐색 = 'a[i] > x' 인 가장 작은 인덱스 i

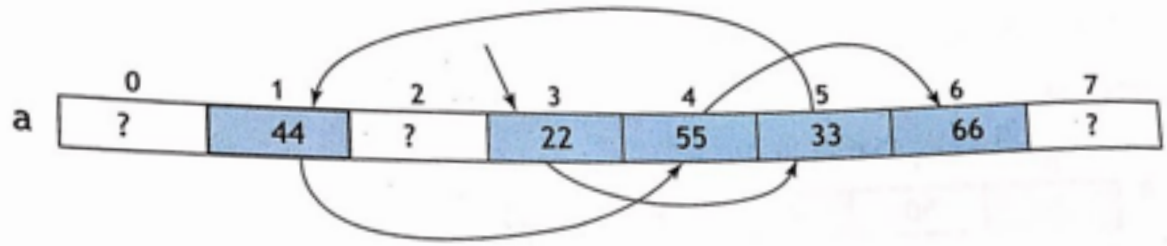
올바른 위치 i가 발견된 후, 라인 4에서 x보다 큰 원소를 한 자리씩 오른쪽으로 이동시킴
System클래스의 static 메소드 [**arraycopy()**]- 배열간에 또는 단일 배열 안에서 원소들을 복사하는 가장 효율적인 방법 제공!
인자는 순서대로 (1)소스배열, (2)소스 배열로부터 복사될 첫 번째 원소의 인덱스, (3)목표 배열, (4)목표 배열에서 첫 번째 원소가 복사될 인덱스, (5)복사될 원소의 수

insert() 메소드는 많은 데이터를 이동시킬 수 있다.
평균적으로 n개 원소의 정렬된 배열에 대한 삽입은 n/2개의 원소를 이동시킨다.
그러므로, 이는 O(n) 연산에 해당된다.

원소의 삭제는 삽입 과정의 반대이다.
이 또한 평균적으로 n/2개의 원소를 이동시켜야 하기 때문에 O(n)연산에 해당한다.

2. 간접 참조

동적인 순서를 갖는 배열에서 원소의 실제 위치를 추적하기 위한 보조 인덱스 배열 사용
코드를 좀 더 복잡하게 만들지만 원소를 이동시킬 필요성 제거
원소가 배열에서 임의에 위치에 저장되도록 허용하고, 순서대로 접근하기 위한 보조 인덱스 배열 사용



[그림 4.3] 배열 원소의 순서의 참조

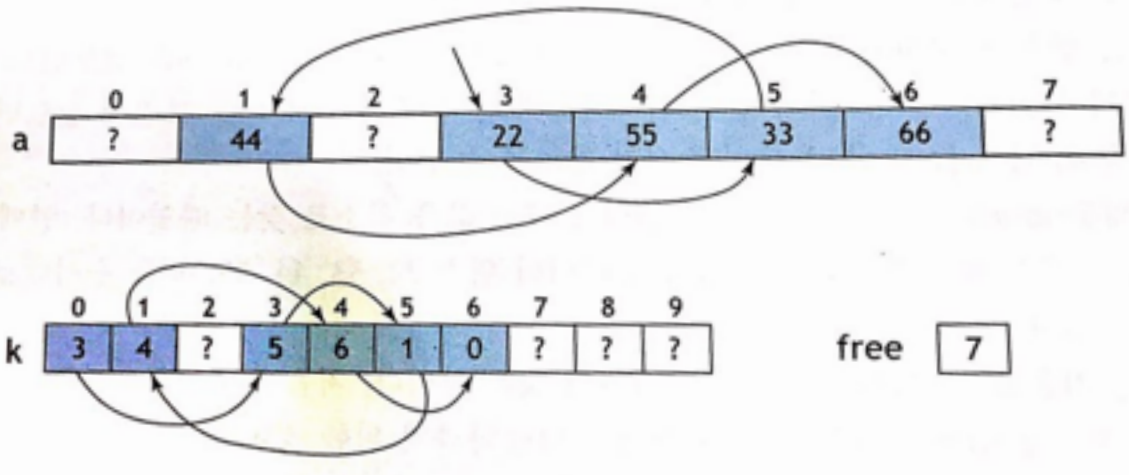
위와 같이 원소는 배열 안 임의에 위치에 저장되어 있으며, 순서는 보조 메커니즘에 의해 결정된다.
각 원소는 번호가 지정된 컴포넌트에 저장된다.

22는 3에, 33은 5에, 44는 1에 저장되어 있다. 따라서 (3,5,1)을 보관해 놓으면 원소들을 순서대로 접근할 수 있다.

-

인덱스 배열(index array) : 다른 배열의 인덱스 값을 원소로 갖는 배열

인덱스 번호 3,5,1,4,6을 인덱스 배열에 저장해 두었다가 데이터 원소에 순서대로 접근하는데 사용
기존보다 개선된 것이지만 최선은 아니다.
이는 배열에서 삽입과 삭제를 단순화 하기 위한 것이었다. 그러나, 이를 인덱스 배열로 인가한것이다.



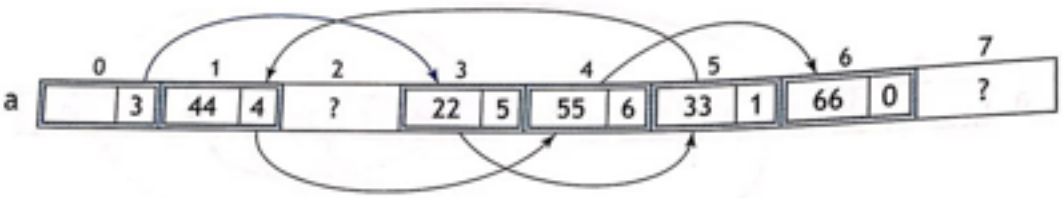
[그림 4.5] 인덱스 배열의 사용

이보다 좋은 방법은 a[]와 동일한 배열 위치를 인덱스 배열 k[]에서 사용하도록 하는 것이다.[그림4.5]
[그림 4.5]를 보면 k[]는 a[]의 원소를 추적한다.
시작 인덱스 3은 k[0]에 저장된다. 이것이 인덱스 체인을 시작시킨다.
k[0] = 3, k[3] = 5, k[5] = 1, k[1] = 4, k[4] = 6, k[6] = 0
인덱스 0은 끝을 나타낸다.

[그림 4.5]에 추가한 변수 free는 인덱스 배열k[]와 데이터 배열 a[] 의 자유 위치를 저장한다.
값 7은 k[7],a[7]이 다음에 사용될 장소임을 의미한다.

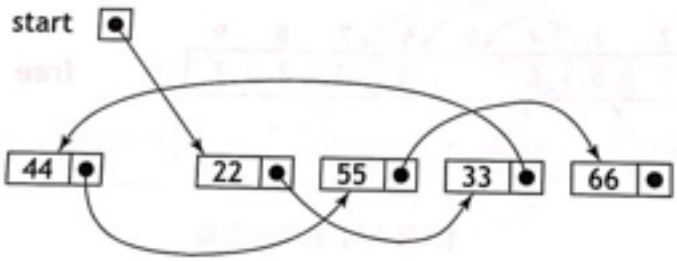
3. 연결 노드

[그림4.5] 처럼 k[] 배열을 별도로 유지할 필요는 없다.



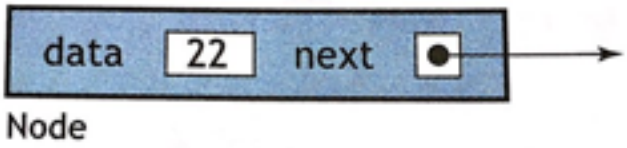
[그림 4.7] 인덱스를 해당 원소와 함께 동일 배열에 저장

[그림4.7]과 같이 데이터원소쌍의 단일 배열로 합칠 수 있다.



[그림 4.8] 원소에 그 참조를 위한 객체의 사용

Java에서는 객체가 그 주소에 의해 직접 접근된다.
객체의 참조가 의미하는 것이 객체가 메모리에 저장되는 주소이다.
그러므로 '주소'의 의미를 배열 인덱스가 아닌 메모리주소(=객체참조)로 번역하면 [그림4.8]에 보인 구조와 같이 단순화시킬 수 있다.
이제 배열 a[] 대신에 단일 start 참조만을 유지하면 된다.



[그림 4.9] Node 객체

Node 클래스는 자기-참조(self-referential) 형태이다.
즉, next필드가 타입 Node로 선언되어 있다.
각 Node객체는 Node 객체에 대한 참조를 갖는 필드를 포함한다.

[그림4.9]는 전형적인 Node 객체이다.
데이터 필드는 정수 22이고, next필드는 다른 Node객체에 대한 참조를 포함하고 있다.

Java에서는 참조 변수가 객체를 가리키거나 null을 가진다.
null값은 이 변수가 아무런 객체도 참조하지 않음을 의미한다.
null 참조 변수에 저장되는 메모리 변수는 0x0(16진수값0)이며, 아무런 객체도 저장되지 않는다.

-

next 필드가 null인 경우, 이유는 생성자가 이를 초기화시키지 않았기 때문이다.
Java에서 객체 참조 형태의 클래스 필드(타입이 클래스이거나 인터페이스)는 생성자에 의해 어떤 기존 객체로 초기화되지 않는 한 자동적으로 null로 초기화된다.

-

외부 접근이 가능한 유일한 노드는 첫 번째 노드뿐이다. 그 이름은 start이다.
그러므로, 다다음 노드의 필드를 참조하기 위해 start.next.next라는 표현을 사용하여야 한다.

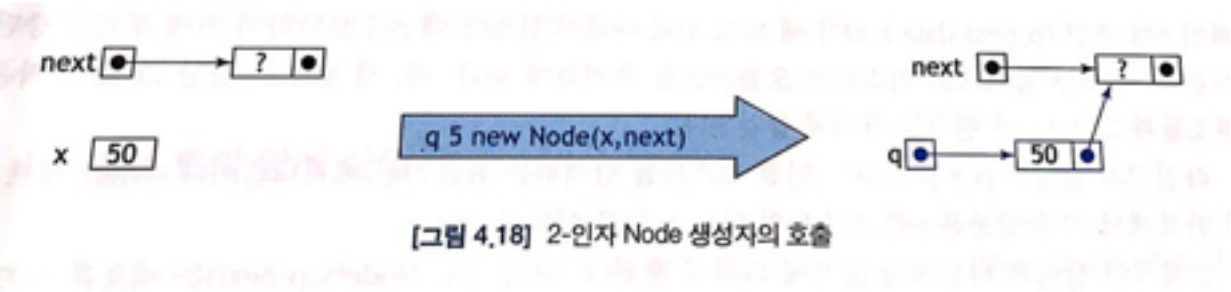
-

Node 참조를 표현하기 위해 화살표를 사용해 연결 리스트를 그리는 이유는
화살표 대신 실제 메모리 주소를 보이면 어느 노드가 어느 노드를 참조하는지를 확인하는데 많은 노력이 필요하다.
이러한 메모리 주소 값은 실행-시간에 결정되며, 다른 컴퓨터에서는 상이한 값을 갖게 되고,
동일 컴퓨터에서도 실행 시점에 따라 상이한 값을 갖게 될 수도 있다.

-

4. 연결 리스트에 대한 원소 삽입

next는 Node 객체에 대한 참조이고 x는 값이 50인 int이다.
이 2개 인자를 생성자로 전달하면 50을 포함하고, next 필드가 인자로 주어진 next포인터가 가리키던 동일 객체를 가리키는 Node 객체가 생성된다. 그리고 나서, 생성자는 새 Node 객체에 대한 참조를 q에 할당해서 리턴한다.



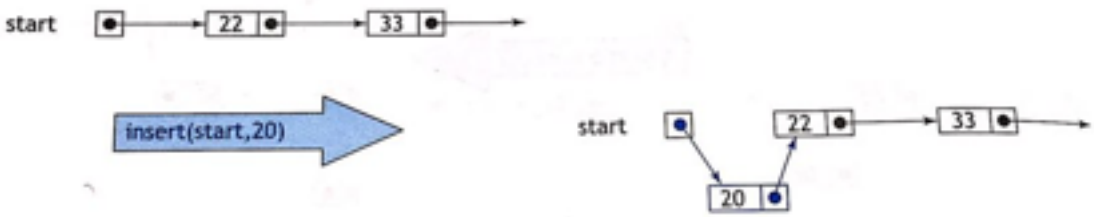
[그림 4.18] 2-인자 Node 생성자의 호출

삽입은 (1) 새로운 노드 앞에 놓일 리스트 노트 p 발견
(2) 새로운 노드를 생성해 부착하는 두 단계로 진행된다.

```
1  void insert(Node start, int x) {
2    // PRECONDITIONS: the list is in ascending order, and x > start.data;
3    // POSTCONDITIONS: the list is in ascending order, and it contains x;
4    Node p = start;
5    while (p.next != null) {
6      if (p.next.data > x) break;
7      p = p.next;
8    }
9    p.next = new Node(x,p.next);
10 }
```

start 노드를 가리키도록 초기화한 후에 (p.next != null)은 p가 리스트의 마지막 원소를 가리킬 때까지 루프를 반복시키고, 조건이 만족되면 p.next는 null이 되어 루프를 중단시킨다.
이러한 방식으로 리스트는 오름차순을 유지하게 된다.
p=p.next 는 연결 리스트를 탐색하는 표준적인 메커니즘이다. while 루프의 각 반복에서 이 할당문은 p를 리스트의 다음 노드로 이동시킨다.

5. 리스트의 앞에 삽입



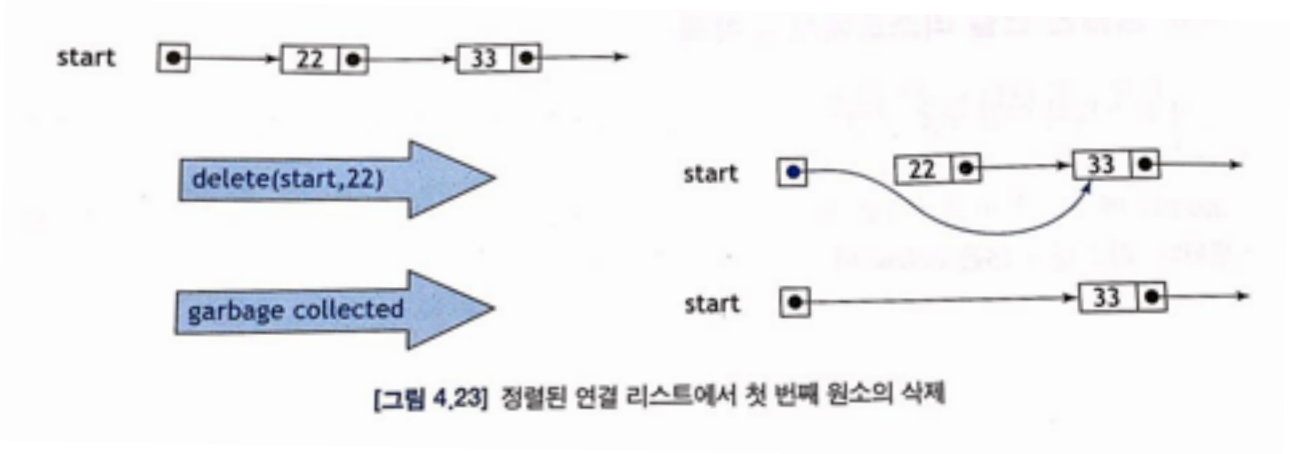
[그림 4.22] 20을 올바르게 삽입

연결 리스트의 구조를 변경하여 첫 번째 실제 데이터 노드 앞에 “빈(dummy)” 헤드 노드를 유지한다.
이는 약간의 추가 공간을 필요로 한다.
삽입이 리스트의 앞에서 수행되어야 하는 경우는 두 가지가 있다.
리스트가 비어 있거나 새 원소가 리스트의 첫 번째 원소보다 작은 경우이다.
첫번째의 경우, start = new Node(x); 새 노드로 start를 재설정한다.
두 번째의 경우, 새 노드를 start에 할당함은 물론이고, 새 노드를 리스트의 나머지 부분에 연결시켜야 한다. 그러나, 리스트의 시작에 대한 유일한 참조는 start 자체이므로, start를 새 노드로 재할당하기 전에 그 참조를 임시 변수에 보관해 두어야 한다.

6. 정렬된 연결 리스트에서의 삭제

순서 리스트의 연결 구조 구현은 원소의 이동을 불필요하게 하여 삽입을 훨씬 효율적으로 해주게 된다. 삭제의 경우도 마찬가지이다.

insert() 메소드와 마찬가지로 delete() 메소드도 (1) 원소를 발견하고, (2) 삭제하는 두 단계로 진행된다.



[그림 4.23] 정렬된 연결 리스트에서 첫 번째 원소의 삭제

리스트의 첫 번째 원소가 x와 동등한 경우에는 start.next를 start로 리턴해서 수행한다. 원래의 start 노드를 가리키는 아무런 참조도 없다면, 이 노드는 Java의 “쓰레기 수집기”에 의해서 삭제된다.

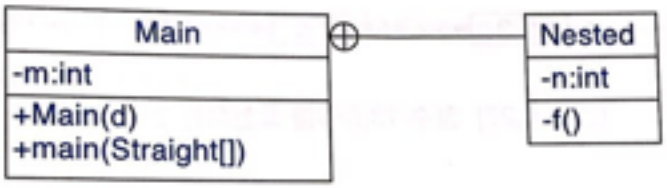
리스트의 첫 번째 원소가 x보다 작다면, x보다 크거나 같은 첫 번째 원소를 탐색한다.
더 큰 원소를 발견하면, 중단된다.
동등한 원소를 발견하면, 삭제가 수행된다.

7. 중첩 클래스

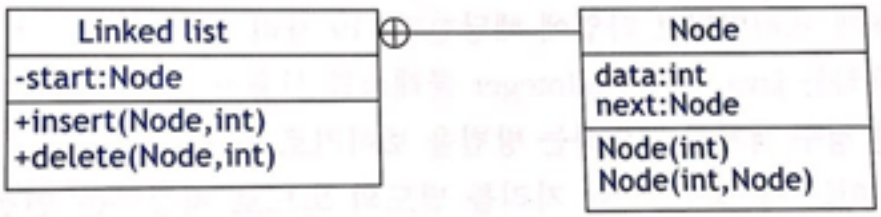
Java의 클래스 멤버는 필드, 생성자, 메소드, 인터페이스, 또 다른 클래스 등이다.
다른 클래스의 멤버인 클래스를 중첩 클래스(nested class)라고 한다.

클래스 Y가 사용될 유일한 장소가 다른 클래스 X의 내부인 경우, 클래스 Y는 클래스 X안에 중첩되어야 한다.
이는 정보 은폐 원리의 중요한 예가 된다.

X가 임의의 타입(클래스나 인터페이스)이고 Y가 X안에 중첩된 또 다른 타입이라면,
X의 모든 멤버는 Y로부터 접근 가능하고, Y의 모든 멤버도 X로부터 접근 가능하다.



[그림 4.25] 중첩 클래스에 대한 UML 다이어그램



[그림 4.26] LinkedList 노드 안에 중첩된 Node 클래스

한 클래스를 다른 클래스 안에 중첩시키는데 사용하는 UML 기호는 [그림 4.25] 안에 보인 것처럼 화살표 머리의 원안에 더하기 표시를 사용한다.

private 중첩 클래스의 모든 멤버들은 바깥쪽 클래스의 모든 위치로부터 접근 가능하므로, 단순성을 위해 대개 접근 수정자(private, public, protected)를 생략하고 선언한다.

보통의 경우, 중첩 클래스는 그 인스턴스가 바깥쪽 클래스의 비-static(내부클래스) 멤버를 접근할 필요가 있지 않는 한, static으로 선언되어야 한다.

Node 클래스는 구현하려는 연결 리스트의 문맥 안에서만 사용이 되기 때문에 List 클래스 안쪽에 중첩되어야 한다.
노드들이 List 메소드나 필드를 접근할 필요가 없으므로 Node 클래스는 static 중첩 클래스로 선언되어야 한다.

Node 클래스를 LinkedList 클래스 안에 숨기는 일은, LinkedList 클래스를 캡슐화시켜 이 클래스를 독자적으로 만들고 그 구현 세부 사항을 숨겨주게 된다. 개발자는 그 클래스 외부의 아무런 코드도 수정할 필요 없이 그 구현을 변경할 수 있다.