

Linked list, Stack

충북대학교 컴퓨터공학과
정영섭

목 차

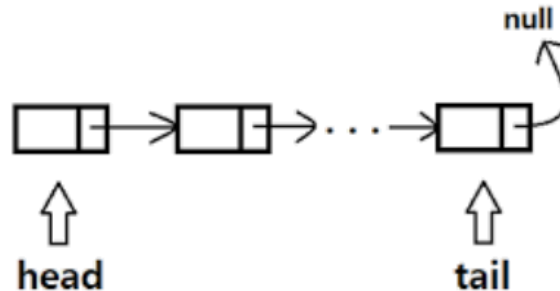
- ❖ Array
- ❖ Linked list
 - Linked list 개량하기
 - Iterator, Collection 상속
 - Array 와의 비교
 - (선택사항) 함수 추가해보기
- ❖ Stack
 - Stack 개량하기
 - Stack 응용하기

주의사항

- 1. IntelliJ IDEA 를 사용하여 코딩
- 2. 향후 코드를 계속 이어서 작성해나가야 하므로,
git 을 통해 관리하거나 동일한 컴퓨터(개인 노트북 등)에서
작업함으로써 이전에 작성한 코드를 분실하지 않도록 할 것

Array, Linked List

- Array
 - A collection of (same-type) "value"s in order
- Linked List 정의
 - A collection of "value"s arranged in a linear (unidirectional)



- Linked List 를 이루는 각 개체를 'Node' 라고 부름
- Linked List 는 결국 'Node'가 순서대로 나열된 형태라고 볼 수 있으므로, 일단 'Node'를 잘 설계&구현하는 것이 첫 걸음
- Q> 그렇다면, 각 'Node'는 어떤 값(변수)과 기능(함수)을 가져야 할까?

Linked List

- Node (노드)

- Src/main/kotlin 에 Node.kt 파일 만들고 아래와 같이 'Node' 클래스 구현
- Generic 타입을 T 라는 이름으로 표현 (이름은 자유롭게 지정 가능(예: K, ...))
- Data class 로써 구현되었고, toString() 을 override 하고 있음

```
1 data class Node<T>(var value: T, var next: Node<T>? = null) {
2     override fun toString(): String {
3         return if (next != null) {
4             "$value -> ${next.toString()}"
5         } else {
6             "$value"
7         }
8     }
9 }
```

코틀린 "문자열" 문법에서,
\$변수, \${변수} 뿐만 아니라,
\${함수호출의 결과값} 형태도 사용 가능하다는 점에 주목

Linked List

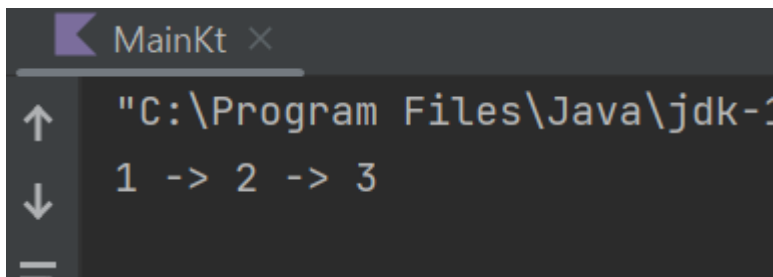
- Node (노드)

- Src/main/kotlin 의 Main.kt 를 아래와 같이 작성 후, 실행

```
1  fun main() {  
2      val node1 = Node(value = 1)  
3      val node2 = Node(value = 2)  
4      val node3 = Node(value = 3)  
5      node1.next = node2  
6      node2.next = node3  
7      println(node1)  
8  }
```

node1, node2, node3 을
.next 를 이용하여 직접
이어주고 있음.

Node 개수가 많아지면
매우 비효율적인 방식.



```
MainKt x  
↑ "C:\Program Files\Java\jdk-11.0.2\bin\java.exe"  
↓ 1 -> 2 -> 3  
=
```

Linked List: push

- Src/main/kotlin 에 `LinkedList.kt` 파일을 아래와 같이 만들기

```
1 class LinkedList<T> {
2     private var head: Node<T>? = null
3     private var tail: Node<T>? = null
4     private var size = 0
5     fun isEmpty(): Boolean {
6         return size == 0
7     }
8     override fun toString(): String {
9         if (isEmpty()) {
10             return "Empty list"
11         }
12         return head.toString()
13     }
14
15     fun push(value: T) {
16         head = Node(value = value, next = head)
17         if (tail == null) {
18             tail = head
19         }
20         size++
21     }
22 }
```

Head-first insertion:
리스트의 가장 앞쪽에
item 한 개를 추가

Linked List: push

- Main.kt 파일 내용을 아래와 같이 작성 후, 실행

```
1  
2 ▶ fun main() {  
3     val list = LinkedList<Int>()  
4     list.push( value: 10)  
5     list.push( value: 30)  
6     list.push( value: 20)  
7     println(list)  
8 }
```

```
MainKt ×  
↑ "C:\Program Files\Java\jdk-17\I  
↓ 20 -> 30 -> 10
```

- DIY> 아래와 같이 push() 함수를 사용할 수 있도록, push() 함수 정의를 수정해보자.

```
2 ▶ fun main() {  
3     val list = LinkedList<Int>()  
4     list.push( value: 10).push( value: 30).push( value: 20)  
5     println(list)  
6 }
```

실행 결과는
수정 전과 동일할 것이다.

빈 칸에 들어갈 것은?

```
15 fun push(value: T):          {  
16     head = Node(value = value, next = head)  
17     if (tail == null) {  
18         tail = head  
19     }  
20     size++  
21     return           
22 }
```


Linked List: append

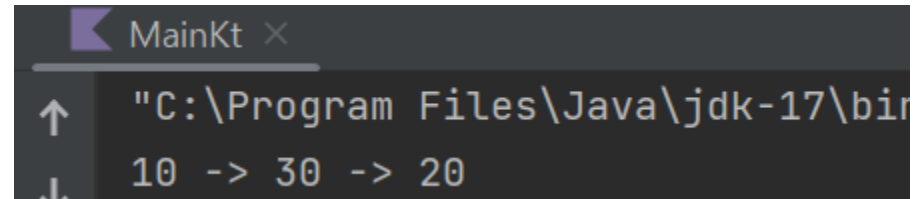
- 가장 앞이 아니라, 가장 뒤에 item 을 추가("Tail-end insertion")하는 `append()` 라는 함수를 새로 추가해보자.
- LinkedList 클래스의 `push()` 함수 아래쪽에, `append()` 함수를 아래와 같이 추가하자.

```
23
24     fun append(value: T) {
25         // 1
26         if (isEmpty()) {
27             push(value)
28             return
29         }
30         // 2
31         tail?.next = Node(value = value)
32         // 3
33         tail = tail?.next
34         size++
35     }
```

Linked List: append

- Main.kt 파일 내용을 아래와 같이 작성 후, 실행

```
1
2 ▶ fun main() {
3     val list = LinkedList<Int>()
4     list.append(10)
5     list.append(30)
6     list.append(20)
7     println(list)
8 }
```



```
MainKt ×
↑ "C:\Program Files\Java\jdk-17\bin
↓ 10 -> 30 -> 20
```

뒤쪽에 item을 추가하는 함수이기 때문에,
호출한 순서대로 표기된 것을 볼 수 있음

- DIY> append() 함수를 아래와 같이 사용할 수 있도록 수정해보자.

```
2 ▶ fun main() {
3     val list = LinkedList<Int>()
4     list.append(10).append(30).append(20)
5     println(list)
6 }
```

```
24 fun append(value: T):          {
25
26
27
28
29
30
31
32
33
34
35
36 }
37 }
```

Linked List: insert

- LinkedList 에 item(또는 node) 을 “임의의 위치”에 추가하는 함수를 만들어보자.
 - 특히, 임의의 Node 를 먼저 찾아내고, 해당 Node 바로 뒤쪽 위치에 새로운 Node 를 추가해주는 함수 insert() 를 만들어보자.
 - 우선, LinkedList 클래스에 아래와 같이 nodeAt() 이라는 함수를 추가해주자.

```
38      fun nodeAt(index: Int): Node<T>? {    주어진 index 번째에 위치한
39          // 1                               Node 객체 리턴
40          var currentNode = head
41          var currentIndex = 0
42          // 2
43          while (currentNode != null && currentIndex < index) {
44              currentNode = currentNode.next
45              currentIndex++
46          }
47          return currentNode
48      }
```

Linked List: insert

- Main.kt 를 수정 후, 실행 (앞서 만든 nodeAt() 함수 테스트)

```
1  
2  ▶ fun main() {  
3      val list = LinkedList<Int>()  
4      list.append(10).append(30).append(20)  
5      println(list)  
6  
7      println(list.nodeAt(index: 2))  
8      println(list.nodeAt(index: 3))  
9  }
```

MainKt ×

↑ "C:\Program Files\Java\jdk
↓ 10 -> 30 -> 20
20
null

- 아래와 같이, LinkedList 클래스에 insert() 함수 정의

```
50 fun insert(value: T, afterNode: Node<T>): Node<T> {  
51     // 1  
52     if (tail == afterNode) {  
53         append(value)  
54         return tail!!  
55     }  
56     // 2  
57     val newNode = Node(value = value, next = afterNode.next)  
58     // 3  
59     afterNode.next = newNode  
60     size++  
61     return newNode  
62 }
```

Linked List: insert

- Main.kt 를 아래와 같이 수정하여 실행

```
2  ▶ fun main() {  
3      val list = LinkedList<Int>()  
4      list.append(10).append(30).append(20)  
5  
6      println("Before inserting: $list")  
7      var middleNode = list.nodeAt( index: 1)!!  
8      for (i in 1 ≤ .. ≤ 3) {  
9          middleNode = list.insert( value: -1 * i, middleNode)  
10     }  
11     println("After inserting: $list")  
12 }
```

```
MainKt x  
↑ "C:\Program Files\Java\jdk-17\bin\java.exe" -javaa  
↓ Before inserting: 10 -> 30 -> 20  
⌵ After inserting: 10 -> 30 -> -1 -> -2 -> -3 -> 20  
⌵ Process finished with exit code 0
```

Linked List: Time complexity

- 지금까지 살펴본 함수들에 대한 Time complexity

	push	append	insert	nodeAt
Behaviour	insert at head	insert at tail	insert after a node	returns a node at given index
Time complexity	$O(1)$	$O(1)$	$O(1)$	$O(i)$, where i is the given index

Linked List: pop

- 맨 앞에 위치한 1개의 item을 제거 (하면서 그 값을 리턴)하는 pop() 함수를 아래와 같이 만들고, main() 작성해서 실행

```
63  
64 fun pop(): T? {  
65     if (!isEmpty()) size--  
66     val result = head?.value  
67     head = head?.next  
68     if (isEmpty()) {  
69         tail = null  
70     }  
71     return result  
72 }
```

```
2  ▶ fun main() {  
3      val list = LinkedList<Int>()  
4      var poppedValue = list.pop()  
5      println(poppedValue)  
6  
7      list.append(10).append(30).append(20)  
8  
9      println("Before popping list: $list")  
10     poppedValue = list.pop()  
11     println("After popping list: $list")  
12     println("Popped value: $poppedValue")  
13 }
```

```
MainKt x  
↑ "C:\Program Files\Java\jdk-17\bin\java.  
↓ null  
↕ Before popping list: 10 -> 30 -> 20  
↕ After popping list: 30 -> 20  
↕ Popped value: 10
```

Linked List: removeLast

- LinkedList 클래스에, 가장 뒤에 있는 item을 삭제(및 값을 리턴)하는 removeLast() 함수를 정의하고, Main.kt 에서 실행

```
74 fun removeLast(): T? {  
75     // 1  
76     val head = head ?: return null  
77     // 2  
78     if (head.next == null) return pop()  
79     // 3  
80     size--  
81     // 4  
82     var prev = head  
83     var current = head  
84     var next = current.next  
85     while (next != null) {  
86         prev = current  
87         current = next  
88         next = current.next  
89     }  
90     // 5  
91     prev.next = null  
92     tail = prev  
93     return current.value  
94 }
```

```
2 fun main() {  
3     val list = LinkedList<Int>()  
4     list.append(10).append(30).append(20)  
5  
6     println("Before removing last node: $list")  
7     val removedValue = list.removeLast()  
8     println("After removing last node: $list")  
9     println("Removed value: $removedValue")  
10 }
```

```
MainKt x  
↑ "C:\Program Files\Java\jdk-17\bin\java.exe  
↓ Before removing last node: 10 -> 30 -> 20  
! After removing last node: 10 -> 30  
+ Removed value: 20
```


Linked List: removeAfter

- LinkedList 클래스에, 임의 위치의 item을 삭제(및 값을 리턴)하는 removeAfter() 함수 만들고, Main.kt 작성 후 실행

```
96 fun removeAfter(node: Node<T>): T? {  
97     val result = node.next?.value  
98     if (node.next == tail) {  
99         tail = node  
100     }  
101     if (node.next != null) {  
102         size--  
103     }  
104     node.next = node.next?.next  
105     return result  
106 }
```

```
2 fun main() {  
3     val list = LinkedList<Int>()  
4     list.append(10).append(30).append(20)  
5  
6     println("Before removing at particular index: $list")  
7     val index = 1  
8     val node = list.nodeAt(index - 1)!!  
9     val removedValue = list.removeAfter(node)  
10    println("After removing at index $index: $list")  
11    println("Removed value: $removedValue")  
12 }
```

```
MainKt x  
↑ "C:\Program Files\Java\jdk-17\bin\java.exe" -javaage  
↓ Before removing at particular index: 10 -> 30 -> 20  
↕ After removing at index 1: 10 -> 20  
- Removed value: 30
```

Linked List: Time complexity

- 지금까지 살펴본 함수들에 대한 Time complexity

	pop	removeLast	removeAfter
Behaviour	remove at head	remove at tail	remove the immediate next node
Time complexity	$O(1)$	$O(n)$	$O(1)$

removeAfter()의 인자로
주어지는 것이
Node이기 때문에
 $O(1)$

- ❖ Q> 만약, removeAt(Int) 함수가 있었다면, removeAt 함수의 복잡도는?

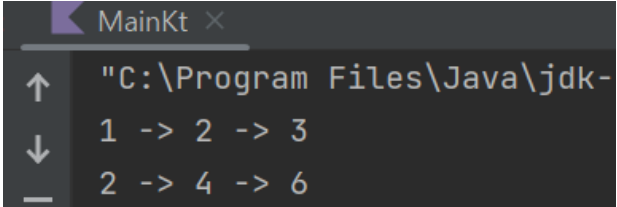
- 

Linked List (개량하기)

- Motivation

- 만약, LinkedList 객체의 각 'item의 2배 값'을 화면에 순서대로 출력하려면 어떻게 해야 할까?
- 아래 코드를 살펴보자.

```
2  ▶ fun main() {  
3      val list = LinkedList<Int>()  
4      list.push( value: 3)  
5      list.push( value: 2)  
6      list.push( value: 1)  
7      println(list)  
8      var the_node = list.nodeAt( index: 0)!!  
9      while(true) {  
10         print("${the_node.value * 2}")  
11         if (the_node.next != null) {  
12             print(" -> ")  
13             the_node = the_node.next!!  
14         }  
15         else  
16             break  
17     }  
18 }
```



```
MainKt x  
↑ "C:\Program Files\Java\jdk-  
↓ 1 -> 2 -> 3  
— 2 -> 4 -> 6
```

while 반복문을 돌면서,
매번 next 를 체크(null 여부)해줘야 하는
번거로움

Linked List (개량하기)

- 지금부터는 Linked List 를 Kotlin 에서 제공되는 문법/기능을 더욱 적극적으로 활용하여 멋지게 구현해보자.
- Src/main/kotlin 에 [LinkedList.kt](#) 파일에 대하여,

```
1 class LinkedList<T> : Iterable<T> {  
2     private var head: Node<T>? = null  
3     private var tail: Node<T>? = null  
4     var size = 0  
5     private set // read only로 만들기  
6  
7     override fun iterator(): Iterator<T> {  
8         return LinkedListIterator( list: this)  
9     }  
10  
11     fun isEmpty(): Boolean {  
12         return size == 0  
13     }  
14     override fun toString(): String {
```

LinkedList 클래스 선언 내용의 상단부에서,
위와 같이 수정
(Iterable 상속)

```
115 class LinkedListIterator<K> (  
116     private val list: LinkedList<K>  
117 ) : Iterator<K> {  
118     private var index = 0  
119     private var lastNode: Node<K>? = null  
120  
121     override fun next(): K {  
122         if (index >= list.size) throw IndexOutOfBoundsException()  
123         lastNode = if (index == 0) {  
124             list.nodeAt( index: 0)  
125         } else  
126             lastNode?.next  
127         index++  
128         return lastNode!!.value  
129     }  
130     override fun hasNext(): Boolean {  
131         return index < list.size  
132     }  
133 }
```

LinkedList 클래스 아래쪽에(클래스 정의가 끝난 중괄호{})
LinkedListIterator 클래스 정의 (iterator 상속 (주의: Iterable과 다름))

Linked List (개량하기)

- Main.kt 파일 작성하여 실행

```
2  ▶ fun main() {  
3      val list = LinkedList<Int>()  
4      list.push(value: 3)  
5      list.push(value: 2)  
6      list.push(value: 1)  
7      println(list)  
8  
9      var s: String = ""  
10     for (item in list)  
11         s += " -> ${item * 2}"  
12     s = s.substring(startIndex: 4)  
13     println(s)  
14 }
```

For 문에서 사용 가능해졌다!

Q> 여기서 변수(반복자) 'item'의 자료형은 뭘까?

- 힌트1: println(변수.javaClass)
- 힌트2: LinkedListIterator 살펴보기

→ 답:

```
MainKt x  
↑ "C:\Program Files\Java\jdk-1  
↓ 1 -> 2 -> 3  
  2 -> 4 -> 6
```

Linked List (개량하기)

- 이번에는, LinkedList 안에 존재하는 item 들을 보다 편리하게 다룰 수 있도록 하기 위해, Collection 을 상속하도록 해보자.
 - LinkedList 클래스에서 아래와 같이 수정하기

```
1 class LinkedList<T> : Iterable<T>, Collection<T> {
2     private var head: Node<T>? = null
3     private var tail: Node<T>? = null
4     override var size = 0
5     private set // read only로 만들기
6
7     override fun iterator(): Iterator<T> {
8         return LinkedListIterator(list: this)
9     }
10
11     override fun isEmpty(): Boolean {
12         return size == 0
13     }
14
15     override fun contains(element: T): Boolean {
16         for (item in this)
17             if (item == element) return true
18         return false
19     }
20
21     override fun containsAll(elements: Collection<T>): Boolean {
22         for (searched in elements)
23             if (!contains(searched)) return false
24         return true
25     }
26
27     override fun toString(): String {
```

[참고] Collection 이 Iterable 도 포함하므로,
Collection 만 상속받아도 상관없음

Collection 은 size 변수와 isEmpty() 를 포함하는
Interface 여서,
이들 앞에 'override'를 붙여주어야 함

새롭게 추가되는 override 함수들

Q> containsAll 함수의 Time Complexity 는?

→



Linked List (개량하기)

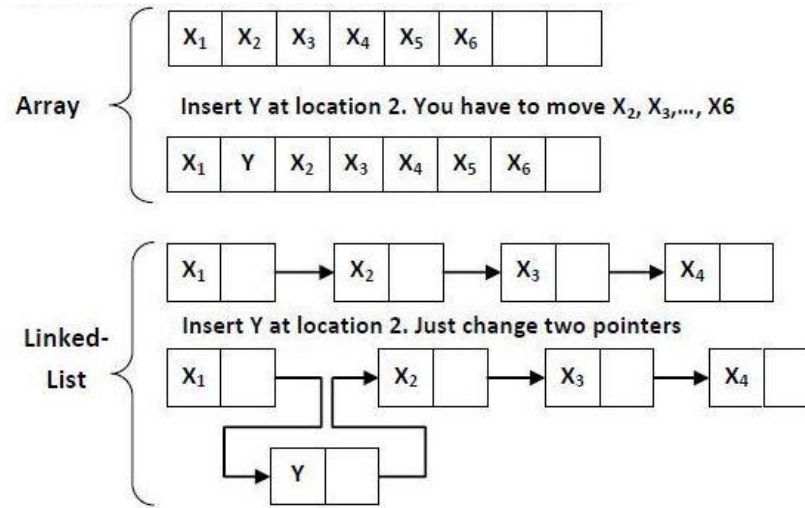
- Main.kt 내용 수정 후 실행

```
3  ▶ fun main() {  
4      val list = LinkedList<Int>()  
5      list.push(value: 3)  
6      list.push(value: 2)  
7      list.push(value: 1)  
8  
9      var list2 = LinkedList<Int>()  
10     list2.append(2).append(3)  
11  
12     println(list.containsAll(list2))  
13 }
```

```
MainKt x  
↑ "C:\Program Files\Java\jdk-17\bin  
↓ true
```

(정리) Linked List 와 Array

● Array 와의 비교



ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

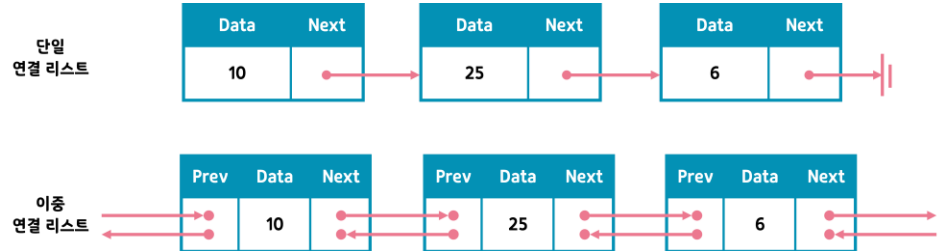
(정리) Linked List 와 Array

- [돌발퀴즈] 배열에 만약 push() 함수가 존재한다면, time complexity 가 어떨까? (참고: push() 함수는 맨 앞에 Node 1개를 추가해주는 함수)
 - 힌트) Array: "동일한 크기를 가진 item"들이 "메모리 상에서 실제로 연속"

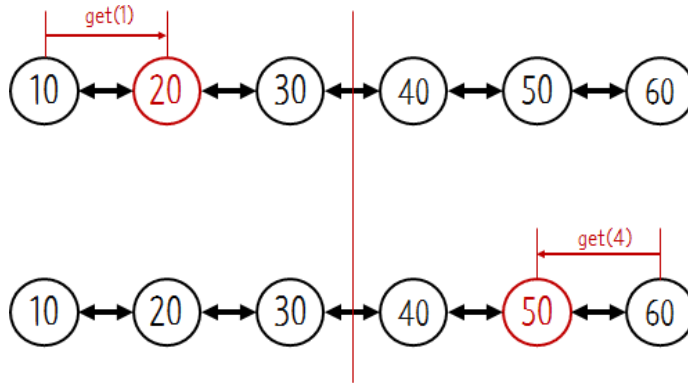
Linked List (개량하기)

- 앞서 살펴본 Linked List 의 한계점을 극복하기 위해 양방향으로 만들수도 있다.

- 양방향 (이중) 연결 리스트



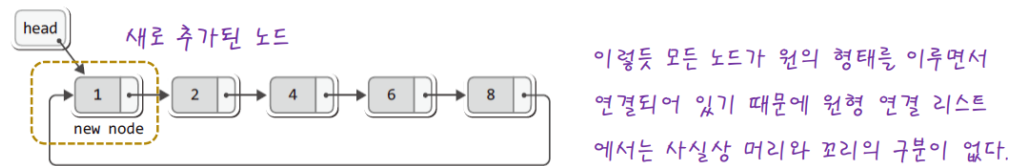
- 장점: 노드 탐색 시에 걸리는 시간을 단축 가능



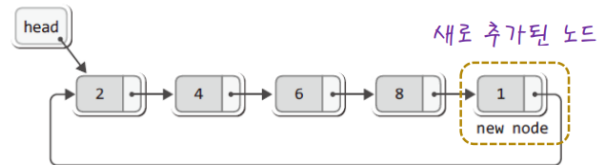
- 단점: 메모리 공간이 추가적으로 필요

Linked List (개량하기)

- 원형 (환형) 으로 만들 수도 있다.
 - 필요에 따라, 원형 '이중' 연결 리스트로도 만들 수 있음



▶ [그림 05-3: 원형 연결 리스트의 머리에 노드 추가]



▶ [그림 05-4: 원형 연결 리스트의 꼬리에 노드 추가]

- 장점: 임의의 노드부터 시작해서, 모든 노드로 접근이 가능

Linked List (개량하기)

- 다음 슬라이드부터 이어지는 LinkedList 개량하는 내용에 대한 설명은 '선택 사항'
 - 시험/과제에서 배제된다는 의미는 아님
 - 자습 가능한 수준이므로, 수업시간을 고려하여 선택적으로 설명
 - 반드시 자습해볼것 (직접 구현하고 결과 확인해보기)
- 단, 본 PPT 자료의 Linked List 내용 이후에 이어지는 "Stack" 부터는 선택사항이 아니라 강의에서 반드시 다룰 내용임

Linked List (개량하기): printInReverse

- DIY> Item 들을 역순으로 화면에 출력하는 함수를 만들어보자.
 - 결과: Main.kt 실행 예시

```
3  ▶ fun main() {  
4      val list = LinkedList<Int>()  
5      list.append(3)  
6      list.append(2)  
7      list.append(1)  
8      list.append(4)  
9      list.append(5)  
10     println(list)  
11     list.printInReverse()  
12 }
```

```
MainKt ×  
↑ "C:\Program Files\Java\jdk-17\bin\j  
3 -> 2 -> 1 -> 4 -> 5  
↓ 5 -> 4 -> 1 -> 2 -> 3  
:jl Process finished with exit code 0
```

Linked List (개량하기): printInReverse

- DIY> Item 들을 역순으로 화면에 출력하는 함수를 만들어보자.
 - 정답 코드
 - 추가한 함수의 Time complexity = $O(n)$

```
127 fun printInReverse() {  
128     this.nodeAt(index: 0)?.printInReverse()  
129 }
```

LinkedList 클래스에 추가

```
10 fun printInReverse() {  
11     this.next?.printInReverse()  
12     // 1  
13     if (this.next != null) {  
14         print(" -> ")  
15     }  
16     // 2  
17     print(this.value.toString())  
18 }
```

Node 클래스에 추가

Linked List (개량하기): getMiddle

- DIY> 순서 상 가운데에 위치한 item 을 화면에 출력하는 getMiddle() 함수 만들기
 - 실행 예시

```
3  ▶ fun main() {  
4      val list = LinkedList<Int>()  
5      list.append(3)  
6      list.append(2)  
7      list.append(1)  
8      list.append(4)  
9      list.append(5)  
10     println(list)  
11     println(list.getMiddle()?.value)  
12 }
```

main() 함수 내용

```
MainKt ×  
↑ "C:\Program Files\Java\jdk-17\  
↓ 3 -> 2 -> 1 -> 4 -> 5  
  1
```

Linked List (개량하기) : getMiddle

- DIY> 순서 상 가운데에 위치한 item 을 화면에 출력하는 getMiddle() 함수 만들기
 - 정답 코드
 - 추가한 함수의 Time complexity = $O(n)$

```
131 fun getMiddle(): Node<T>? {  
132     var slow = this.nodeAt( index: 0)  
133     var fast = this.nodeAt( index: 0)  
134     while (fast != null) {  
135         fast = fast.next  
136         if (fast != null) {  
137             fast = fast.next  
138             slow = slow?.next  
139         }  
140     }  
141     return slow  
142 }
```

힌트> 토끼와 거북이
(토끼가 거북이보다 2배 빠르다)

LinkedList 클래스에 추가

Linked List (개량하기) : reversed

- DIY> item들을 역순으로 가지고 있는 새로운 LinkedList 를 생성해서 리턴하는 함수 만들기
 - 실행 예시

```
3  ▶ fun main() {  
4      val list = LinkedList<Int>()  
5      list.append(3)  
6      list.append(2)  
7      list.append(1)  
8      list.append(4)  
9      list.append(5)  
10  
11      println("Original: $list")  
12      println("Reversed: ${list.reversed()}")  
13 }
```

```
MainKt ×  
↑ "C:\Program Files\Java\jdk-17\bin\j  
↓ Original: 3 -> 2 -> 1 -> 4 -> 5  
= Reversed: 5 -> 4 -> 1 -> 2 -> 3
```

Linked List (개량하기) : reversed

- DIY> item들을 역순으로 가지고 있는 새로운 LinkedList 를 생성해서 리턴하는 함수 만들기
 - 정답 코드
 - 추가한 함수의 Time complexity = $O(n)$

```
144     private fun addInReverse(list: LinkedList<T>, node: Node<T>)
145     {
146         val next = node.next
147         if (next != null) {
148             addInReverse(list, next)
149         }
150         list.append(node.value)
151     }
152
153     fun reversed(): LinkedList<T> {
154         val result = LinkedList<T>()
155         val head = this.nodeAt(index: 0)
156         if (head != null) {
157             addInReverse(result, head)
158         }
159         return result
160     }
```

LinkedList 클래스에 추가되는
함수들

Linked List (개량하기) : mergeSorted

- DIY> A객체에서 B객체에 대하여, B객체의 item들 사이에 A객체의 item들을 (대소비교하여) 적절한 위치에 삽입한 결과 (즉, merge한 결과)를 가지는 새로운 LinkedList 객체를 리턴
 - 실행 예시

```
3  ▶ fun main() {  
4      val list = LinkedList<Int>()  
5      list.append(1)  
6      list.append(2)  
7      list.append(3)  
8      list.append(4)  
9      list.append(5)  
10  
11     val other = LinkedList<Int>()  
12     other.append(-1)  
13     other.append(-2)  
14     other.append(-3)  
15     other.append(-4)  
16     other.append(-5)  
17  
18     println("Left: $list")  
19     println("Right: $other")  
20     println("Merged: ${list.mergeSorted(other)}")  
21 }
```

```
MainKt x  
↑ "C:\Program Files\Java\jdk-17\bin\java.exe" -javaagent:C:\Us  
↓ Left: 1 -> 2 -> 3 -> 4 -> 5  
Right: -1 -> -2 -> -3 -> -4 -> -5  
Merged: -1 -> -2 -> -3 -> -4 -> -5 -> 1 -> 2 -> 3 -> 4 -> 5
```

Linked List (개량하기) : mergeSorted

- DIY> A객체에서 B객체에 대하여, B객체의 item들 사이에 A객체의 item들을 (대소비교하여) 적절한 위치에 삽입한 결과 (즉, merge한 결과)를 가지는 새로운 LinkedList 객체를 리턴
 - 정답 코드 (아래 코드들을 LinkedList 클래스에 추가)
 - Time complexity = $O(m+n)$
 - m: # of nodes (this)
 - n: # of nodes (otherList)

```
162         private fun append(  
163             result: LinkedList<T>,  
164             node: Node<T>  
165         ): Node<T>? {  
166             result.append(node.value)  
167             return node.next  
168         }
```

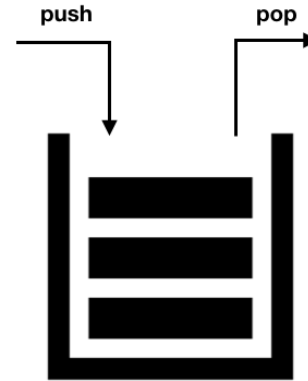
기존에 존재하던 append() 함수와
이름은 동일하지만 전혀 다른
매개변수를 가진 함수를 추가로
정의 가능!

```
170         fun mergeSorted(  
171             otherList: LinkedList<T>  
172         ): LinkedList<T> {  
173             if (this.isEmpty()) return otherList  
174             if (otherList.isEmpty()) return this  
175             val result = LinkedList<T>()  
176             var left = nodeAt(index: 0)  
177             var right = otherList.nodeAt(index: 0)  
178             while (left != null && right != null) {  
179                 if ((left.value as Int) < (right.value as Int)) {  
180                     left = append(result, left)  
181                 } else {  
182                     right = append(result, right)  
183                 }  
184             }  
185             while (left != null) {  
186                 left = append(result, left)  
187             }  
188             while (right != null) {  
189                 right = append(result, right)  
190             }  
191             return result  
192         }
```

Stack

- 입구이자 출구가 1개 있는 구조

- 먼저 들어갔을수록,
나중에 나오게 됨
 - 즉, 나중에 들어갔을수록
먼저 나오게 됨
(LIFO: Last-in First-out)
- 주요 기능
 - 넣기 (push)
 - 안에 들어있는 것들의 제일 위쪽에
item 1개 추가
 - 빼기 (pop)
 - 가장 위에 있는 item 1개를 제거



Stack

- Src/main/kotlin 에서 Stack.kt 파일에 아래와 같이 구현
 - StackInterface 에서는, push(), pop() 만 존재
 - Stack 에서 실제로 Stack 으로서 동작하기 위한 변수, 함수 구현

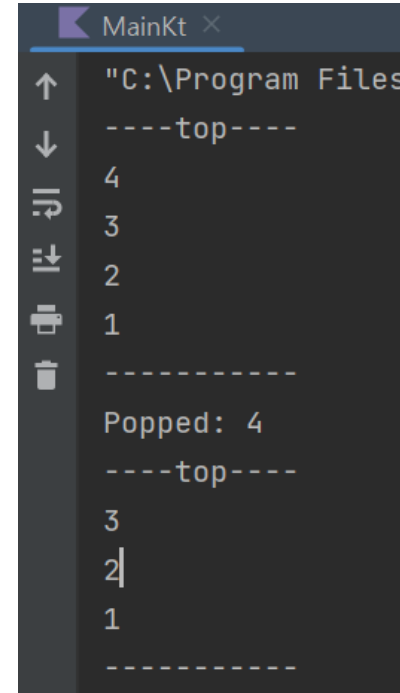
```
interface StackInterface<Element> {  
    fun push(element: Element)  
    fun pop(): Element?  
}
```

```
class Stack<Element>() : StackInterface<Element> {  
    private val storage = arrayListOf<Element>()  
    override fun toString() = buildString { this: StringBuilder  
        appendLine( value: "----top----")  
        storage.asReversed().forEach { it: Element  
            appendLine( value: "$it")  
        }  
        appendLine( value: "-----")  
    }  
  
    override fun push(element: Element) {  
        storage.add(element)  
    }  
  
    override fun pop(): Element? {  
        if (storage.size == 0) {  
            return null  
        }  
        return storage.removeAt(storage.size - 1)  
    }  
}
```

Stack

- Main.kt 에서 아래와 같이 작성 후 실행

```
3  ▶ fun main() {  
4      val stack = Stack<Int>().apply { this: Stack<Int>  
5          push( element: 1)  
6          push( element: 2)  
7          push( element: 3)  
8          push( element: 4)  
9      }  
10     print(stack)  
11     val poppedElement = stack.pop()  
12     if (poppedElement != null) {  
13         println("Popped: $poppedElement")  
14     }  
15     print(stack)  
16 }
```



```
MainKt x  
↑ "C:\Program Files  
↓ ----top----  
4  
3  
2  
1  
-----  
Popped: 4  
----top----  
3  
2  
1  
-----
```

- Q> push, pop 의 Time complexity 는?

- 

Stack (개량하기)

- Stack 에 존재하는 top 위치의 item 을 pop하지 않고 엿보는(?) 함수 만들어 보자.

StackInterface 수정

```
1 interface StackInterface<Element> {  
2     val count: Int  
3     get  
4  
5     fun peek(): Element?  
6     val isEmpty: Boolean  
7     get() = count == 0  
8     fun push(element: Element)  
9     fun pop(): Element?  
10 }
```

Stack 에 추가

```
15 override fun peek(): Element? {  
16     return storage.lastOrNull()  
17 }  
18 override val count: Int  
19     get() = storage.size
```


Stack (개량하기)

- Stack 에 존재하는 top 위치의 item 을 pop하지 않고 엿보는(?) 함수 만들어보자.

Stack 에서 pop() 내용 개선

```
32  override fun pop(): Element? {  
33      if (storage.size == 0) {  
34          return null  
35      }  
36      return storage.removeAt(index: count - 1)  
37  }
```

이 부분은 isEmpty 로 바꿀 수 있을 것

main() 작성 및 실행

```
3  fun main() {  
4      val stack = Stack<Int>().apply { this: Stack<Int>  
5          push( element: 1)  
6          push( element: 2)  
7          push( element: 3)  
8          push( element: 4)  
9      }  
10     print(stack)  
11     println(stack.peek())  
12     print(stack)  
13 }
```

Stack (응용하기)

- Stack이 Iterable, Collection 인터페이스를 상속받도록 만들수도 있겠으나, 그렇게하면 더 이상 Stack 이라고 보기 어렵게 될 것임
 - 따라서, Stack은 해당 인터페이스를 상속받지 않도록 하자.
- Stack을 이용하여 괄호 개수가 짝이 맞는지 체크하는 함수를 만들어보자.
 - 짝이 맞으면 true, 안 맞으면 false 리턴.
 - Main.kt 를 우측과 같이 구현하여 실행
 - 기본 제공되는 String 클래스의 멤버함수를 직접 정의해서 이용하고 있는 부분에 주목.
 - import 없이도 접근/사용 가능한 범위에서는 이 함수 이용 가능해짐
 - Time complexity = $O(n)$
 - n: 주어진 문자열의 길이 (글자 개수)

```
1 fun String.checkParentheses(): Boolean {
2     val stack = Stack<Char>()
3     for (character in this) {
4         when (character) {
5             '(' -> stack.push(character)
6             ')' -> if (stack.isEmpty) {
7                 return false
8             } else {
9                 stack.pop()
10            }
11        }
12    }
13    return stack.isEmpty
14 }
15
16 fun main() {
17     var s = "h((e))llo(world)()"
18     println(s.checkParentheses())
19     println("(hello world".checkParentheses())
20 }
```

문제 풀이

1. LinkedList 에서 제일 뒤에 item을 추가하는 작업의 Time complexity 는 $O(n)$ 이다. (O, X)
2. Stack 의 pop() 기능의 Time complexity 는 $O(1)$ 이다. (O, X)
3. Stack 에 가장 나중에 넣은 item 은 가장 먼저 나오게 된다. (O, X)

요약

- ❖ LinkedList 는 임의 위치의 item들에 대한 접근을 허용한다.
- ❖ LinkedList 는 Node 들을 순서에 따라서 가지고 있는 구조이다.
- ❖ Stack 은 item 을 넣는 위치와 빼는 위치가 동일하다.
- ❖ Stack 에 가장 먼저 넣은 item 이 가장 나중에 나오게 될 것이다.