06. 스택

스택(Stack)

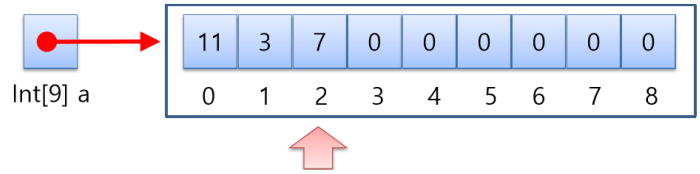
- 후입선출(**LIFO**: last-in-first-out) 프로토콜을 구현하는 자료구조

ADT: Stack *(ADT: Abstract Data Type)*

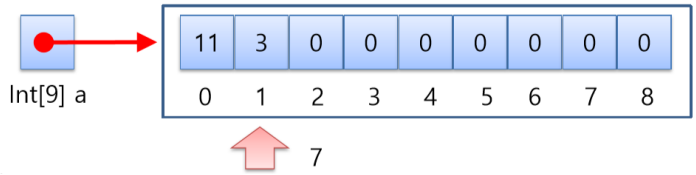
- 스택이란 LIFO접근 프로토콜을 유지하는 원소들의 컬렉션이다.
- 연산:
  - Peak: 스택이 공백이 아니면 톱의 원소를 리턴
  - Pop: 스택이 공백이 아니면 톱의 원소를 삭제해서 리턴
  - Push: 주어진 원소를 스택의 톱에 추가
  - Size: 스택에 있는 원소의 수를 리턴

배열을 이용한 ArrayStack

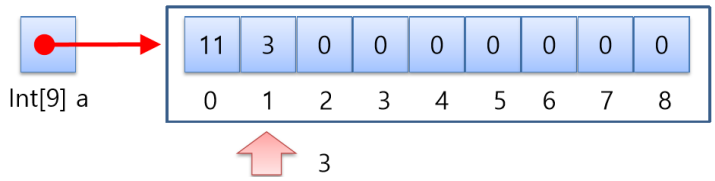
➢ Push(11), Push(3), **Push(7)**,



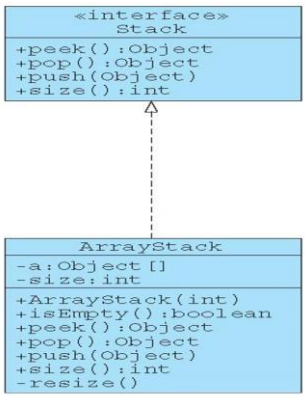
➢ Push(11), Push(3), Push(7), **Pop( )**, Peek( )



➢ Push(11), Push(3), Push(7), Pop( ), **Peek( )**



<스택 인터페이스의 구현>




```
public interface Stack {
    public Object peek();
    public Object pop();
    public void push( Object item );
    public int size();
    public boolean isEmpty();
}
```

```
public class ArrayStack implements Stack {
     private static final int INITIAL_LENGTH=10;
     private String[] stack = new String[INITIAL_LENGTH];
     private int size = 0;

     public Object peek() {
         if ( isEmpty() ) {
             return "Nothing";
         }
         else {
             return (Object)this.stack[size-1];
         }
     }

     public Object pop() {
         if ( isEmpty() ) {
             return "Nothing";
         }
         else {
             return this.stack[--size];
         }
     }

     public void push(Object item) {
         if ( size == this.stack.length ) {
             resize();
         }
         this.stack[size++] = (String)item;
     }


     private void resize() {
         String[] temp = new String[this.stack.length + INITIAL_LENGTH];
         System.arraycopy(this.stack, 0, temp, 0, this.size);
         stack = temp;
     }

     public int size() {
          return this.size;
     }

     public boolean isEmpty() {
          if ( this.size == 0 )return true;
          elsereturn false;
     }
}
```

응용: 후위식의 평가

- **중위 표기(infix notation)**: 일상적인 산술식 표기

  Ex) (8 -3) * (5 +6) , A/B**C+D*E–A*C

- **후위 표기(postfix notation):** 연산자가 항상 피연산자 뒤에 등장

  모호성이 존재하지 않으므로 괄호가 필요 없음. 일반적으로 컴파일러들이 사용.

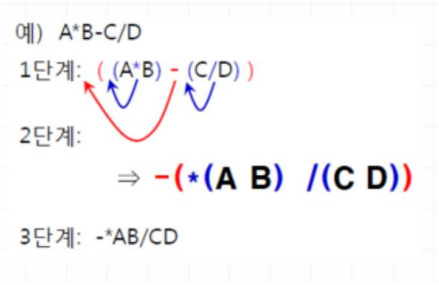
  피연산자는 스택에 삽입, 연산자를 만나면 스택에 마지막으로 삽입된 2개의 수를 2개의 피연산자로 가정. 이들을 삭제해서 해당 수식 계산 후 결과를 스택에 다시 삽입.

  Ex) 8 3 – 5 6 + * , ABC**/DE*+AC*–

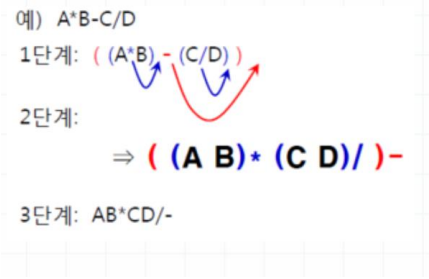
- **전위 표기(Prefix notation):** 연산자가 항상 피연산자 앞에 등장

  Ex) –+/A**BC*DE*AC

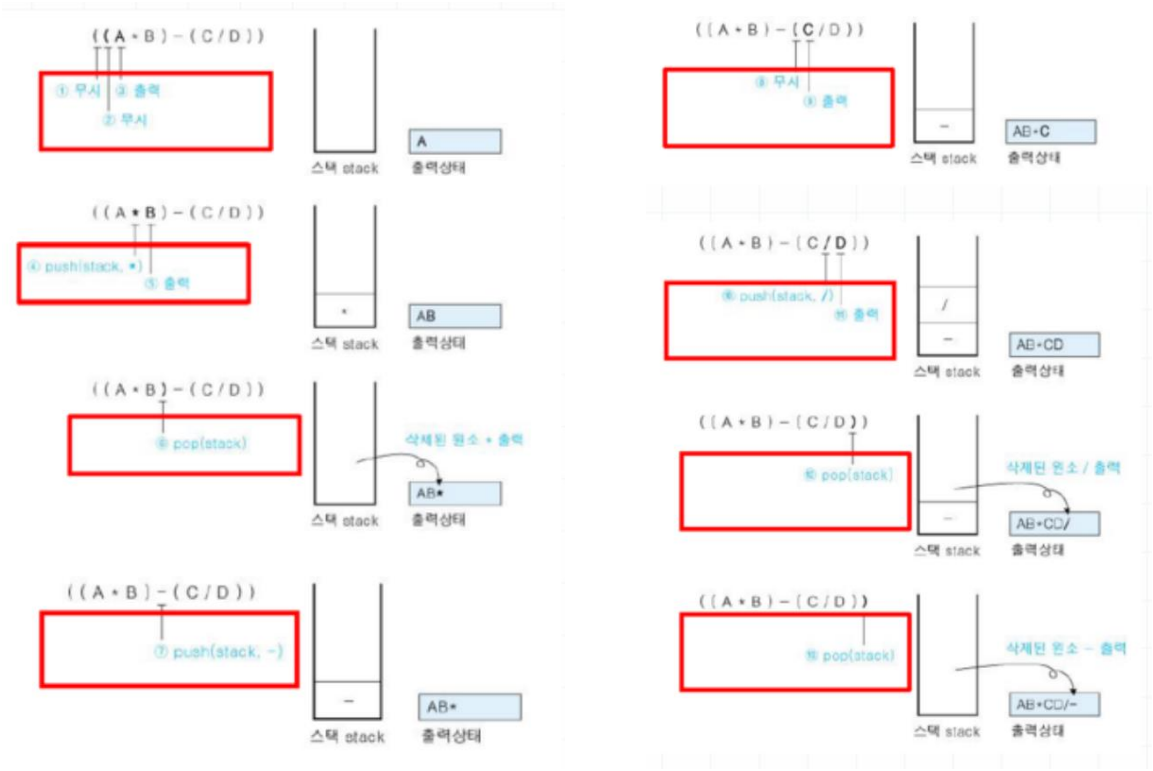
중위-> 전위표기법



중위-> 후위표기법



스택을 사용한 중위-> 후위표기법

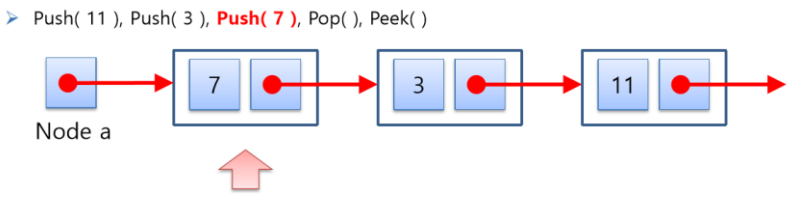


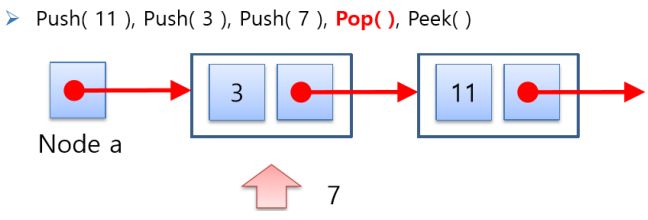
ArrayStack의 단점

- 스택이 꽉 찼을 때 배열을 재구축해야 하므로 비효율적
- 배열은 원소를 인접한 메모리 위치에 유지하므로 동적으로 성장할 수 없음
- 배열을 확장하려면 전체 배열을 재할당하고 복사해야함
-

LinkedList를 사용한 LinkedStack

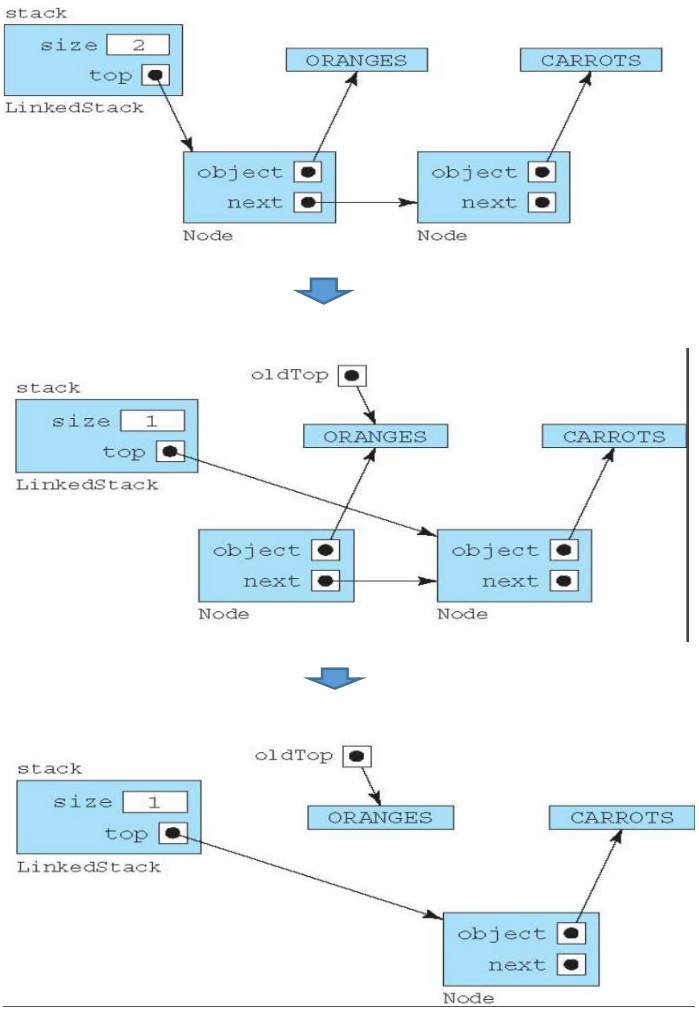
<stack 인터페이스의 연결 구현>





```
public class LinkedListStack implements Stack {
     private Node start;
     private int size = 0;

     public Object peek() {
         if ( isEmpty() ) {
             return "Nothing";
         }
         else {
             return this.start.data;
         }
     }

     public Object pop() {
         if ( isEmpty() ) {
             return "Nothing";
         }
         else {
             String temp = this.start.data;
             this.start = this.start.next;
             size--;
             return temp;
         }
     }

     public void push(Object item) {
          Node temp = new Node( (String)item );
          temp.next = this.start;
          this.start = temp;
          size++;
     }

     public int size() {
          return this.size;
     }

     public boolean isEmpty() {
          if ( this.size == 0 )return true;
          elsereturn false;
     }
}
```

Pop()메소드 호출 결과



Garbage collector에 의해 old top 삭제됨. (메모리 공간이 heap으로 리턴)

java.util.Stack

java.util.ArrayList

07. 큐

큐(Queue)

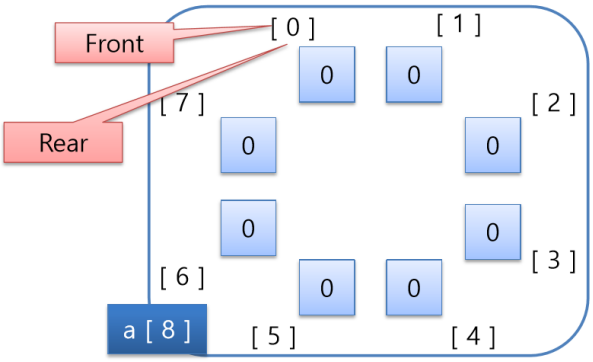
- 선입선출(**FIFO**: fitst-in-first-out) 프로토콜을 구현하는 자료 구조
- 원소의 삽입은 큐의 뒤(**rear**)에서 수행되고, 제거는 큐의 앞(**front**)에서 진행

ADT: Queue

- 큐란 FIFO접근 프로토콜을 유지하는 원소들의 컬렉션이다.
- 연산:
  - Add: 주어진 원소를 큐의 뒤에 삽입
  - First: 큐가 공백이 아니면 큐의 맨 앞에 있는 원소를 리턴
  - Remove: 큐가 공백이 아니면 큐의 앞에 있는 원소를 삭제 후 리턴
  - Size: 큐에 있는 원소의 수를 리턴

배열을 이용한 큐 – Circular Queue

➢ 선입선출 (FIFO : First In First Out)




```
public class CircularQueue {
    int[] a;
    int front;
    int rear;

    CircularQueue( ) {
        this.a = new int[8];
        this.front = 0;
        this.rear = 0;
    }
}
```

➢ 초기

- Queue로 활용할 저장공간(배열) 선언
- front와 rear는 각각 처음을 가리키는 형태

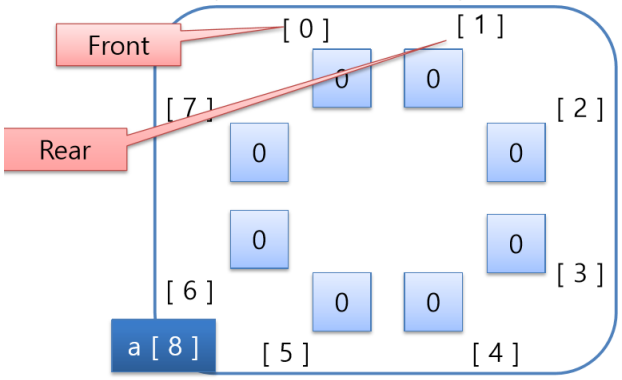
- **Add**

```
public boolean Add( int item ) {
      if ( this.isFull() ) {
            return false;
      }
      a[++rear] = item;
      return true;
}
private boolean isFull( ) {
      if ( rear < front ) {
            rear += INITIAL_LENGTH;
      }
      if ( (rear - front)
            == (INITIAL_LENGTH - 1) ) {
            return true;
      }
      return false;
}
```

- Add
  - isFull ( )
  - 먼저, 큐가 가득 찼는지 확인
    - Rear가 Front보다 작은 경우
    - (Rear – Front)가 (큐 크기-1)라면 꽉 찬 것!

- Add
  - isFull ( )
  - 먼저, 큐가 가득 찼는지 확인
    - Rear가 Front보다 작은 경우
    - (Rear – Front)가 (큐 크기-1)라면 꽉 찬 것!

- 선입선출 (FIFO : First In First Out)

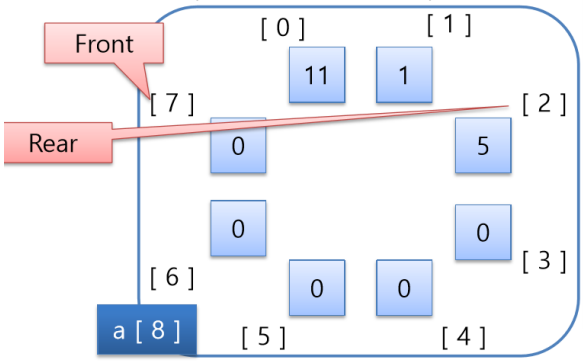



```
public boolean Add( int item ) {
      if ( this.isFull() ) {
            return false;
      }
      a[++rear] = item;
      return true;
}
private boolean isFull( ) {
      if ( rear < front ) {
            rear += INITIAL_LENGTH;
      }
      if ( (rear - front)
            == (INITIAL_LENGTH - 1) ) {
            return true;
      }
      return false;
}
```

- Add
  - ++rear
  - a [rear] = item

- **Delete**

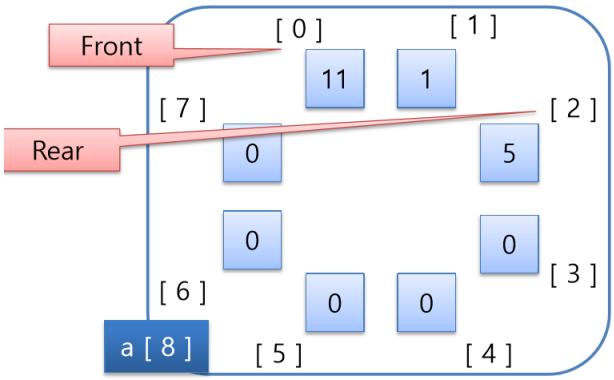
- 선입선출 (FIFO : First In First Out)




```
public int Delete( ) {
      if ( this.isEmpty() ) {
            return -1;
      }
      if ( front ==
            (INITIAL_LENGTH - 1) ) {
            front = -1;
      }
      return this.a[++front];
}
private boolean isEmpty( ) {
      if ( front == rear ) {
            return true;
      }
      return false;
}
```

- Delete
  - isEmpty( )
  - front와 rear가 같은 index를 가리킨다면 큐는 비어있다고 볼 수 있음

➢ 선입선출 (FIFO : First In First Out)



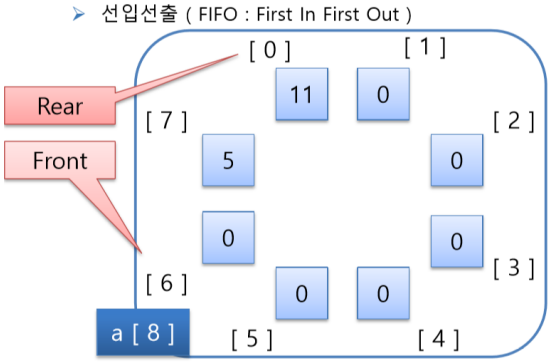

```
public int Delete( ) {
      if ( this.isEmpty() ) {
            return -1;
      }
      if ( front ==
            (INITIAL_LENGTH - 1) ) {
            front = -1;
      }
      return this.a[++front];
}
private boolean isEmpty( ) {
      if ( front == rear ) {
            return true;
      }
      return false;
}
```

➢ Delete

- front의 index 다음의 index가 삭제 될 값
- 주의 할 점은 큐의 index 마지막 번호인 경우 0번으로 가도록 해야 한다는 점

- **size**

➢ 선입선출 (FIFO : First In First Out)



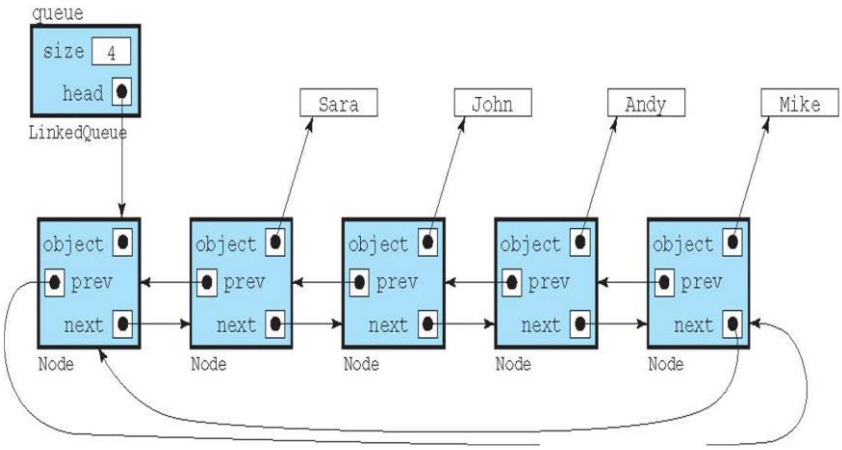

```
public int Size( ) {
      if ( rear < front ) {
            rear += INITIAL_LENGTH;
      }
      return rear - front;
}
```

➢ Size

- 끝(Rear) 에서 시작부분(Front) 를 빼면 큐에 들어있는 원소의 개수를 알 수 있음
- 다만, Rear의 Index가 0인 경우 고려해야 함

Linkedlist로 구현한 큐 – Doubly Linked List

- 장점:
  - 삽입과 삭제를 위한 위치가 항상 동일하게 뒤와 앞이기 때문에 배열 구현보다 구현이 더 빠르다.
  - 제거된 노드가 garbage collector에 의해 자동으로 삭제되기 때문에 공간을 낭비하지 않는다.




Node a

```
public class DoubleLinkedList {
    private Node a;
}
```

```
public class Node {
    public int data;
    public Node next;
    public Node prev;

    public Node(int data) {
        this.data = data;
        next = null;
        prev = null;
    }
}
```

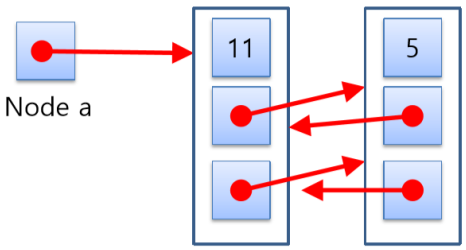
➢ 초기

- Queue로 활용할 노드 생성
- front와 rear는 각각 처음을 가리키는 형태

```
private boolean isEmpty( ) {
    if ( a == null ) {
        return true;
    }
    return false;
}
```

- **Add**

➢ 선입선출 (FIFO : First In First Out)



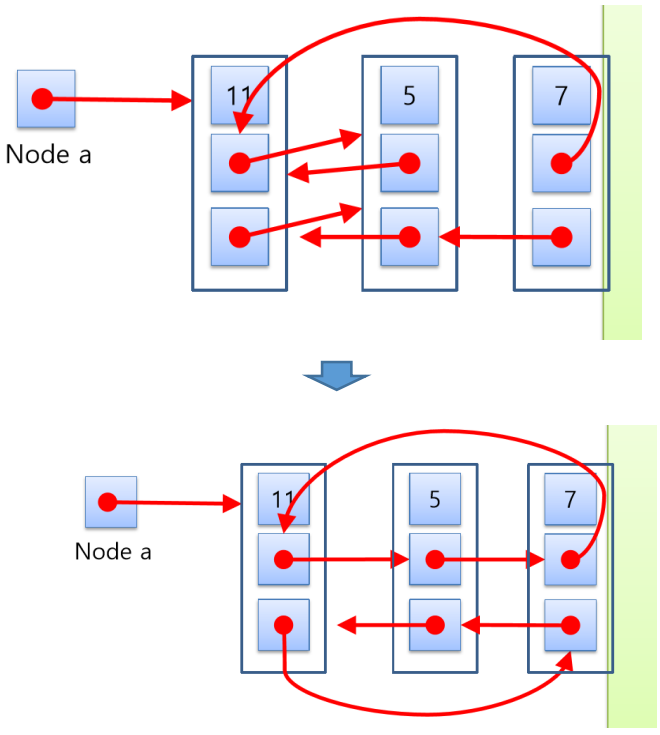

```
public void Add( int item ) {
     Node nw = new Node(item);
     if ( isEmpty( ) ) {
          a = nw;
     }
     else {
          if ( a.next == null ) {
               nw.next = a;
               nw.prev = a;
               a.next = nw;
               a.prev = nw;
          }
          else {

          }
     }
}
```

-

➢ Add

- 처음에 비어있는지 확인 후, 비어 있다면 그냥 바로 추가하면 됨
- 비어 있지 않고 값이 있다면? → 값이 하나인 경우
  값이 하나 이상인 경우


```
public void Add( int item ) {
     Node nw = new Node(item);
     if ( isEmpty( ) ) {
          a = nw;
     }
     else {
          if ( a.next == null ) {
               nw.next = a;
               nw.prev = a;
               a.next = nw;
               a.prev = nw;
          }
          else {
               nw.next = a;
               nw.prev = a.prev;

               nw.prev.next = nw;
               a.prev = nw;
          }
     }
}
```

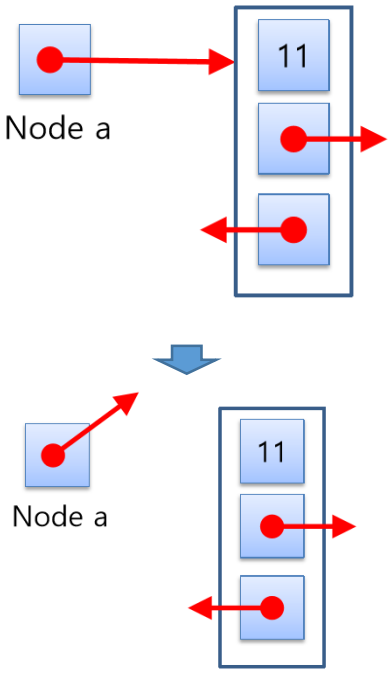


➢ Add

- 처음에 비어있는지 확인 후, 비어 있다면 그냥 바로 추가하면 됨
- 비어 있지 않고 값이 있다면? → 값이 하나인 경우
  값이 하나 이상인 경우

- **Delete**

➢ 선입선출 (FIFO : First In First Out)



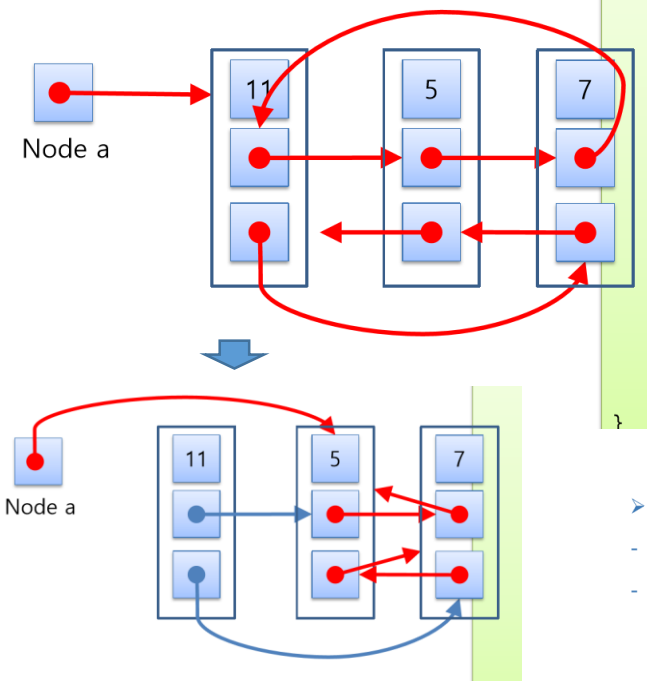

```
public int Delete( ) {
     if ( isEmpty( ) ) {
          return -1;
     }
     else {
          if ( a.next == null ) {
               int rtn = a.data;
               a = null;
               return rtn;
          }
          else {

          }
     }
}
```

➢ Delete

- 처음에 비어있는지 확인 후, 비어 있다면 삭제에 실패
- 비어 있지 않고 값이 있다면? → 값이 하나인 경우 / 값이 하나 이상인 경우

❖ **Doubly Linked List**

➢ 선입선출 (FIFO : First In First Out)



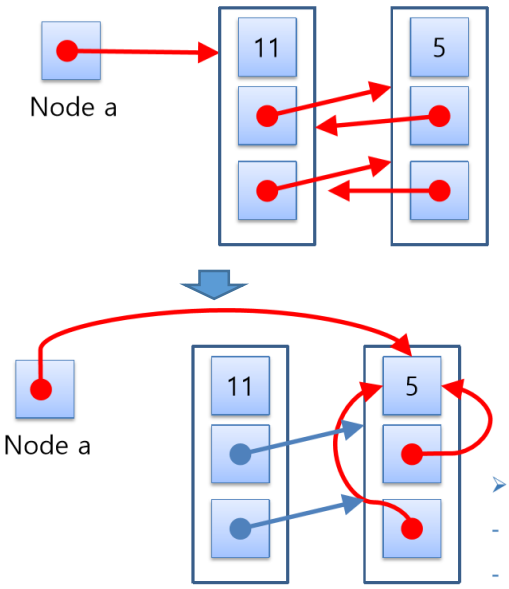

```
public int Delete( ) {
     if ( isEmpty( ) ) {
          return -1;
     }
     else {
          if ( a.next == null ) {
               int rtn = a.data;
               a = null;
               size--;
               return rtn;
          }
          else {
               int rtn = a.data;
               a = a.next;
               a.prev = a.prev.prev;
               a.prev.prev.next = a;
               size--;
               return rtn;
          }
     }
}
```

➢ Delete

- 처음에 비어있는지 확인 후, 비어 있다면 삭제에 실패
- 비어 있지 않고 값이 있다면? → 값이 하나인 경우 / 값이 하나 이상인 경우

❖ Doubly Linked List

➢ 선입선출 (FIFO : First In First Out)



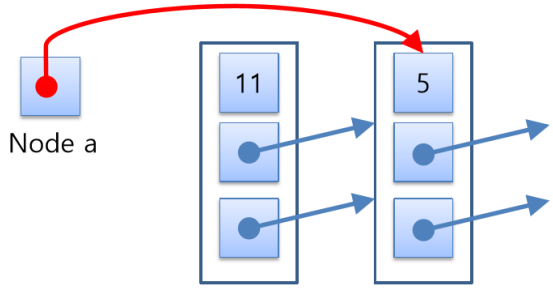

```
public int Delete( ) {
    if ( isEmpty( ) ) {
        return -1;
    }
    else {
        if ( a.next == null ) {
            int rtn = a.data;
            a = null;
            size--;
            return rtn;
        }
        else {
            int rtn = a.data;
            a = a.next;
            a.prev = a.prev.prev;
            a.prev.next = a;
            size--;
            return rtn;
        }
    }
}
```

➢ Delete

- 처음에 비어있는지 확인 후, 비어 있다면 삭제에 실패
- 비어 있지 않고 값이 있다면? → 값이 하나인 경우
  → 값이 하나 이상인 경우 : 값이 2개인 경우

❖ Doubly Linked List

➢ 선입선출 (FIFO : First In First Out)




```
else {
    int rtn = a.data;
    a = a.next;
    if ( a.prev.prev == a ) {
        a.next = null;
        a.prev = null;
        size--;
        return rtn;
    }
    a.prev = a.prev.prev;
    a.prev.next = a;
    size--;
    return rtn;
}
```

➢ Delete

- 처음에 비어있는지 확인 후, 비어 있다면 삭제에 실패
- 비어 있지 않고 값이 있다면? → 값이 하나인 경우
  → 값이 하나 이상인 경우 : 값이 2개인 경우