제8장 연결구조 확장

이 장에서는 연결구조를 이용한 리스트 구현의 여러 가지 확장을 살펴본다.

8.1. 교육목표

이 장의 교육목표는 다음과 같다.

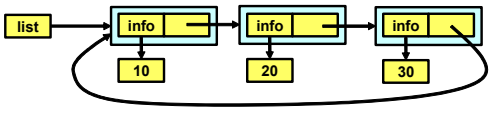
교육목표

- 순환 연결 리스트
- 이중 연결 리스트
- 배열을 이용한 연결구조 방식의 리스트 구현

8.2. 순환 연결 리스트

순환 연결 리스트

- 연결 리스트에서 마지막 노드의 링크를 첫 노드를 가리키도록 변형한 리스트



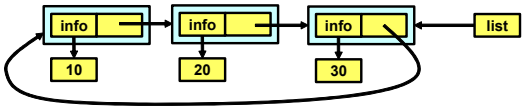
  - 장점. 어떤 노드에서 출발해도 전체 리스트를 방문할 수 있다.
- 기존 연산들은 항상 마지막 노드가 첫 노드를 가리키도록 수정해야 한다.
  - 수정시 가장 큰 문제점
    - 첫 노드의 삭제 또는 리스트 맨 앞에 노드의 삽입: 마지막 노드까지 이동하여야 한다.

순환 연결 리스트는 7장에서 살펴본 순환 연결 큐와 유사하다. 보통 연결구조 방식의 리스트에서는 첫 번째 노드를 가리키는 정보만 유지하고 마지막 노드의 연결 정보는 null 값을 가지게 된다. 이것을 변형하여 마지막 노드가 첫 노드를 가리키도록 하면 어떤 노드에서 출발하여도 전체 리스트를 방문할 수 있는 이점을 얻을 수 있다. 하지만 이전처럼 첫 번째 노

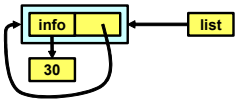
드를 가리키는 정보만 유지할 경우에는 첫 노드로 새 요소가 삽입되거나 첫 노드를 삭제해야 하는 경우에 비용이 너무 많이 소요된다.

순환 연결 리스트 – 계속

- 해결책: 첫 노드 대신에 마지막 노드를 가리키는 포인터 유지

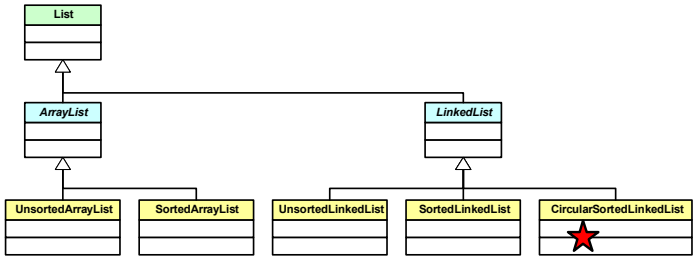


- 첫 노드와 마지막 노드를 접근하기가 모두 용이함
  - 마지막 노드의 내용: list.info
  - 첫 노드의 내용: list.next.info



이것을 해결하여 위해 첫 노드에 대한 정보를 유지하지 않고 마지막 노드에 대한 정보를 유지할 수 있다. 이렇게 하면 첫 노드와 마지막 노드에 대한 접근이 모두 용이해진다. 따라서 첫 노드로 새 요소를 삽입하는 비용이나 첫 노드를 삭제하는 비용이 모두 상수 시간이다. 뿐만 아니라 정렬 리스트를 순환 연결 리스트로 구현하면 가장 큰 값과 가장 작은 값을 상수 시간에 접근할 수 있으므로 이를 이용하여 각 종 연산들을 매우 효율적으로 개선할 수 있다.

List Class Hierarchy



이 절에서는 순환 정렬 연결 리스트를 구현하는 방법을 살펴본다. 순환 정렬 연결 리스트를 구현하기에 앞서 기존 리스트 관련 클래스 계층 구조에서 어디에 순환 연결 리스트를 위치해야 하는지 결정해야 한다. 순환 연결 리스트도 기존 연결구조 방식의 리스트와 마찬가지로 ListNode라는 내부 클래스를 사용해야 하며, isFull, isEmpty, size와 같은 메소드들은 LinkedList 추상 클래스에 정의되어 있는 것을 그대로 사용할 수 있다. 하지만 삽입, 삭제,

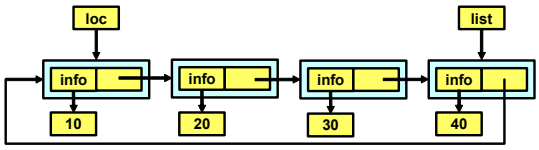
검색 메소드는 기존 일반 정렬 연결 리스트의 정의되어 있는 메소드를 그대로 사용할 수 없다. 이것은 기존 정렬 연결 리스트는 list라는 멤버 변수가 첫 노드를 가리킨 반면 순환 연결 방식에서는 마지막 노드를 가리키고 있기 때문이다. 따라서 순환 정렬 연결 리스트는 SortedLinkedList를 상속받는 것보다는 LinkedList를 상속받는 것이 올바르다.

CircularSortedLinkedList

```
public class CircularSortedLinkedList extends LinkedList{
    public CircularSortedLinkedList() { super(); }
    public boolean search(Object item) throws ListUnderflowException{ … }
    public void insert(Object item){   … }
    public boolean delete(Object item) throws ListUnderflowException{ … }
    public Object retrieve(Object item) throws ListUnderflowException{ … }
    public Iterator iterator() {
        return (list==null) ? new ListIterator(null) : new ListIterator(list.next);
    }
}
```

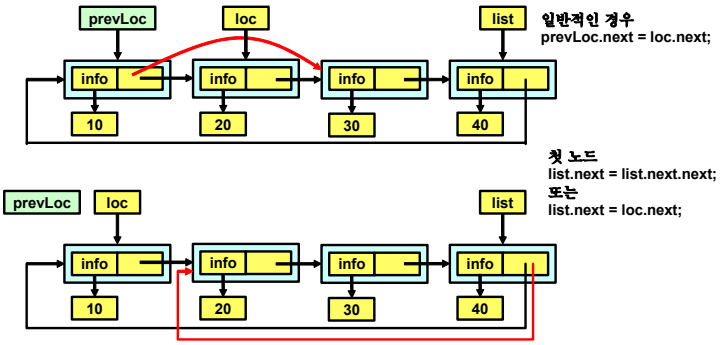
CircularSortedLinkedList의 search

```
public boolean search(Object item) throws ListUnderflowException{
    if(isEmpty()) throw new ListUnderflowException("…");
    Comparable x = (Comparable)item;
    ListNode loc = list.next;
    for(int i=0; i<size; i++){
        int compResult = x.compareTo(loc.info);
        if(compResult==0) return true;
        else if(compResult<0) return false;
        else loc = loc.next;
    }
    return false;
} // CircularSortedList::search
```

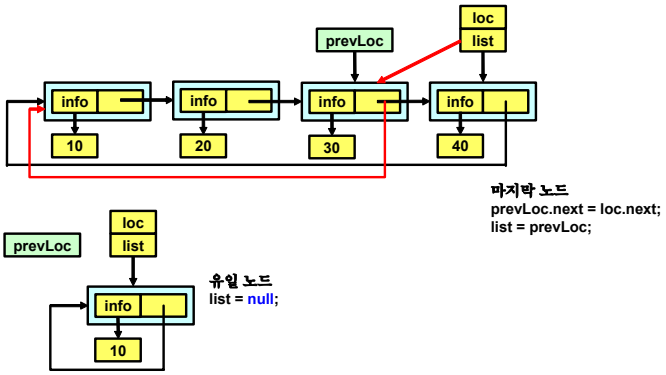


순환 정렬 연결 리스트에서 검색은 기존 정렬 연결 리스트에서 검색과 loc의 초기화만 다를 뿐 같다. 삭제와 삽입 연산은 loc의 초기화뿐만 아니라 첫 노드와 마지막 노드가 조작되면 순환을 유지하기 위한 추가 작업이 필요하다.

CircularSortedLinkedList의 delete



CircularSortedLinkedList의 delete



항상 연결구조 방식의 구현에서는 크게 네 가지 경우를 고려해야 한다. 첫 노드의 삭제, 첫 노드의 삭제의 특수한 경우로 리스트가 노드에 하나밖에 없는 경우, 중간 노드의 삭제, 마지막 노드의 삭제를 고려해야 한다. 중간 노드의 삭제는 기존 정렬 연결 리스트와 동일하지만 첫 노드의 삭제 또는 마지막 노드의 삭제에 대해서는 순환을 유지하기 위한 추가 작업이 필요하다.

```
public boolean delete(Object item) throws ListUnderflowException{
    if(isEmpty()) throw new ListUnderflowException("…");
    Comparable x = (Comparable)item;
    ListNode loc = list.next; ListNode prevLoc = null;
    for(int i=0;i<size;i++){
        int compResult = x.compareTo(loc.info);
        if(compResult==0){    // 삭제할 노드가 존재하는 경우
            if(prevLoc==null){ // 첫 노드를 삭제하는 경우
                if(loc == loc.next) list = null;  // 첫 노드가 유일 노드인 경우
                else list.next = loc.next;
            }
            else{
                prevLoc.next = loc.next;
                if(loc==list) list = prevLoc;     // 마지막 노드를 삭제하는 경우
            }
            size--;
            return true;
        }
        else if(compResult<0) return false;
        else{
            prevLoc = loc; loc = loc.next;
        }
    } // for
    return false;
} // CircularSortedList::delete
```

if(loc == loc.next) 대신에
if(size==1)를 사용할 수 있음

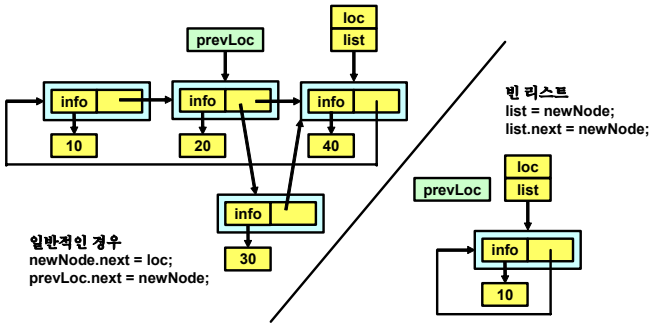
첫 노드와 마지막 노드를 모두 접근하기가 용이하다는 순환 연결 구조의 특성을 이용하여 보다 효율적으로 delete 메소드를 구현할 수 있다. 즉, 첫 노드와 마지막 노드와 비교하여 첫 노드보다 작은 경우와 첫 노드와 큰 경우에 대해서는 리스트 전체를 검색하여 삭제할 노드가 있는지 찾는 과정을 생략할 수 있다.

첫 노드와 마지막 노드를
모두 접근하기가 용이하다는
순환 연결 구조의 특성을
이용한 delete 메소드

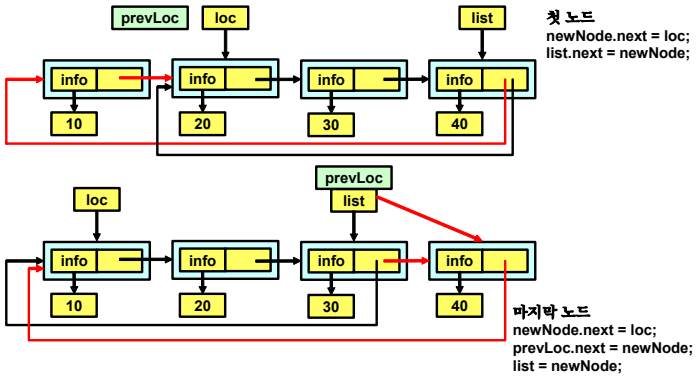
```
public boolean delete(Object item) throws ListUnderflowException{
    if(isEmpty()) throw new ListUnderflowException("…");
    Comparable x = (Comparable)item;
     if(x.compareTo(list.next.info)<0||x.compareTo(list.info)>0){
         return false;
     }
     // 나머지 부분은 동일

} // CircularSortedList::delete
```

CircularSortedLinkedList의 insert



CircularSortedLinkedList의 insert



삽입도 삭제와 마찬가지로 크게 네 가지 경우를 고려해야 한다. 리스트 맨 앞에 추가하는 경우, 맨 앞에 추가의 특수한 경우로 리스트가 비어 있는 경우, 중간에 삽입하는 경우, 리스트 맨 끝에 삽입하는 경우를 고려해야 한다. 각 경우에 유사점이 있는 경우에는 이를 활용하여 구현하는 것이 바람직하다. 예를 들어 중간에 삽입하는 경우와 맨 끝에 삽입하는 경우 모두 동일하게 newNode,next = loc; prevLoc.next = newNode;는 실행되어야 한다.

```
public void insert(Object item){
    Comparable x = (Comparable)item;
    ListNode newNode = new ListNode();
    newNode.info = item; newNode.next = null;
    ListNode loc = (list==null)? null: list.next;
    ListNode prevLoc = null;
    for(int i=0;i<size;i++){ // 삽입할 위치 찾기
        if(x.compareTo(loc.info)<0) break;
        else{
            prevLoc = loc;  loc = loc.next;
        }
    }  // for
    if(prevLoc==null){ // 리스트 맨 앞에 삽입되는 경우
        // 리스트가 비어있는 경우
        if(list==null){ list = newNode; newNode.next = newNode; }
        else{ newNode.next = loc; list.next = newNode; }
    }
    else{
        newNode.next = loc;   prevLoc.next = newNode;
        // 리스트의 맨 마지막에 삽입되는 경우
        if(prevLoc == list) list = newNode;
    }
    size++;
} // CircularSortedList::insert
```

위 슬라이드에 있는 삽입 연산은 기존 정렬 연결 리스트와 동일한 방법으로 구현한 것이다. 먼저 삽입할 위치를 찾은 다음, 삽입할 위치에 따라 필요한 각 종 연결을 변경하는 방식이다. 다음 슬라이드에 있는 삽입 연산은 순환 연결 리스트의 특성을 활용하고 있다. 즉, 새 요소를 추가할 때 그 요소를 리스트의 첫 요소와 마지막 요소와 비교하여 첫 요소보다 작거나 마지막 요소보다 클 경우에는 이것을 먼저 처리하여 준다. 두 경우는 정렬된 순환 연결 구조의 특성 상 같은 위치에 삽입되는 형태가 된다.

첫 노드와 마지막 노드를 모두 접근하기가 용이하다는 순환 연결 구조의 특성을 이용한 insert 메소드

```
public void insert(Object item){
    Comparable x = (Comparable)item;
    ListNode newNode = new ListNode();
    newNode.info = item; newNode.next = null;
    size++;
    if(list==null){
        list = newNode;  newNode.next = newNode;  return;
    } // 리스트가 비어 있는 경우
    if(x.compareTo(list.next.info)<0||x.compareTo(list.info)>0){
        newNode.next = list.next;  list.next = newNode;
        if(x.compareTo(list.info)>0) list = newNode; // 맨 끝에 삽입되는 경우
        return;
    } // 삽입하고자 하는 노드가 첫 노드보다 작거나 마지막 노드보다 큰 경우
    // 첫 노드와 비교할 필요 없음
    ListNode loc = list.next.next;  ListNode prevLoc = list.next;
    for(int i=0;i<size;i++){
        if(x.compareTo(loc.info)<0){
            newNode.next = loc;  prevLoc.next = newNode;  return;
        }
        else{
            prevLoc = loc; loc = loc.next;
        }
    }  // for
} // CircularSortedList::insert
```

CircularSortedLinkedList의 장단점

- 연산 측면에서는 SortedLinkedList에 비해 향상된 것이 없다.
- 그러나
  - CircularSortedLinkedList는 일반 정렬 연결리스트와 달리 첫 노드와 마지막 노드에 대한 접근이 용이하다.
  - 따라서 다음과 같은 것은 일반 정렬 연결리스트보다 우수하다.
    - 가장 큰 값의 추출
    - 리스트에 저장된 값들 사이에 있는 값인지 비교하는 것
    - 정렬된 데이터를 연속해서 삽입하는 경우
  - 일반 정렬 연결리스트에 마지막 노드를 가리키는 포인터를 하나 더 유지하는 것은? good alternative

순환 정렬 연결 리스트의 장점을 기존 정렬 연결 리스트와 비교하여 보자. 연산의 성능 측면에서 보면 향상된 것이 거의 없다. 그러나 순환 정렬 연결 리스트는 첫 노드뿐만 아니라 마지막 노드에 대한 접근이 용이하므로 다음과 같은 서비스는 기존 정렬 연결 리스트보다 우수하다.

첫째, 가장 큰 값의 추출

둘째, 리스트에 저장된 값들 사이에 있는 값인지 알 필요가 있는 경우

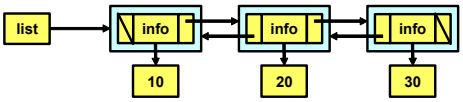
셋째, 정렬된 데이터를 연속해서 삽입하는 경우

마지막 서비스에 경우에는 삽입 연산이 순환 연결 리스트의 특성을 활용하고 있어야 한다. 이런 측면에서 보았을 때 일반 정렬 연결 리스트에서 마지막 노드를 가리키는 포인터를 하나 더 유지하는 방법을 생각해 볼 수 있다. 오히려 이 방법이 순환 연결 리스트에 비해 우수할 수 있다.

8.3. 이중 연결 리스트

이중 연결 리스트

- 이중 연결 리스트(doubly linked list)란 첫 노드와 마지막 노드를 제외하고는 모두 후속 요소뿐만 아니라 선행 요소를 가리키는 포인터를 추가로 가지고 있는 리스트

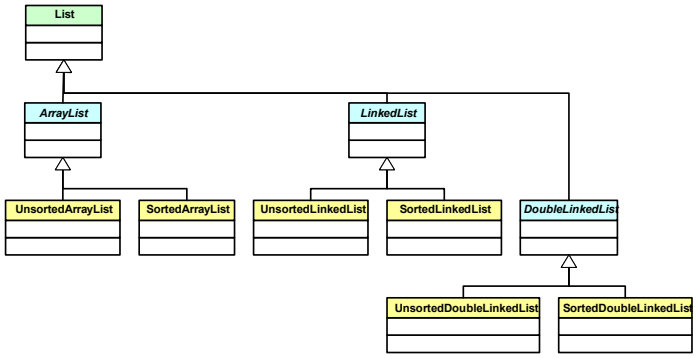


- 어떤 노드에서 출발해도 전체 리스트를 방문할 수 있다.
- 역순으로 노드들을 방문할 수 있다.
- 노드의 구성: back, info, next
  - 3N+1

```
protected class DLListNode{
    public Object info;
    public DLListNode back;
    public DLListNode next;
}
```

이전까지의 연결구조 방식의 자료구조의 각 노드는 한 쪽 방향의 연결만을 유지하고 있었다. 이중 연결 리스트는 이와 달리 양쪽 방향의 연결을 한 노드에 모두 유지한다. 따라서 한 노드에 3개의 참조를 유지해야 하므로 노드 당 차지하는 공간이 많다. 이중 연결 리스트는 현재 위치한 노드로부터 이전 연결구조 방식의 자료구조처럼 후속 요소들을 접근할 수 있을 뿐만 아니라 선행 요소들도 접근할 수 있다. 이 기능 때문에 기존 연결구조 방식의 자료구조와 달리 역순으로 노드들을 방문할 수 있다는 장점을 지니고 있다.

List Class Hierarchy



이중 연결 리스트는 노드 자체가 일반 연결 리스트와 다르므로 LinkedList 클래스와 상속 관계를 가질 수 없다. 따라서 DoubleLinkedList라는 추상 클래스를 만들어 List 인터페이스를 구현하도록 하고, 이중 정렬 연결 리스트와 이중 비정렬 연결 리스트는 DoubleLinkedList를 상속받아 구현하도록 한다.

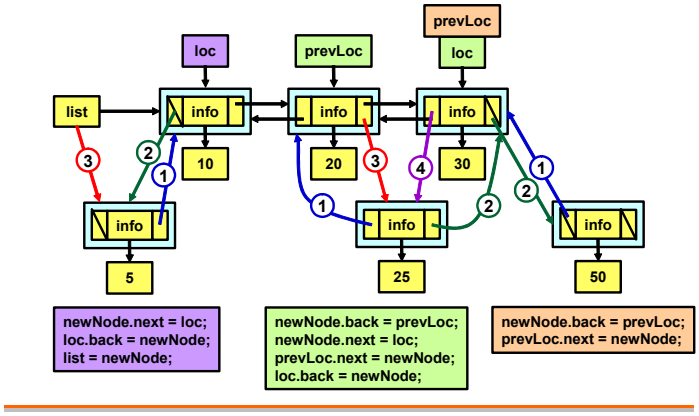
```
public abstract class DoubleLinkedList implements List{
    protected class ListNode{
        public Object info;
        public ListNode back;
        public ListNode next;
    }
    protected class ListIterator{
        ...
    }
    protected ListNode list = null;
    protected int size = 0;
    public DoubleLinkedList() {}
    public abstract boolean search(Object item) throws ListUnderflowException;
    public abstract void insert(Object item);
    public abstract boolean delete(Object item) throws ListUnderflowException;
    public abstract Object retrieve(Object item) throws ListUnderflowException;
    public boolean isFull(){ return false; }
    public boolean isEmpty(){ return (list == null); }
    public void clear{ list = null; }
    public int size(){ return size; }
    public Iterator iterator() { return new ListIterator(list); }
} // DoubleLinkedList class
```

DoubleLinkedList에서 노드를 나타내는 클래스의 이름은 ListNode이고, 반복자 클래스의 이름은 ListIterator이다. 즉, LinkedList가 사용하는 이름과 동일한 이름을 사용한다. 하지만 내부 클래스의 이름이므로 이렇게 같은 이름을 사용하여도 문제가 되지 않는다.

```
protected class ListIterator{
    public ListNode cursor;
    public int traverseCount = 0;
    public ListIterator(ListNode node){
        cursor = node;
    } // ListIterator(ListNode)
    public boolean hasBack(){ return (traverseCount>0); }
    public boolean hasNext(){  return (traverseCount<size); }
    public Object back(){
        Object tmp = cursor.info;
        cursor = cursor.back;
        traverseCount--;
        return tmp;
    } // back
    public Object next(){
        Object tmp = cursor.info;
        cursor = cursor.next;
        traverseCount++;
        return tmp;
    } // next
    public void remove(){
        throw new UnsupportedOperationException();
    } // remove
}
```

이중 연결 리스트에서 반복자 클래스에는 기존 연결 리스트에는 없는 back과 hasBack 메소드가 추가로 구현되어야 한다. 즉, 주어진 노드로부터 그것의 선행 노드를 방문할 수 있는 메소드를 제공해주어야 한다.

SortedDoubleLinkedList의 Insert

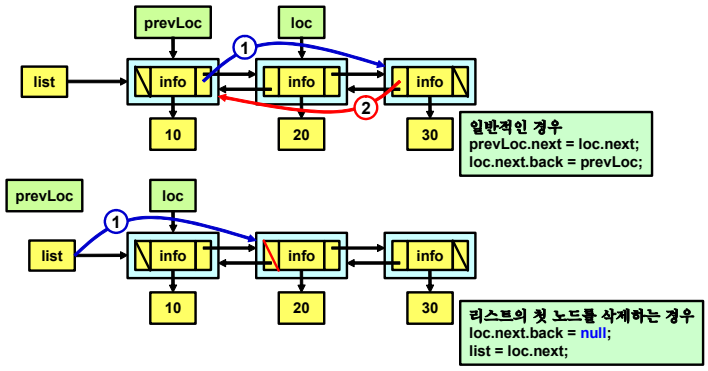


이중 정렬 연결 리스트에서 삽입 연산은 일반 정렬 연결 리스트의 삽입과 마찬가지로 크게 네 가지 경우를 고려해야 한다. 위 슬라이드에서 이 중 리스트가 빈 경우에 삽입하는 경우를 제외한 나머지 세 가지 경우에 대해 연결을 변경하는 방법을 설명하고 있다.

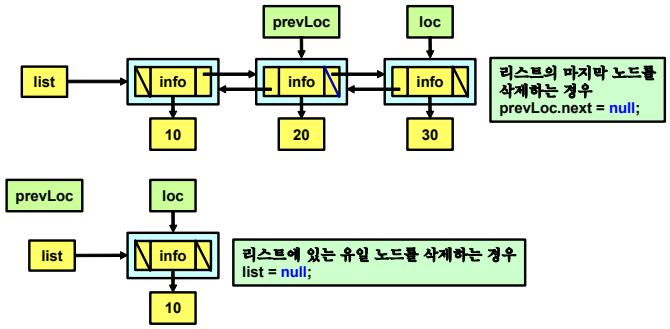
```
public void insert(Object item){
    Comparable x = (Comparable)item;
    ListNode newNode = new ListNode();
    newNode.info = item;  newNode.back = null;  newNode.next = null;
    ListNode loc = list;
    ListNode prevLoc = null;
    for(int i=0;i<size;i++){ // 삽입할 위치를 찾는다.
        if(x.compareTo(loc.info)<0) break;
        else{
            prevLoc = loc;  loc = loc.next;
        }
    } // for
    if(prevLoc==null){ // 리스트 맨 앞에 삽입되는 경우
        list = newNode;  newNode.next = loc;
        if(list!=null) loc.back = newNode;  // 빈 리스트가 아닌 경우
    }
    else{
        newNode.back = prevLoc;
        newNode.next = loc;
        prevLoc.next = newNode;
        if(loc!=null) loc.back = newNode; // 맨 마지막에 삽입되는 경우가 아니면
    }
    size++;
} // SortedDoubleLinkedList::insert
```

if(prevLoc==null)이 참이면 새 요소를 리스트의 첫 노드로 삽입하는 경우이다. 이 경우는 다시 리스트가 빈 경우와 그렇지 않은 경우로 나누어진다. 반대로 조건이 거짓이면 새 요소를 리스트 중간 또는 끝에 삽입하는 경우이다. 리스트 중간에 삽입되는 경우에는 새 요소가 선행 및 후속 요소를 모두 가지게 되지만 리스트 마지막에 삽입되는 경우에는 선행요소만 가지게 된다.

SortedDoubleLinkedList의 delete



SortedDoubleLinkedList의 delete



이중 연결 리스트에서 삭제도 크게 네 가지 경우를 고려해야 한다. 리스트에 존재하는 유일 요소를 삭제하는 경우와 리스트의 맨 앞 요소를 삭제하는 경우를 같이 고려할 수 있다. 이 때 유일한 차이점은 하나는 list가 첫 요소의 후속 요소를 가리키도록 변경되어야 하지만 다른 하나는 list 값이 null이 되어야 한다. 리스트 중간에 있는 요소를 삭제하는 경우와 리스트 끝에 있는 요소를 삭제하는 경우도 같이 고려할 수 있다. 두 경우의 유일한 차이점은 리스 끝에 있는 요소는 후속 노드가 없으므로 그것의 선행 노드 정보만 변경하면 된다.

```
public boolean delete(Object item) throws ListUnderflowException{
    if(isEmpty()) throw new ListUnderflowException("…");
    Comparable x = (Comparable)item;
    ListNode loc = list; List prevLoc = null;
    for(int i=0;i<size;i++){
        int compResult = x.compareTo(loc.info);
        if(compResult==0){    // 삭제할 노드가 존재하는 경우
            if(prevLoc==null){ // 첫 노드를 삭제하는 경우
                if(loc.next==null) list = null; // 삭제할 노드가 유일 노드인 경우
                else{
                    loc.next.back = null; list = loc.next;
                }
            }
            else{
                prevLoc.next = loc.next;
                if(loc.next != null) loc.next.back = prevLoc; // 마지막 노드가 아닌 경우
            }
            return true;
        }
        else if(compResult<0) return false;
        else{
            prevLoc = loc; loc = loc.next;
        }
    } // for
    return false;
}
```

8.4. 배열을 이용한 연결 리스트의 구현

배열을 이용한 연결구조의 구현

색인	info	next
0	David	2
1	null	5
2	John	4
3	Peter	6
4	Mary	3
5	null	END
6	Robert	END

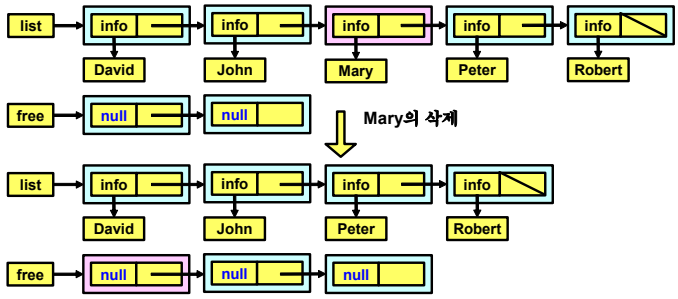
list	0	free	1	size	5

```
class ArrayLinkedList{
    public static final int DEF_MAX_CAPACITY = 50;
    public static final int END = -1;
    private class ListNode{
        public Object info;
        public int next;
    }
    private int list = END;
    private ListNode[] nodes;
    private int size = 0;
    private int free = 0;
    …
}
```

배열을 이용한 연결구조에서 연결은 배열의 색인정보가 된다. 따라서 null 대신에 -1을 이용하여 리스트의 끝을 나타낸다.

이 슬라이드에서 보여 주고 있듯이 배열을 이용하여 연결 리스트를 구현할 수도 있다. 일반적으로 연결구조 방식에서는 새 요소가 추가될 때마다 동적으로 메모리 공간을 확보한다. 하지만 배열을 이용한 연결 리스트에서는 미리 필요한 만큼의 충분한 메모리 공간을 확보한 다음에 이 공간을 사용자가 시스템을 대신하여 직접 관리한다. 배열을 이용한 연결 리스트에서 배열의 각 항은 연결구조의 노드를 나타낸다. 따라서 크게 두 가지 차이점이 있다. 첫째, 배열을 이용한 연결 리스트에서 삽입 연산은 더 이상 사용할 항이 없어 처리할 수 없는 경우가 있다. 둘째, 각 노드의 연결 정보는 기존과 달리 주소 정보가 아니라 배열의 색인 정보가 된다. 그러므로 첫 노드를 가리키는 정보 역시 배열의 색인 정보로 나타낸다. 또한 일반적인 연결구조에서 마지막 노드임을 나타내기 위해 그 노드의 연결 값을 null 값으로 사용한다. 배열을 이용한 연결 리스트의 구현에서는 null 값 대신에 -1이라는 값을 사용한다. 이것은 -1은 유효한 색인 정보가 될 수 없는 값이기 때문이다.

배열을 이용한 연결구조의 구현 – 계속

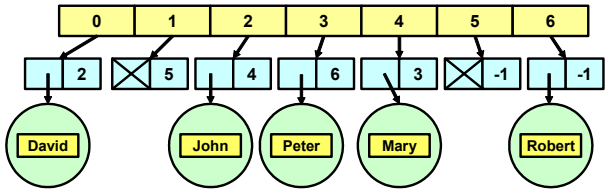


- 배열을 이용한 연결구조의 구현은 내부적으로 두 개의 리스트를 유지한다.
  - 즉, 프로그래머가 스스로 빈 공간을 관리해야 한다.

보다 자세하게 구현 원리를 살펴보면 배열을 이용한 연결구조는 내부적으로 두 개의 리스트를 유지한다. 하나는 요소들이 저장되어 있는 유효한 노드들을 연결해 놓은 리스트이고, 다른 하나는 요소들이 저장되어 있지 않은 빈 노드를 연결해 놓은 리스트이다. 만약 새 요소를 추가해야 하면 빈 노드들의 리스트에서 하나의 노드를 선택하여 이 노드에 새 요소를 대입하고 이 노드를 유효한 노드들의 리스트에 포함한다. 반대로 기존 요소를 삭제해야 하면 그 노드를 유효한 노드들의 리스트에서 제거하고 빈 노드들의 리스트로 옮긴다.

배열을 이용한 연결구조의 구현 – 계속

- 배열을 이용한 연결구조 구현의 장점
  - 매번 삽입할 때마다 공간 할당이 이루어지지 않는다.
  - 동적 메모리 할당을 제공하지 않는 프로그래밍 환경에서는 이 방법이 유일한 대안이다.
- 배열을 이용한 연결구조 구현의 단점
  - 일반적인 연결구조 방식과 달리 삽입 연산의 경우 리스트가 꽉 찬 경우를 고려해야 한다.



배열을 이용한 연결 리스트는 일반 연결 리스트와 달리 매번 삽입할 때마다 공간 할당이 이루어지지 않는다는 장점을 지니고 있다. 하지만 배열의 본질적인 특성 때문에 삽입 연산에서 공간이 부족한 경우를 고려해야 한다.

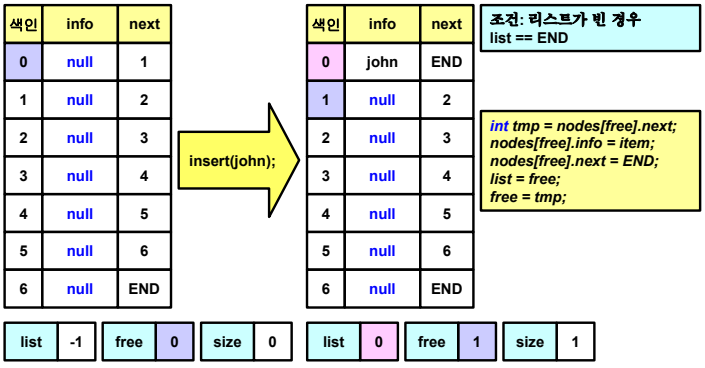
ArrayLinkedList의 생성자

```
public ArrayLinkedList(int capacity){
    if(capacity<0) setup(DEF_LIST_CAPACITY);
    else setup(capacity);
} // ArrayLinkedList(int)
public void setup(int capacity){
    nodes = new ListNode[capacity];
    // 빈 노드들의 리스트 생성
    for(int i=0;i<capacity;i++){
        nodes[i] = new ListNode();
        nodes[i].info = null;
        nodes[i].next = i+1;
    } // for
    nodes[capacity-1].next = END;
} // ArrayLinkedList::setup(int)
```

- 미리 모든 노드를 만든다.
- 새로운 item을 삽입할 때에는 새 노드를 만들지 않고 배열에 만들어 놓은 노드에 item을 삽입한다.

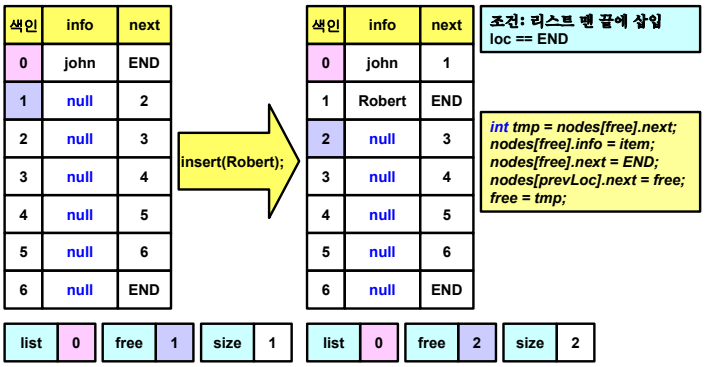
배열을 이용한 연결 리스트는 리스트를 처음 생성할 때 모든 노드를 미리 만든다. 또한 앞서 언급한 바와 같이 내부적으로 두 개의 리스트를 유지해야 하므로, 처음 생성할 때에는 배열의 모든 항들을 연결하여 빈 노드들의 리스트를 만들어야 한다.

SortedArrayLinkedList의 insert



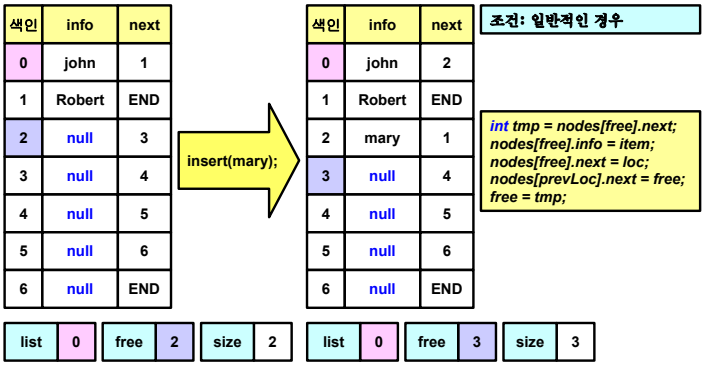
배열을 이용한 정렬 연결 리스트에서 삽입은 기존 정렬 리스트에서 다음과 같은 특성을 제외하고는 차이가 없다. 첫째, 새 요소를 추가하기 위해 new를 사용하여 노드를 만들지 않고, 빈 노드들의 리스트로부터 하나의 노드를 받는다. 둘째, 프로그래밍 할 때 사용하는 표기 방식이 다르다. 기존에는 loc.info = item과 같은 형태로 프로그램을 작성하였지만 배열을 이용한 구현에서는 nodes[loc].info = item와 같이 배열 선택식이 포함된 형태의 표현식을 사용하게 된다.

SortedArrayLinkedList의 insert

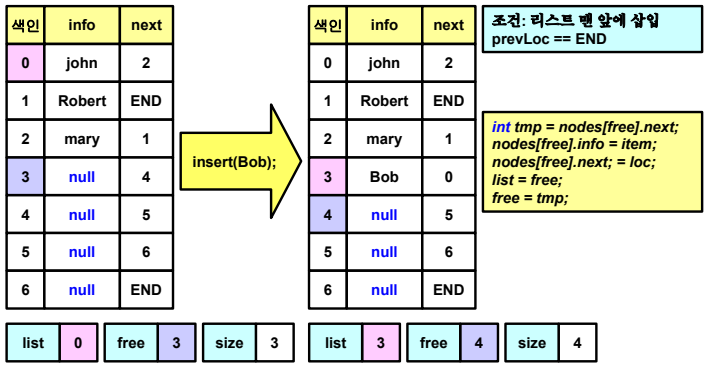


정렬 연결 리스트에서 삽입은 우선 삽입할 위치를 찾아야 한다. 위치를 찾으면 빈 리스트에서 노드를 가지고 와서 이 노드에 데이터를 추가하고 이 노드를 결정된 위치에 삽입한다.

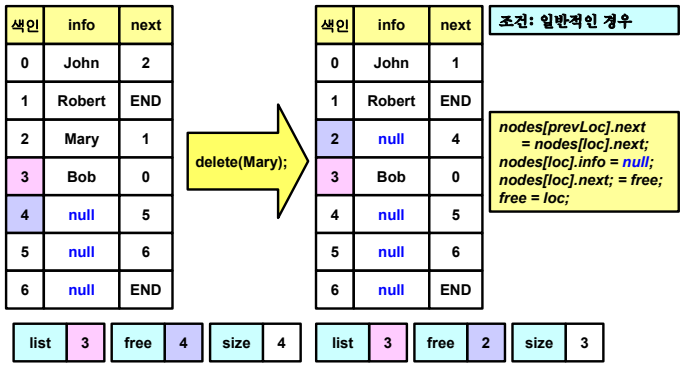
SortedArrayLinkedList의 insert



SortedArrayLinkedList의 insert



SortedArrayLinkedList의 delete



리스트 중간에 있는 Mary가 삭제되는 경우 이 노드는 빈 리스트로 옮겨져야 한다. 리스트의 특성 상 비정렬 리스트의 경우에는 맨 앞에 삽입하는 것이 가장 효율적이므로 삭제되는 노드는 빈 리스트의 첫 노드가 된다.