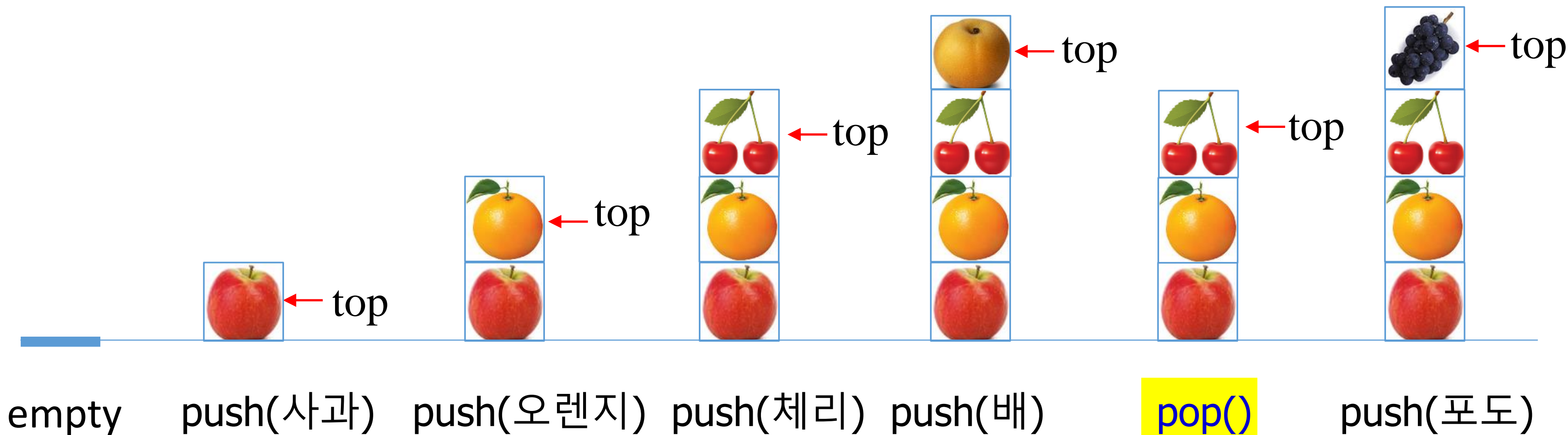
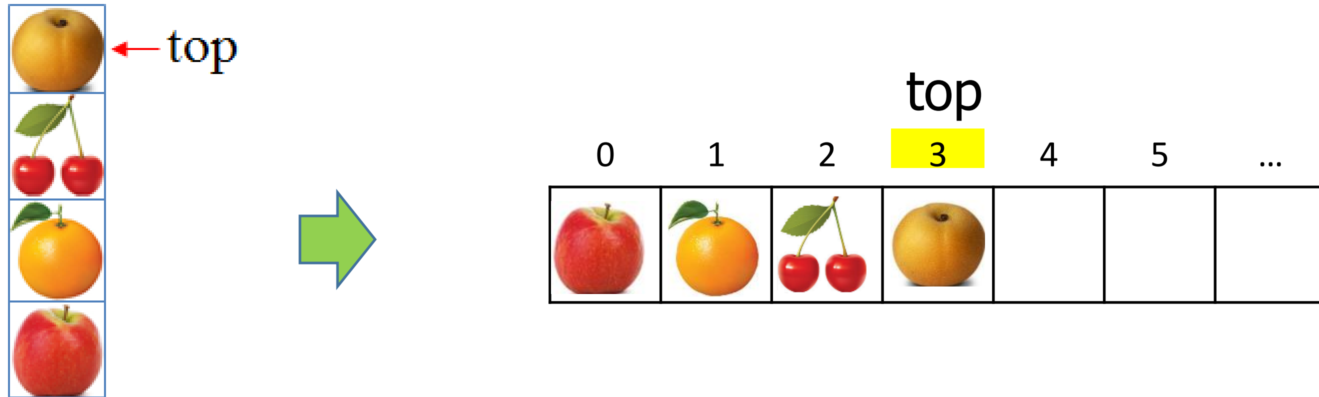
제 3장 스택과 큐

3.1 스택

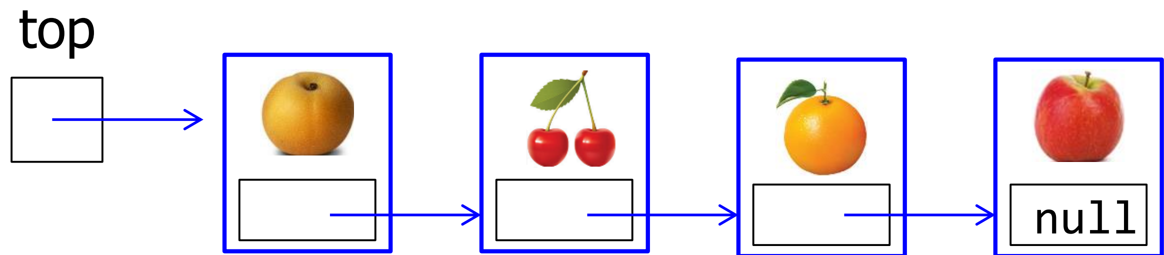
- 한 쪽 끝에서만 item(항목)을 삭제하거나 새로운 item을 저장하는 자료구조
- 새 item을 저장하는 연산: push
- Top item을 삭제하는 연산: pop
- 후입 선출(Last-In First-Out, LIFO) 원칙하에 item의 삽입과 삭제 수행



배열로 구현된 스택



단순연결리스트로 구현된 스택



배열로 구현한 ArrayStack 클래스

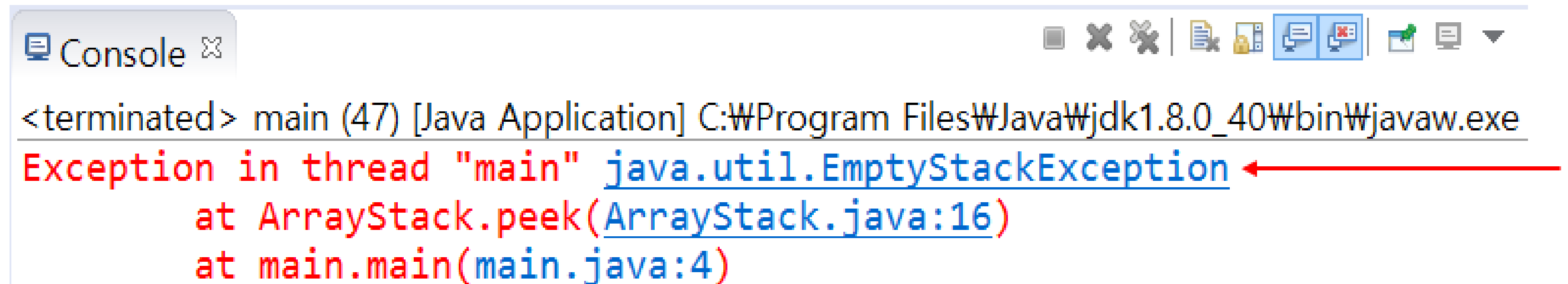
```
01 import java.util.EmptyStackException;
02 public class ArrayStack<E> {
03     private E   s[];          // 스택을 위한 배열
04     private int top;          // 스택의 top 항목의 배열 원소 인덱스
05     public ArrayStack() { // 스택 생성자
06         s = (E[]) new Object[1];  // 초기에 크기가 1인 배열 생성
07         top = -1;
08     }
09     public int     size()    { return top+1;}         // 스택에 있는 항목의 수를 리턴
10     public boolean isEmpty() { return (top == -1);} // 스택이 empty이면 true 리턴

    // peek(), push(), pop(), resize() 메소드 선언

}
```

- Line 01: java.util 라이브러리에 선언된 `EmptyStackException` 클래스를 이용하여 underflow 발생 시 프로그램 종료
- Line 05~08: ArrayStack 클래스의 생성자
  - 크기가 1인 배열 s와 top = -1을 가진 객체 생성
- Line 09: 스택에 있는 item의 수 리턴
- Line 10: 스택이 empty인지를 검사하는 메소드

```
01 public E peek() {  // 스택 top 항목의 내용만을 리턴
02     if (isEmpty()) throw new EmptyStackException(); // underflow 시 프로그램 정지
03     return s[top];
04 }
```

- peek() 메소드: 스택의 top에 있는 item을 리턴
- 만일 스택이 empty일 때에는 EmptyStackException을 발생시켜 예외 발생 에러 메시지 출력 후 프로그램 종료

underflow 발생 시 프로그램 종료



```
01 public void push(E newItem) { // push 연산
02     if (size() == s.length)
03         resize(2*s.length);    // 스택을 2배의 크기로 확장
04     s[++top] = newItem;        // 새 항목을 push
05 }
```

- push() 메소드: 새 item을 스택에 삽입
- overflow가 발생하면, 2.1절에서 선언된 resize() 메소드를 호출하여 배열의 크기를 2배로 확장
- Line 04: top을 1 증가시킨 후에 newItem을 s[top]에 저장

```java
01 public E pop() {  // pop 연산
02     if (isEmpty()) throw new EmptyStackException(); // underflow시 프로그램 정지
03     E item = s[top];
04     s[top--] = null;          // null로 초기화
05     if (size() > 0 && size()==s.length/4)
06         resize(s.length/2);  // 스택을 1/2 크기로 축소
07     return item;
08 }
```

- pop() 메소드: 스택 top item을 삭제한 후 리턴
- Line 04: s[top]을 null로 만들어서 s[top]이 참조하던 객체를 가비지 컬렉션 처리
- s[top]을 null로 만든 이후에는 top을 1 감소
- Line 05~06: 스택의 item 수가 배열 s의 1/4만 차지하면, 메모리 낭비를 줄이기 위해 resize()를 호출하여 배열 s의 크기를 1/2로 축소
  - resize() 실행 이후에 배열 s의 1/2은 item들이 차지하고 나머지 1/2은 비어있게 됨

```java
01 public class main {
02     public static void main(String[] args) {
03         ArrayStack<String> stack = new ArrayStack<String>();
04
05         stack.push("apple");
06         stack.push("orange");
07         stack.push("cherry");
08         System.out.println(stack.peek());
09         stack.push("pear");
10         stack.print();
11         stack.pop();
12         System.out.println(stack.peek());
13         stack.push("grape");
14         stack.print();
15     }
16 }
```

- main 클래스에서 일련의 push와 pop 연산을 수행하고 두 차례 peek 연산을 수행

프로그램의 수행결과

```
Problems  @ Javadoc  Console
<terminated> ArrayStack (1) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
cherry
apple    orange   cherry   pear     <- top
cherry
apple    orange   cherry   grape    <- top
```

단순연결리스트로 구현한 ListStack 클래스

- Node 클래스: 2.2절의 Node 클래스와 동일
- Line 01: 자바 util 라이브러리에 선언된 EmptyStackException 클래스이고, underflow 발생 시 프로그램 정지
- Line 05~08: ListStack 객체를 생성하기 위한 생성자, 객체는 스택 top item을 가진 Node 레퍼런스와 스택의 item 수를 저장하는 size를 가짐
- Line 09~10: 각각 스택의 item 수를 리턴, 스택이 empty이면 true를 리턴하는 메소드

스택의 item을 위한 Node 클래스

```
01 public class Node <E> {
02     private E       item;
03     private Node<E> next;
04     public Node(E newItem, Node<E> node){  // 생성자
05         item = newItem;
06         next = node;
07     }
08     // get과 set 메소드들
09     public E       getItem() { return item; }
10     public Node<E> getNext() { return next; }
11     public void    setItem(E newItem)      { item = newItem; }
12     public void    setNext(Node<E> newNext){ next = newNext; }
13 }
```

```java
01 import java.util.EmptyStackException;
02 public class ListStack <E> {
03     private Node<E> top;    // 스택 top 항목을 가진 Node를 가리키기 위해
04     private int size;       // 스택의 항목 수
05     public ListStack() {    // 스택 생성자
06         top = null;
07         size = 0;
08     }
09     public int     size()    { return size;}       // 스택의 항목의 수를 리턴
10     public boolean isEmpty() { return size == 0;} // 스택이 empty이면 true 리턴
       //peek(), push(), pop() 메소드 선언
   }
```

```java
01 public E peek() {  // 스택 top 항목만을 리턴
02     if (isEmpty()) throw new EmptyStackException(); // underflow 시 프로그램 정지
03     return top.getItem();
04 }
```

- peek() 메소드: 스택의 top item을 리턴
- 스택이 empty인 경우, underflow 가 발생한 것이므로 프로그램 종료

```
01 public void push(E newItem){  // 스택 push 연산
02     Node newNode = new Node(newItem, top);   // 리스트 앞부분에 삽입
03     top = newNode;                           // top이 새 노드 가리킴
04     size++;                                  // 스택 항목 수 1 증가
05 }
```

- push() 메소드: 스택에 새 item을 push하는 메소드
- Line 02: Node 객체를 생성하여 newItem을 newNode에 저장하고, top이 가진 레퍼런스를next에 복사
- 이후 top이 새 Node를 가리키도록
- 즉, 새 노드를 항상 연결리스트의 가장 앞에 삽입
- Line 04: 스택의 item 수인 size 1 증가

```
01 public E pop() {    // 스택 pop연산
02     if (isEmpty()) throw new EmptyStackException(); // underflow 시 프로그램 정지
03     E topItem = top.getItem();    // 스택 top 항목을 topItem에 저장
04     top = top.getNext();          // top이 top 바로 아래 항목을 가리킴
05     size--;                       // 스택 항목 수를 1 감소
06     return topItem;
07 }
```

- pop() 메소드: 스택이 empty가 아닐 때, top이 가리키는 노드의 item을 topItem에 저장한 뒤 line 06에서 이를 리턴
- Line 04: top을 top이 참조하던 노드를 가리키게
- Line 05: size 1 감소

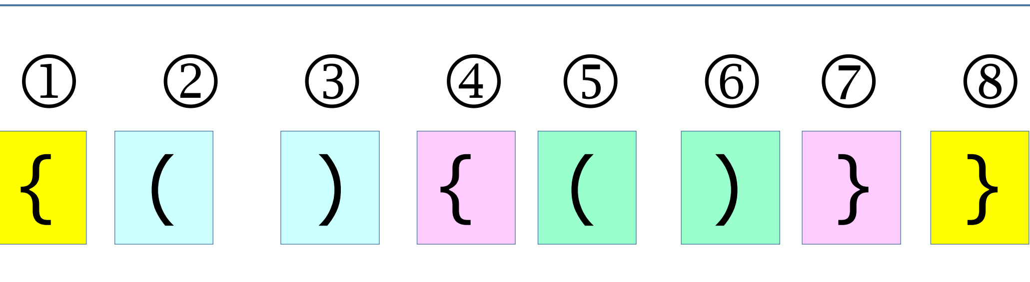
프로그램의 수행결과

```
Problems  @ Javadoc  Console
<terminated> ListStack (1) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
cherry
pear      cherry  orange  apple
cherry
grape     cherry  orange  apple
```

top → pear

top → grape

[예제 1]



③ (⇔)

⑥ (⇔)

⑦ { ⇔ }

① {

② ({

{

④ { {

⑤ ({ {

{ {

{

mpty push({) push(() pop() push({) push(() pop() pop()

스택의 응용

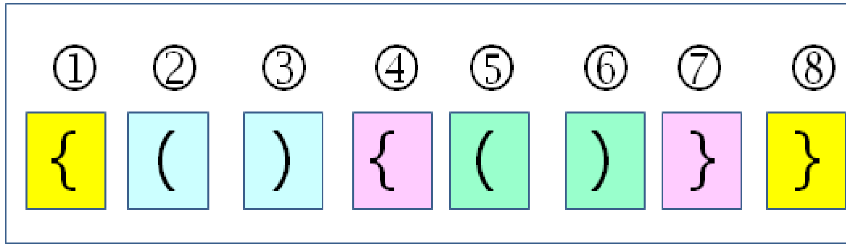
- 컴파일러의 괄호 짝 맞추기
- 회문(Palindrome) 검사하기
- 후위표기법(Postfix Notation) 수식 계산하기
- 중위표기법(Infix Notation) 수식의 후위표기법 변환

컴파일러의 괄호 짝 맞추기

[핵심 아이디어] 왼쪽 괄호는 스택에 push, 오른쪽 괄호를 읽으면 pop 수행

- pop된 왼쪽 괄호와 바로 읽었던 오른쪽 괄호가 다른 종류이면 에러 처리, 같은 종류이면 다음 괄호를 읽음
- 모든 괄호를 읽은 뒤 에러가 없고 스택이 empty이면, 괄호들이 정상적으로 사용된 것
- 만일 모든 괄호를 처리한 후 스택이 empty가 아니면 짝이 맞지 않는 괄호가 스택에 남은 것이므로 에러 처리

[예제]



① {

② ({

③ (⇔) {

④ { {

⑤ ({ {

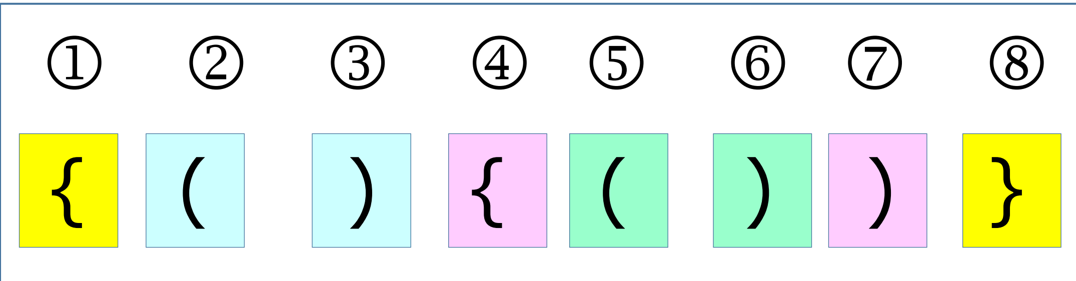
⑥ (⇔) { {

⑦ { ⇔ } {

⑧ { ⇔ }

empty push({) push(() pop() push({) push(() pop() pop() pop()

[예제]



mpty push({) push(() pop() push({) push(() pop() pop()

회문 검사하기

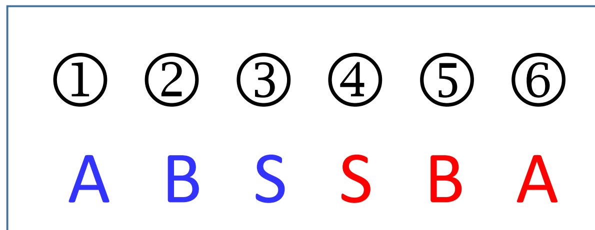
- 회문(Palindrome): 앞으로부터 읽으나 뒤로부터 읽으나 동일한 스트링

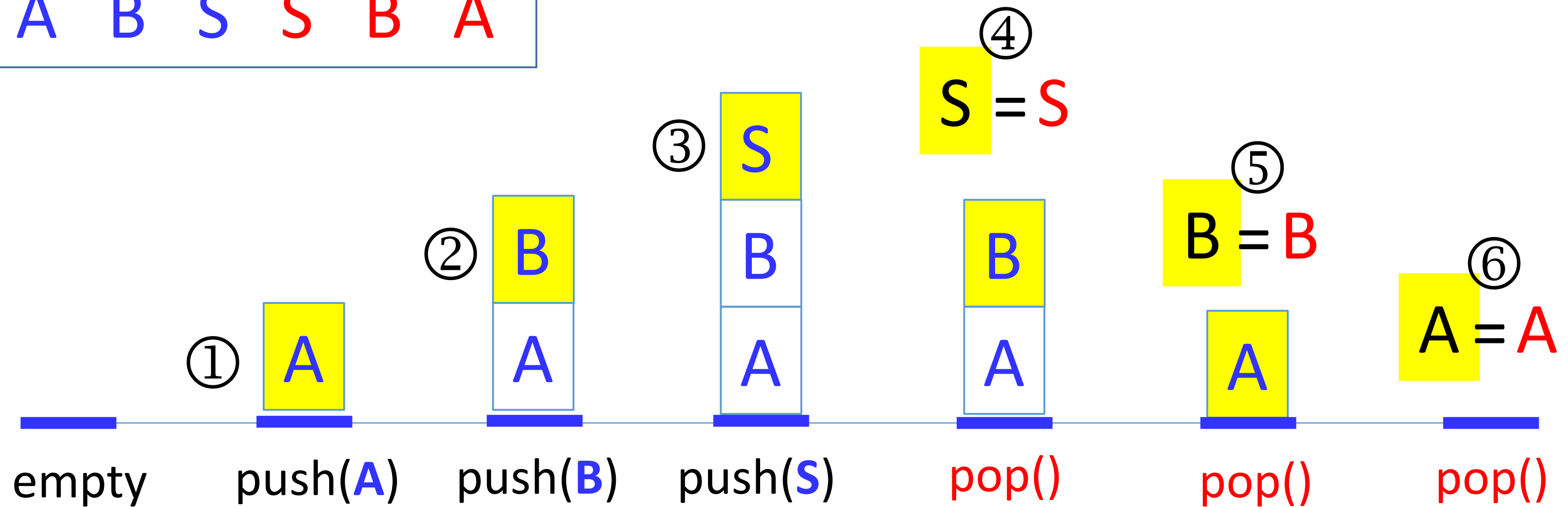
[핵심 아이디어] 전반부의 문자들을 스택에 push한 후, 후반부의 각 문자를 차례로 pop한 문자와 비교

- 회문 검사하기는 주어진 스트링의 앞부분 반을 차례대로 읽어 스택에 push한 후, 문자열의 길이가 짝수이면 뒷부분의 문자 1 개를 읽을 때마다 pop하여 읽어 들인 문자와 pop된 문자를 비교하는 과정을 반복 수행
- 만약 마지막 비교까지 두 문자가 동일하고 스택이 empty가 되면, 입력 문자열은 회문

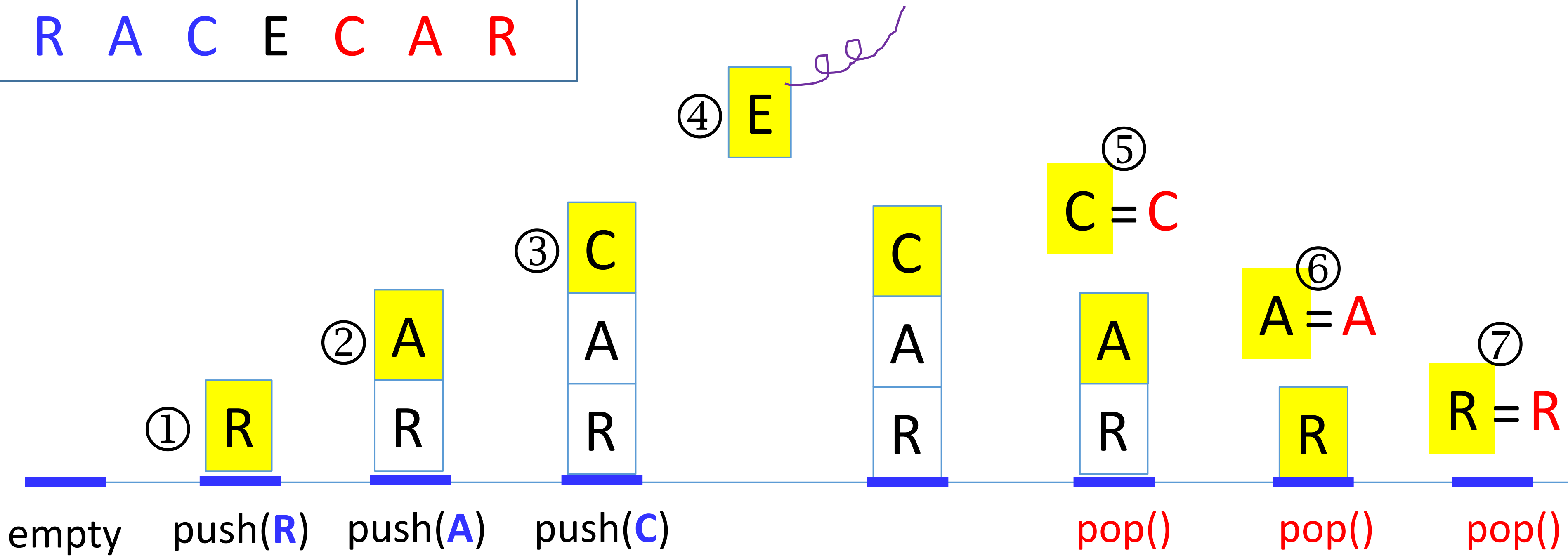
- 문자열의 길이가 홀수인 경우, 주어진 스트링의 앞부분 반을 차례로 읽어 스택에 push한 후, 중간 문자를 읽고 버린다. 이후 짝수 경우와 동일하게 비교 수행

[예제]





[예제]



수식의 표기법

- 프로그램을 작성할 때 수식에서 +, −, *, /와 같은 이항연산자는 2개의 피연산자들 사이에 위치
- 이러한 방식의 수식 표현이 중위표기법(Infix Notation)
- 컴파일러는 중위표기법 수식을 후위표기법(Postfix Notation)으로 바꾼다.
  - 그 이유는 후위표기법 수식은 괄호 없이 중위표기법 수식을 표현할 수 있기 때문
- 전위표기법(Prefix Notation): 연산자를 피연산자들 앞에 두는 표기법

중위표기법 수식과 대응되는 후위표기법, 전위표기법 수식

중위표기법	후위표기법	전위표기법
A + B	A B +	+ A B
A + B – C	A B + C –	+ A – B C
A + B * C – D	A B C * + D –	– + A * B C D
(A + B) / (C – D)	A B+ C D – /	/ + A B – C D

후위표기법 수식 계산

- [핵심 아이디어] 피연산자는 스택에 push하고, 연산자는 2회 pop하여 계산한 후 push

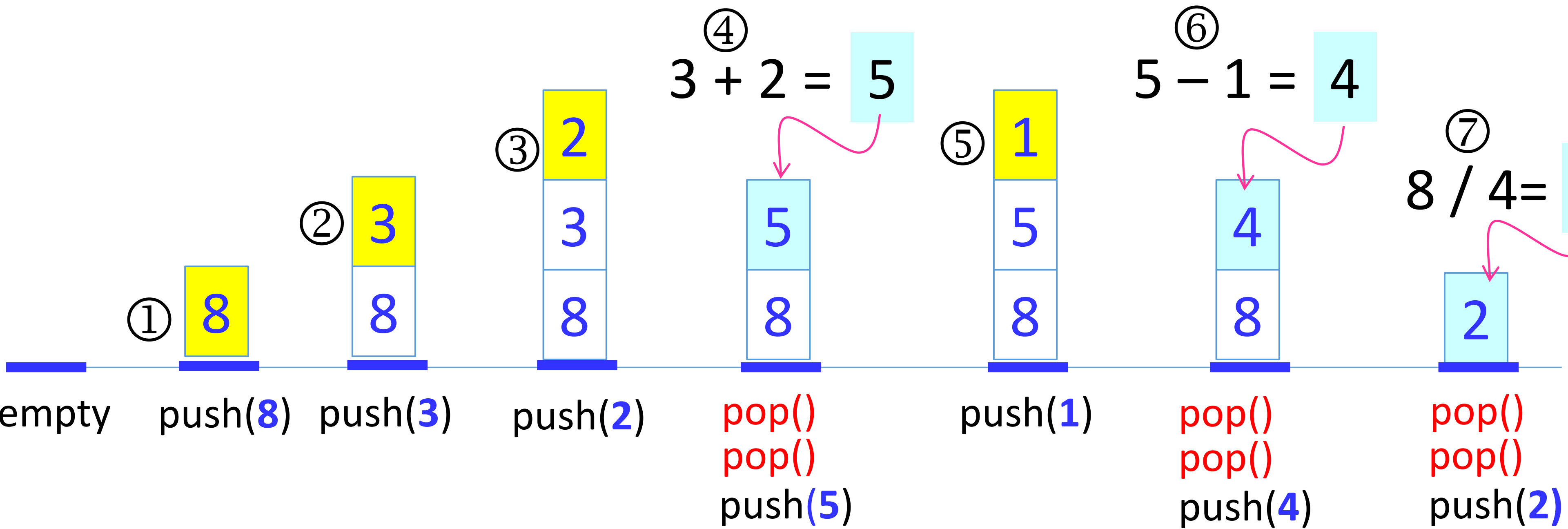
후위표기법으로 표현된 수식 계산 알고리즘

- 입력을 좌에서 우로 문자를 한 개씩 읽는다. 읽은 문자를 C라고하면

  [1] C가 피연산자이면 스택에 push

  [2] C가 연산자(op)이면 pop을 2회 수행한다. 먼저 pop된 피연산자가 A이고, 나중에 pop된 피연산자가 B라면, (A op B)를 수행하여 그 결과 값을 push

[예제]



중위표기법 수식을 후위표기법으로 변환

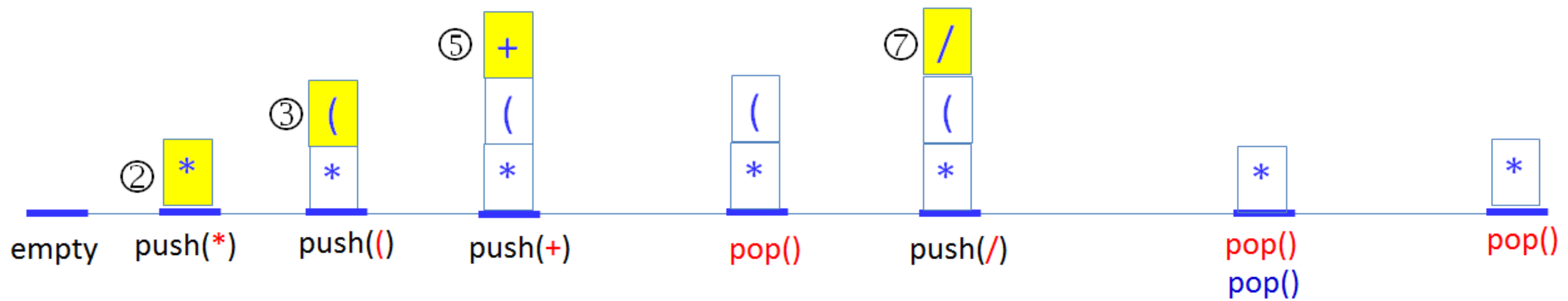
- [핵심 아이디어] 왼쪽 괄호나 연산자는 스택에 push하고, 피연산자는 출력

- 입력을 좌에서 우로 문자를 1개씩 읽는다. 읽은 문자가
  1. 피연산자이면, 읽은 문자를 출력
  2. 왼쪽 괄호이면, push
  3. 오른쪽 괄호이면, 왼쪽 괄호가 나올 때까지 pop하여 출력. 단, 오른쪽이나 왼쪽 괄호는 출력하지 않음
  4. 연산자이면, 자신의 우선순위보다 낮은 연산자가 스택 top에 올 때까지 pop하여 출력하고 읽은 연산자를 push
- 입력을 모두 읽었으면 스택이 empty될 때까지 pop하여 출력

[예 제]

①	②	③	④	⑤	⑥	⑦	⑧	⑨
A	*	(	B	+	C	/	D	)

출력: A ①   A B ④   A B C ⑥   A B C + ⑦   A B C + D ⑧   A B C + D / ⑨   A B C + D / *



스택 자료구조의 응용

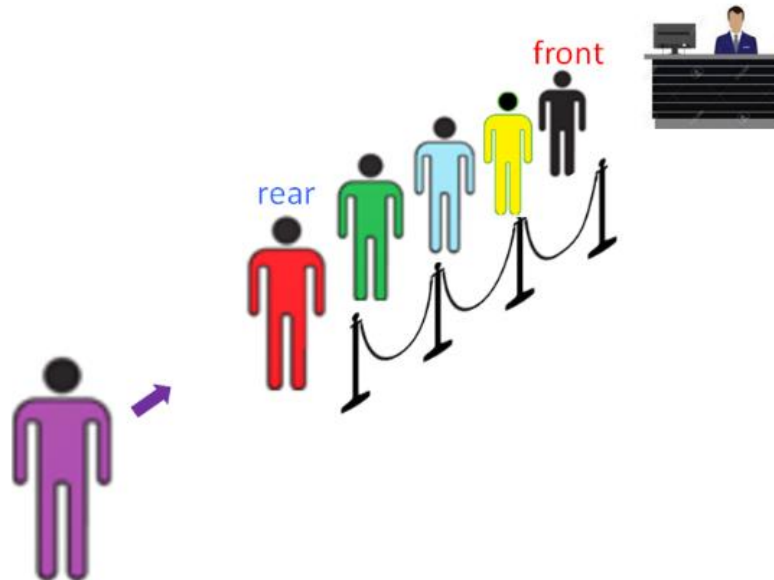
- 미로 찾기
- 트리의 방문(4장)
- 그래프의 깊이우선탐색(9장)
- 프로그래밍에서 매우 중요한 함수(메소드) 호출 및 재귀호출도 스택 자료구조를 바탕으로 구현

수행시간

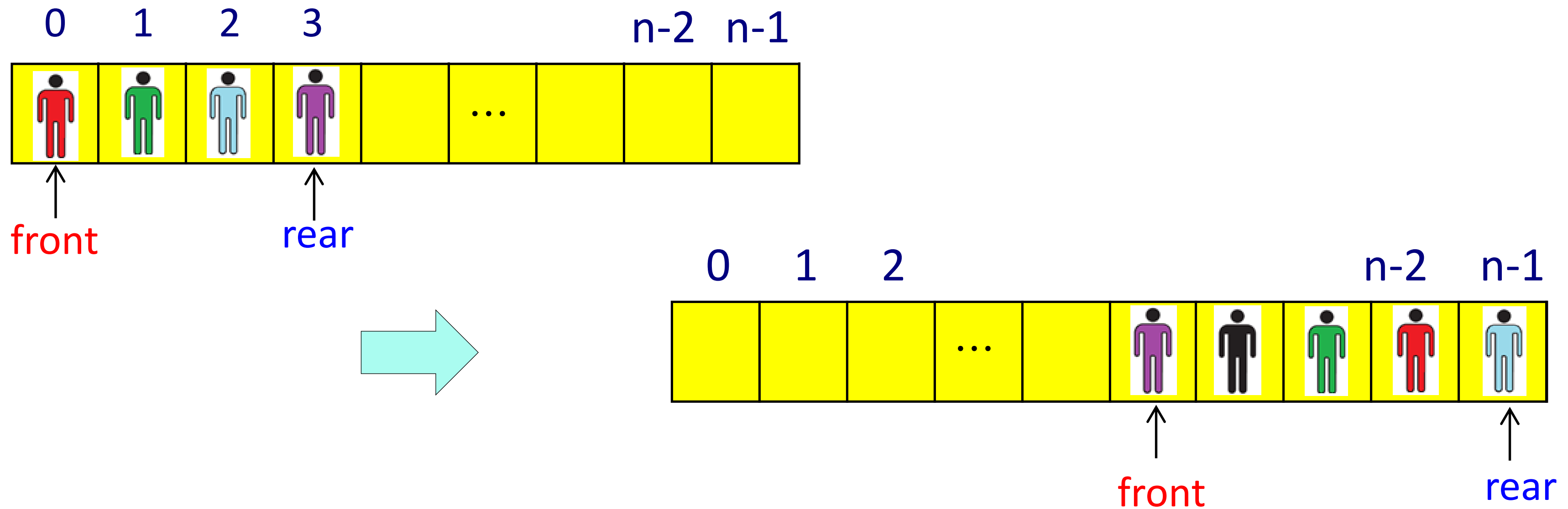
- 배열로 구현한 스택의 push와 pop 연산은 각각 O(1) 시간이 소요
- 배열 크기를 확대 또는 축소시키는 경우에 스택의 모든 item들을 새 배열로 복사해야 하므로 O(N) 시간이 소요
  - 상각분석: 각 연산의 평균 수행시간은 O(1) 시간
- 단순연결리스트로 구현한 스택의 push와 pop 연산은 각각 O(1) 시간이 걸리는데, 연결리스트의 앞 부분에서 노드를 삽입하거나 삭제하기 때문
- 배열과 단순연결리스트로 구현된 스택의 장단점은 2장의 리스트를 배열과 단순연결리스트로 구현하였을 때의 장단점과 동일

3.2 큐

- 큐(Queue): 삽입과 삭제가 양 끝에서 각각 수행되는 자료구조
- 일상생활의 관공서, 은행, 우체국, 병원 등에서 번호표를 이용한 줄서기가 대표적인 큐
- 선입 선출(First-In First-Out, FIFO) 원칙하에 item의 삽입과 삭제 수행

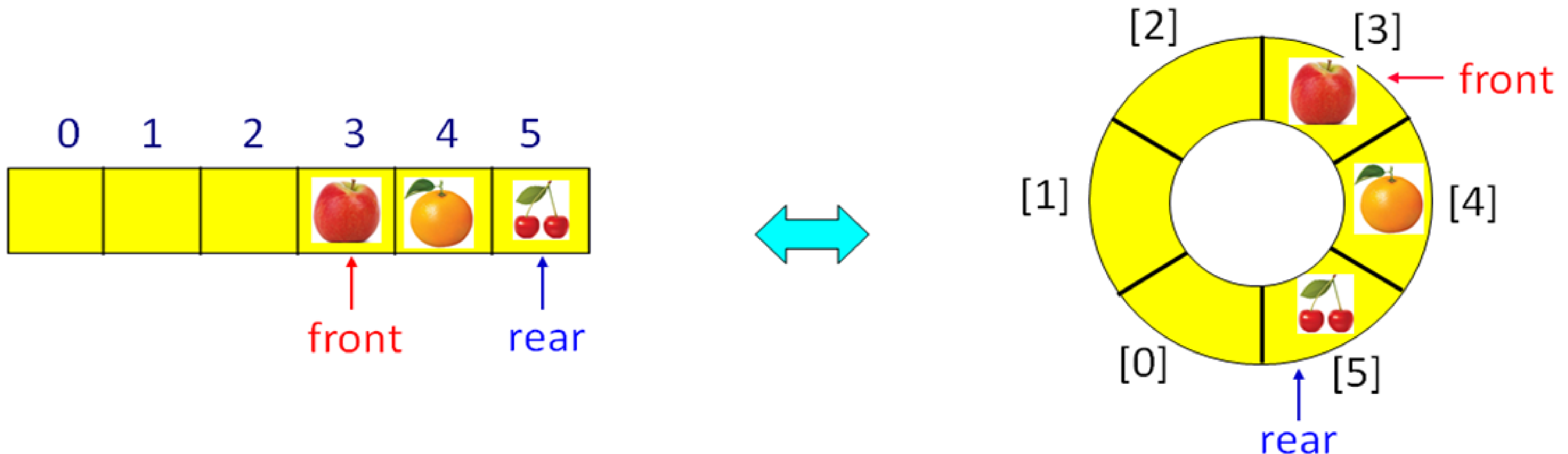


- 큐를 배열로 구현하는 경우, 큐에서 삽입과 삭제를 거듭하게 되면 그림과 같이 큐의 item들이 배열의 오른쪽 부분으로 편중되는 문제가 발생
- 왜냐하면 새 item들은 뒤에 삽입되고 삭제는 앞에서 일어나기 때문

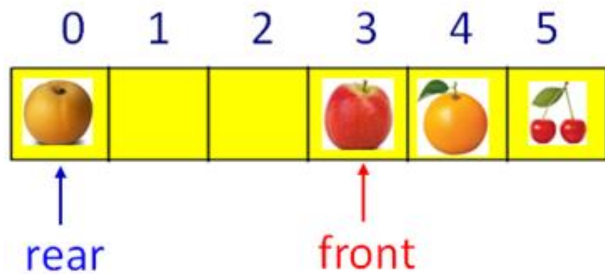


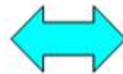
항목 이동 해결 방안

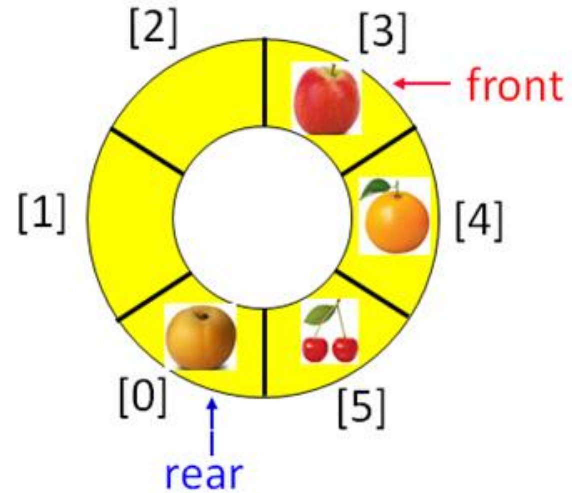
- [방법 1] 큐의 item들을 배열의 앞부분으로 이동.
  - 수행시간이 큐에 들어있는 item 의 수에 비례하는 단점
- [방법 2] 배열을 원형으로, 즉, 배열의 마지막 원소가 첫 원소와 맞닿아 있다고 여김



새 item 삽입 후





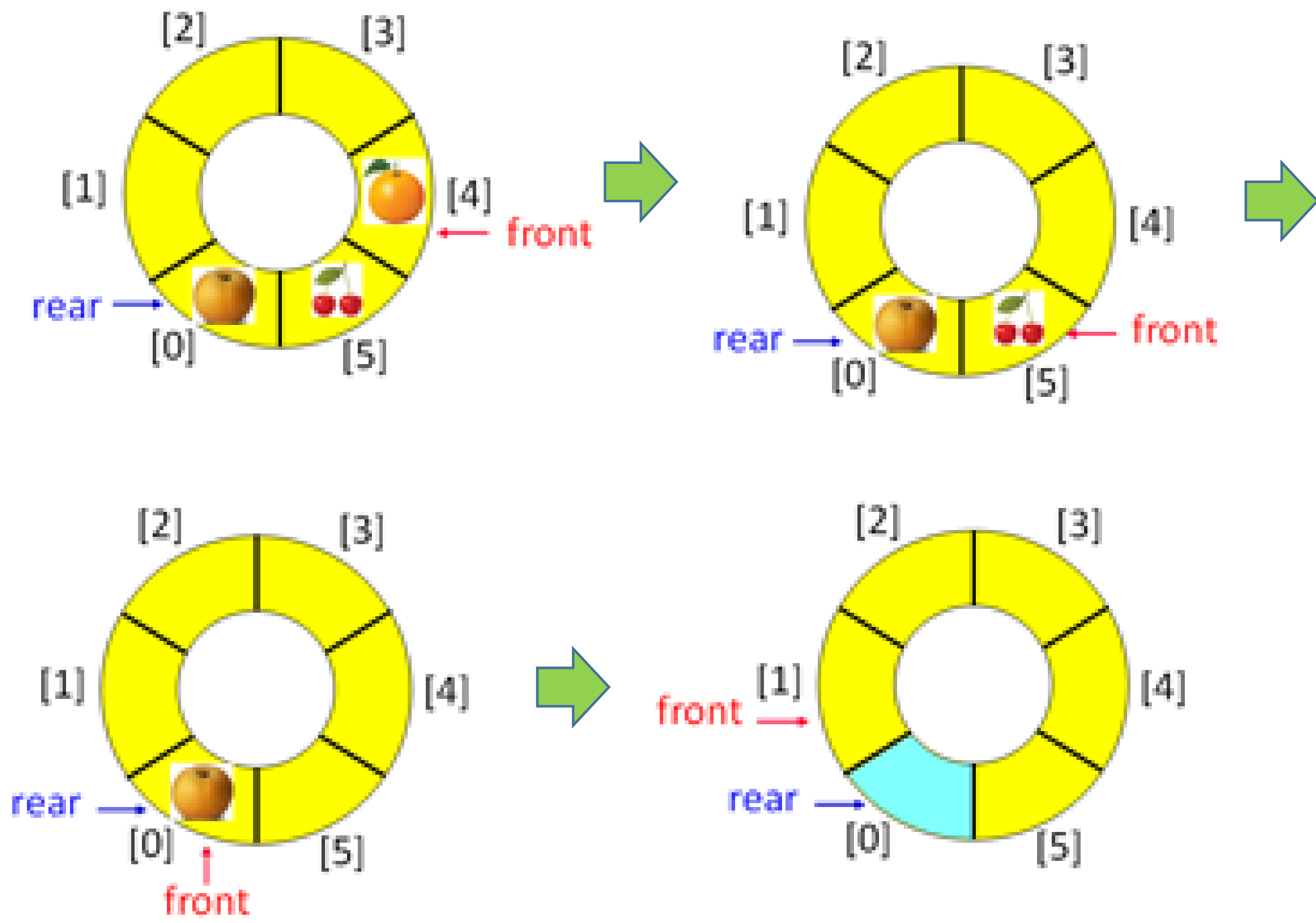


- 배열의 앞뒤가 맞닿아 있다고 생각하기 위해 배열의 rear 다음의 비어있는 원소의 인덱스

$$rear = (rear + 1) \% N$$

- 여기서 N은 배열의 크기이다.

연속된 삭제 연산 수행



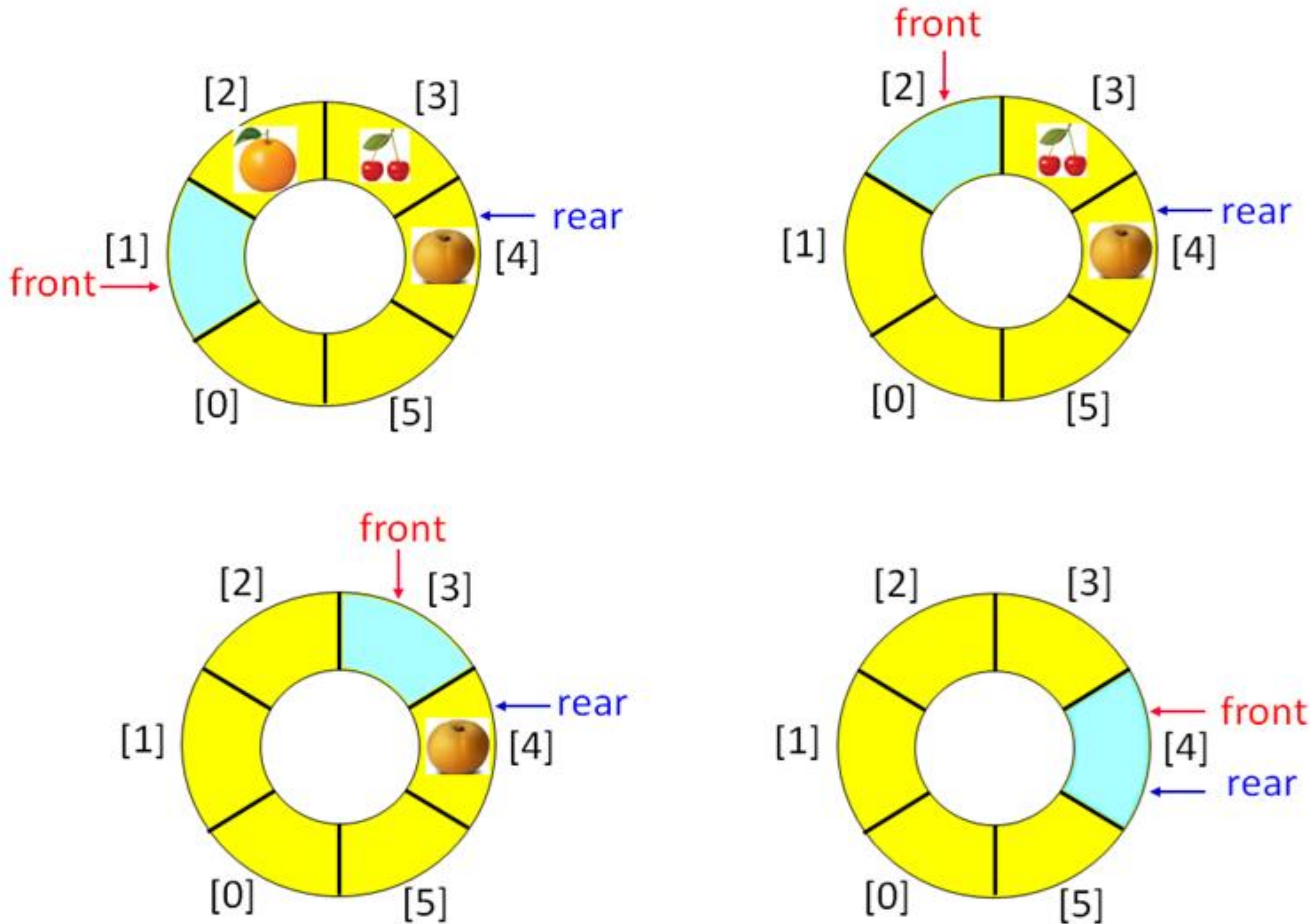
큐의 마지막 item을 삭제한 후에
큐가empty임에도 불구하고 rear는 삭제된
item을 아직도 가리키고 있다.

큐가 empty일 때 문제 해결 방안

- [방법 1] item을 삭제할 때마다 큐가 empty가 되는지 검사하고, 만일 empty가 되면, front = rear = 0을 만든다.
  - 삭제할 때마다 empty 조건을 검사하는 것은 프로그램 수행의 효율성이 저하됨

- [방법 2] front를 실제의 가장 앞에 있는 item의 바로 앞의 비어있는 원소를 가리키게 한다.
  - 배열의 크기가 N이라면 실제로 N-1개의 공간만 item들을 저장하는데 사용
  - [방법 1]에서 item을 삭제할 때마다 조건을 한번 더 검사하는 것은 '이론적인' 수행시간을 증가시키지 않으나 일반적으로 프로그램이 수행될 때 조건을 검사하는 프로그램의 실제 실행시간은 검사하지 않는 프로그램보다 더 오래 소요됨

연속된 삭제 연산 수행



Empty가 되면 front == rear가 된다.

큐를 배열로 구현한 ArrayQueue 클래스

```
01 import java.util.NoSuchElementException;
02 public class ArrayQueue <E>{
03     private E[] q;    // 큐을 위한 배열
04     private int front, rear, size;
05     public ArrayQueue() {          // 큐 생성자
06         q = (E[]) new Object[2]; // 초기에 크기가 2인 배열 생성
07         front = rear = size = 0;
08     }
09     public int     size()    { return size;}          // 큐에 있는 항목의 수를 리턴
10     public boolean isEmpty() { return (size == 0);} // 큐가 empty이면 true를 리턴

    // add(), remove(), resize() 메소드 선언
}
```

- Line 01: 큐에서 underflow가 발생했을 때 java 라이브러리에 선언된 `NoSuchElementException`을 이용하여 프로그램 정지
- 참고로 `EmptyQueueException`은 자바 라이브러리에 선언되어 있지 않음

```java
44     public static void main(String[] args) {
45         ListQueue<String> q = new ListQueue<String>();
46             q.remove();
47             q.add("apple");     q.add("orange");
48             q.add("cherry");    q.add("pear");
49             q.print();
50 
51             q.remove(); q.print();
52             q.remove(); q.print();
53 
54             q.add("grape");     q.print();
55     }
56 }
```

Problems @ Javadoc Console

```
<terminated> ListQueue [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
Exception in thread "main" java.util.NoSuchElementException
        at ListQueue.remove(ListQueue.java:23)
        at ListQueue.main(ListQueue.java:46)
```

- Line 05~08: ArrayQueue 객체 생성자로, 객체는 1차원 배열 q와 3개의 필드인 front, rear, size를 가짐
- 초기의 배열 크기는 2이고, 각 필드를 0으로 초기화
- Line 09: 큐의 item 수를 리턴하는 메소드
- Line 10: 큐가 empty이면 true를 리턴하는 메소드

```
01 public void add(E newItem) {    // 큐 삽입 연산
02     if ((rear+1)%q.length == front) // 비어있는 원소가 1개뿐인 경우(즉, 큐가 full인 경우)
03         resize(2*q.length);     // 큐의 크기를 2배로 확장
04     rear = (rear+1) % q.length;
05     q[rear] = newItem;          // 새 항목을 add
06     size++;
07 }
```

- add() 메소드: 큐에 새 item을 삽입
- Line 02: 삽입할 빈 자리가 있는지 확인한다. 만일 (rear+1)%q.length (즉, rear 다음 원소의 인덱스)가 front와 같으면 overflow 가 발생한 것이므로, resize() 메소드를 호출하여 배열을 2배 크기로 확장
- Line 04: rear를 (rear+1)%q.length로 갱신한 후, line 05에서 newItem을 q[rear]에 저장
- Line 06: size 1 증가

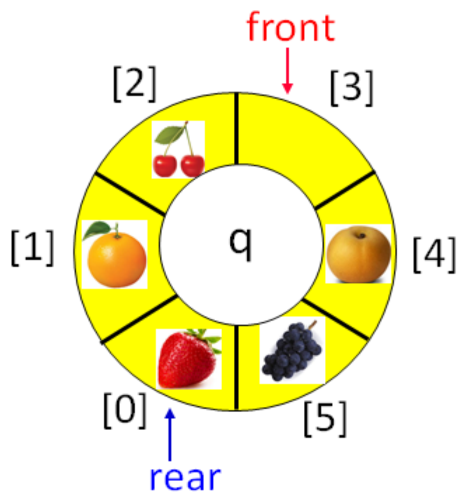
```java
01 public E remove() {  //큐 삭제 연산
02     if (isEmpty()) throw new NoSuchElementException(); // underflow 경우프로그램 정지
03     front  = (front+1) % q.length;
04     E item = q[front];
05     q[front] = null;    // null로 만들어 가비지 컬렉션되도록
06     size--;
07     if (size > 0 && size == q.length/4) // 큐의 항목수가  배열 크기의 1/4가 되면
08         resize(q.length/2);              // 큐를 1/2 크기로 축소
09     return item;
10 }
```

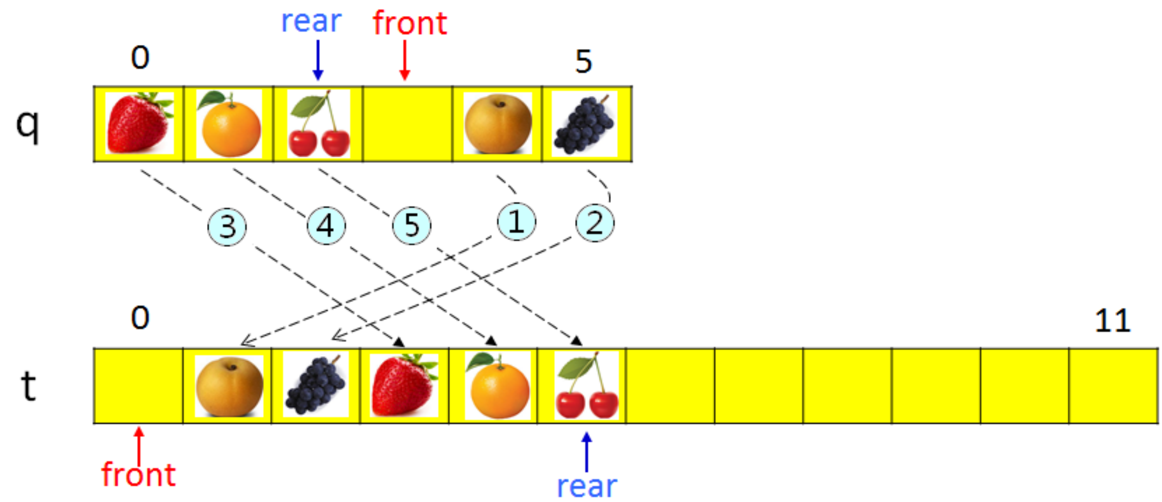
- remove() 메소드: 큐에서 item을 삭제
- Line 02: underflow를 체크하고 underflow 발생시 NoSuchElementException을 throw하여 프로그램 정지
- front를 (front+1)%q.length로 갱신한 후, q[front]를 변수 item에 저장하여 line 09에서 리턴
- Line 05~06: q[front]를 null로 만들어 가비지 컬렉션이 되도록 하고, size를 1 감소
- Line 07~08: 삭제 후 배열에 1/4만큼만 item들로 채워져 있으면 배열 크기를 1/2로 감소시키기 위해 resize() 메소드 호출

```
01 private void resize(int newSize) {     // 큐리 배열 크기 조절
02     Object[] t = new Object[newSize]; // newSize 크기의 새로운 배열 t 생성
03     for(int i = 1, j=front+1; i <size+1; i ++, j++){
04         t[i] = q[j%q.length];          // 배열q의 항목들을 배열 t[1]로부터 복사
05     }
06     front = 0;
07     rear  = size;
08     q = (E[]) t;          // 배열 t를 배열 q로
09 }
```

- resize() 메소드 2.1절의 resize()와 거의 동일
- Line 06~07: front를 0으로 rear는 큐에 있는 item의 수인 size로 갱신하고,
- Line 08: q가 새 배열 t를 참조

2배로 확장시킨 큐





Main 클래스

```
01 public class main {
02     public static void main(String[] args) {
03         ArrayQueue<String> queue = new ArrayQueue<String>();
04         queue.add("apple");    queue.add("orange");
05         queue.add("cherry");   queue.add("pear");     queue.print();
06         queue.remove();        queue.print();
07         queue.add("grape");    queue.print();
08         queue.remove();        queue.print();
09         queue.add("lemon");    queue.print();
10         queue.add("mango");    queue.print();
11         queue.add("lime");     queue.print();
12         queue.add("kiwi");     queue.print();
13         queue.remove();        queue.print();
14     }
15 }
```

프로그램의 수행 결과

Problems @ Javadoc Console Console

```
<terminated> ArrayQueue [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
null   apple  orange  cherry  pear   null   null   null
null   null   orange  cherry  pear   null   null   null
null   null   orange  cherry  pear   grape  null   null
null   null   null    cherry  pear   grape  null   null
null   null   null    cherry  pear   grape  lemon  null
null   null   null    cherry  pear   grape  lemon  mango
lime   null   null    cherry  pear   grape  lemon  mango
lime   kiwi   null    cherry  pear   grape  lemon  mango
lime   kiwi   null    null    pear   grape  lemon  mango
```

front

rear

큐를 연결리스트로 구현한 ListQueue 클래스

```
01 import java.util.NoSuchElementException;
02 public class ListQueue <E> {
03     private Node<E> front, rear;
04     private int size;
05     public ListQueue() {   // 생성자
06         front = rear = null;
07         size  = 0;
08     }
09     public int     size()    { return size; }         // 큐의 항목의 수를 리턴
10     public boolean isEmpty() { return size() == 0; } // 큐가 empty이면 true리턴
       // add(), remove() 메소드 선언
   }
```

- Node 클래스:2.2절의 Node 클래스와 동일
- Line 01: java.util 라이브러리에 선언되어 있는 NoSuchElementException 클래스이고, underflow 발생 시 프로그램 종료

- Line 05~08: ListQueue 객체의 생성자, ListQueue객체는 큐의 첫 item을 가진Node 레퍼런스를 저장하는 front와 마지막 Node를 가리키는 rear, 큐의 item 수를 저장하는 size를 가진다.
- Line 09~10: 각각 스택의 item 수를 리턴하고, 스택이 empty이면 true를 리턴하는 메소드

```
01 public void add(E newItem){
02     Node newNode = new Node(newItem, null);       // 새 노드 생성
03     if (isEmpty()) front = newNode; // 큐가 empty이었으면 front도 newNode를 가리키게 한다
04     else rear.setNext(newNode);     // 그렇지않으면 rear의 next를 newNode를 가리키게 한다
05     rear = newNode;                 // 마지막으로 rear가 newNode를 가리키게 한다
06     size++;                         // 큐 항목 수 1 증가
07 }
```

- add() 메소드: 새 item을 큐의 뒤(rear)에 삽입
- Line 02: 노드를 생성
- Line 03: 연결리스트가 empty인 경우 front가 새 노드를 가리키도록
- 연결 리스트가 empty가 아닌 경우 line 04에서 rear가 참조하는 노드의 next 가 새 노드(newNode)를 가리키도록 하여 새 노드를 연결리스트의 마지막 노드로 연결
- Line 05: rear가 새 노드를 가리키게 하고, line 06에서 size 1 증가

```
01 public E remove() {
02     if (isEmpty()) throw new NoSuchElementException(); // underflow 경우에 프로그램 정지
03     E frontItem = front.getItem();   // front가 가리키는 노드의 항목을 frontItem에 저장
04     front = front.getNext();         // front가 front 다음 노드를 가리키게 한다.
05     if (isEmpty()) rear = null;      // 큐가 empty이면 rear = null
06     size--;                          // 큐 항목 수 1 감소
07     return frontItem;
08 }
```

- remove() 메소드: item을 큐의 앞(front)에서 삭제
- Line 02: underflow를 검사
- Line 03: front가 가리키는 노드의 item을 line 07에서 리턴
- Line 05: 삭제 후 연결리스트가 empty가 된 경우 rear를 null로 갱신
- Line 06: size 1 감소

프로그램의 수행 결과

```
<terminated> ListQueue [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
apple    orange   cherry   pear
orange   cherry   pear
cherry   pear
cherry   pear     grape
```

rear

front

큐 자료구조의 응용

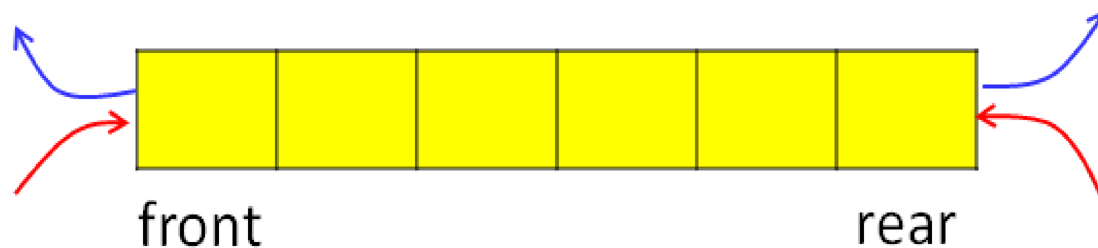
- CPU의 태스크 스케줄링(Task Scheduling)
- 네트워크 프린터
- 실시간(Real-time) 시스템의 인터럽트(Interrupt) 처리
- 다양한 이벤트 구동 방식(Event-driven) 컴퓨터 시뮬레이션
- 콜 센터의 전화 서비스 처리 등
- 4장의 이진트리의 레벨순서 순회(Level-order Traversal)
- 9장의 그래프에서 너비우선탐색(Breath-First Search) 등

수행시간

- 배열로 구현한 큐의 add와 remove 연산은 각각 O(1) 시간이 소요
- 배열 크기를 확대 또는 축소시키는 경우에 큐의 모든 item들을 새 배열로 복사해야 하므로 O(N) 시간이 소요. 상각분석: 각 연산의 평균 수행시간은 O(1)
- 단순연결리스트로 구현한 큐의add와 remove 연산은 각각 O(1) 시간, 삽입 또는 삭제 연산이 rear 와 front로 인해 연결리스트의 다른 노드들을 일일이 방문할 필요 없기 때문
- 배열과 단순연결리스트로 구현한 큐의 장단점은 2장의 리스트를 배열과 단순연결리스트로 구현하였을 때의 장단점과 동일

3.3 데크

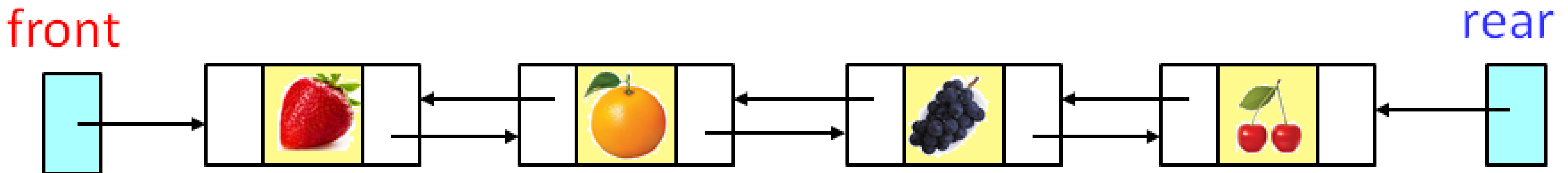
- 데크(Double-ended Queue, Deque): 양쪽 끝에서 삽입과 삭제를 허용하는 자료구조
- 데크는 스택과 큐 자료구조를 혼합한 자료구조
- 따라서 데크는 스택과 큐를 동시에 구현하는데 사용



응용

- 스크롤(Scroll)
- 문서 편집기 등의 undo 연산
- 웹 브라우저의 방문 기록 등
  - 웹 브라우저 방문 기록의 경우, 최근 방문한 웹 페이지 주소는 앞에 삽입하고, 일정 수의 새 주소들이 앞쪽에서 삽입되면 뒤에서 삭제가 수행

- 데크를 이중연결리스트로 구현하는 것이 편리
- 단순연결리스트는 rear가 가리키는 노드의 이전 노드의 레퍼런스를 알아야 삭제가 가능하기 때문



수행시간

- 데크를 배열이나 이중연결리스트로 구현한 경우, 스택과 큐의 수행시간과 동일
- 양 끝에서 삽입과 삭제가 가능하므로 프로그램이 다소 복잡
- 이중연결리스트로 구현한 경우는 더 복잡함
- 자바 SE 7은 java.util 패키지에서 Deque 인터페이스를 제공, Queue 클래스에서 상속됨

요약

- 스택은 한 쪽 끝에서만 item을 삭제하거나 새로운 item을 저장하는 후입선출(LIFO) 자료구조
- 스택은 컴파일러의 괄호 짝 맞추기, 회문 검사하기, 후위표기법수식 계산하기, 중위표기법 수식을 후위표기법으로 변환하기, 미로 찾기, 트리의 노드 방문, 그래프의 깊이우선탐색에 사용. 또한 프로그래밍에서 매우 중요한 메소드 호출 및 재귀호출도 스택 자료구조를 바탕으로 구현
- 큐는 삽입과 삭제가 양 끝에서 각각 수행되는 선입선출(FIFO) 자료구조

- 배열로 구현된 큐에서 삽입과 삭제를 거듭하게 되면 큐의 item들이 오른쪽으로 편중되는 문제가 발생한다. 이를 해결하기 위한 방법은 배열을 원형으로, 즉, 배열의 마지막 원소가 첫 원소와 맞닿아 있다고 생각하는 것
- 큐는 CPU의 태스크 스케줄링, 네트워크 프린터, 실시간 시스템의 인터럽트 처리, 다양한 이벤트 구동 방식 컴퓨터 시뮬레이션, 콜 센터의 전화 서비스 처리 등에 사용되며, 이진트리의 레벨순회와 그래프의 너비우선탐색에 사용

- 데크는 양쪽 끝에서 삽입과 삭제를 허용하는 자료구조로서 스택과 큐 자료구조를 혼합한 자료구조
- 데크는 스크롤, 문서 편집기의 undo 연산, 웹 브라우저의 방문 기록 등에 사용

스택, 큐, 데크 자료구조의 연산 수행시간 비교

자료구조	구현	삽입	삭제	비고
스택 큐 데크	배열	O(1)*	O(1)*	* 상각분석 평균시간(부록 I)
	연결리스트†	O(1)	O(1)	†데크는 이중연결리스트로 구현