1. 기초 강의

동영상 강의 컨텐츠 확인 > 2. 연결 리스트

Link :

https://swexpertacademy.com/main/learn/course/subjectDetail.do?courseId=CONTENTS_REVIEW&subjectId=AYVXaMEKQSIDFARs

※ 출석은 강의 수강 내역으로 확인합니다.

2. 실전 강의

2.0 Linked List 개념 설명 및 종류

데이터가 자료의 주소 값으로 서로 연결(Link)되어 있는 구조

단순 연결 리스트, 이중 연결 리스트, 환형 연결 리스트 등의 구조가 있다.

0) Array와 비교 및 계산복잡도

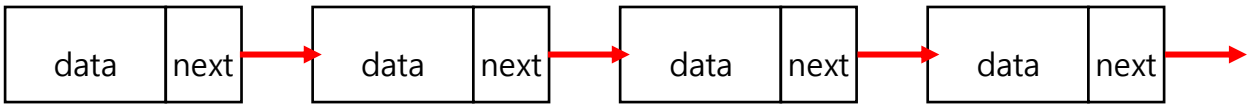
	Array	Linked List
장점	무작위 접근 가능	빠른 자료 삽입, 삭제 자유로운 크기 조절
단점	느린 자료 삽입, 삭제 크기 조절 불가능	순차 접근만 가능 메모리 추가 할당

Operation	Time complexity
Access i-th element	O(N)
Insert element at here	O(1)
Delete element	O(N)(search time)

1) 단순 연결 리스트 (Singly Linked List)

각 노드에서 단방향으로 연결되는 리스트

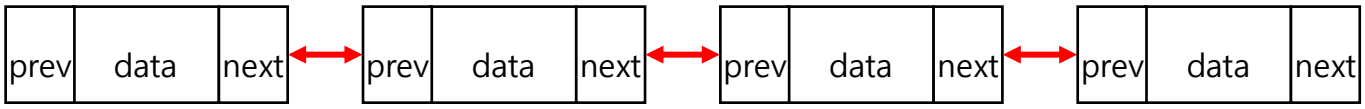
후행 노드는 쉽게 접근 가능하지만, 선행 노드 접근이 복잡한 단점 존재



2) 이중 연결 리스트 (Doubly Linked List)

각 노드에서 양방향(선행, 후행)으로 연결되는 리스트

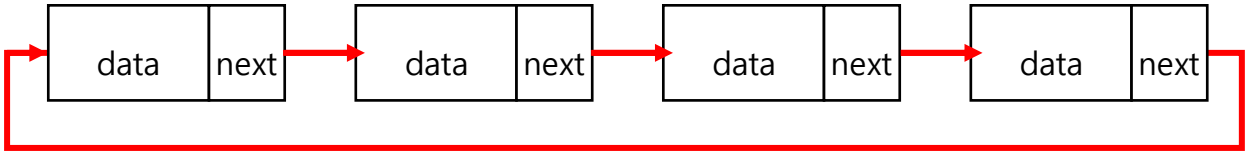
양 방향 접근이 용이하지만, 메모리를 추가적으로 사용



3) 원형 연결 리스트 (Circular Linked List)

각 노드에서 단방향으로 진행되는 리스트

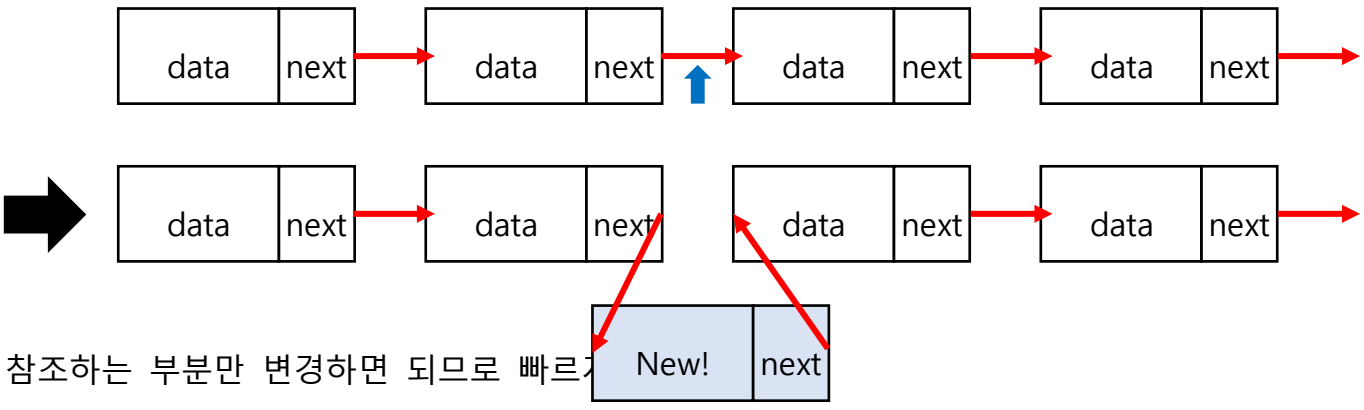
한 노드에서 모든 노드로 접근이 가능



2.1 Linked List의 동작

데이터 삽입

파란 화살표 표시된 위치에 데이터 삽입

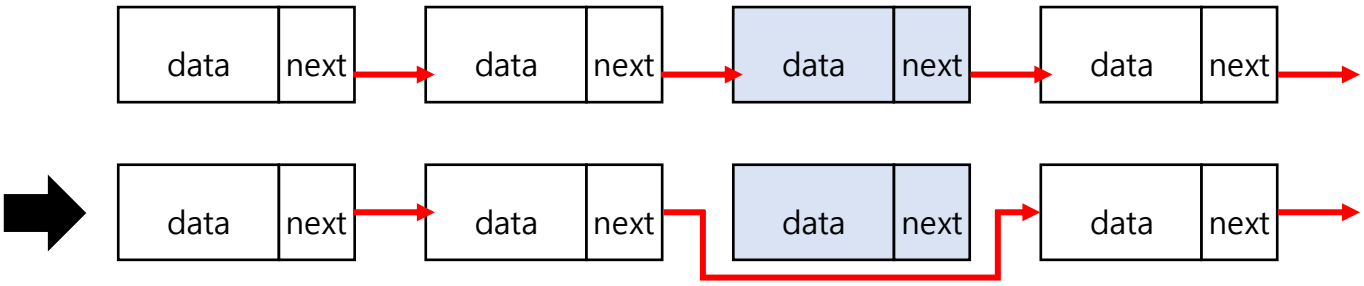


참조하는 부분만 변경하면 되므로 빠르

이중, 환형 연결 리스트에서도 동일하게 수행

데이터 삭제

파란 화살표 표시된 데이터 삭제



참조하는 부분만 변경하면 되므로 빠르게 수행 가능
이중, 환형 연결 리스트에서도 동일하게 수행

2.1 정적 할당을 사용한 Linked List 구현

연결 리스트를 포함해 앞으로 여러분들은 동적 할당이 필요한 자료구조를 구현하게 될 것입니다. 하지만 알고리즘 문제를 풀 때 동적 할당을 일일이 하게 되면 성능이 떨어지고 실수할 여지가 많아집니다.
Pro 시험에 대비하여, 동적 할당 대신 메모리 풀(memory pool)을 통해 정적 할당 하는 팁을 알려드립니다. 연결 리스트의 노드를 동적 할당하는 방식과 메모리 풀을 이용한 방식입니다.

```
// Singly Linked List Node
struct Node {
        int data;
        Node* next;
};

// 1. 동적 할당 방식
Node* new_node(int data) {
        Node* node = new Node;

        node->data = data;
        node->next = nullptr;
```

```
	return node;
}

// 2. 정적 할당(메모리 풀) 방식
constexpr size_t MAX_NODE = 10000;

int node_count = 0;
Node node_pool[MAX_NODE];

Node* new_node(int data) {
	node_pool[node_count].data = data;
	node_pool[node_count].next = nullptr;

	return &node_pool[node_count++];
}
```

메모리 풀은 사용될 노드를 한 번에 모두 할당한 다음, 필요할 때마다 하나씩 꺼내 쓰는 방식입니다.
메모리 풀의 장점은 다음과 같습니다.
1. 동적 할당을 하는 오버헤드가 없어집니다.
2. 사용이 끝날 때마다(특히 여러 개의 테스트 케이스가 있는 경우) 메모리를 해제할 필요가 없습니다.
3. 모든 노드가 메모리 상에서 뭉쳐 있기 때문에 캐시 효율이 높아집니다.

실제로 프로그램을 개발할 때는 동적 할당을 써야 하겠지만, 알고리즘 문제를 풀 때는 이러한 정적 할당을 쓰는 것이 수행 시간에 있어서 더 유리합니다.

2 초기화

```
Node head;

void init() {
	head.next = nullptr;
}
```

보통 Singly Linked List 를 구현할 때 편의를 위해 더미 노드 head 를 만들어 사용합니다. (Doubly Linked List 의 경우 head, tail 두 개의 더미 노드를 사용하죠)

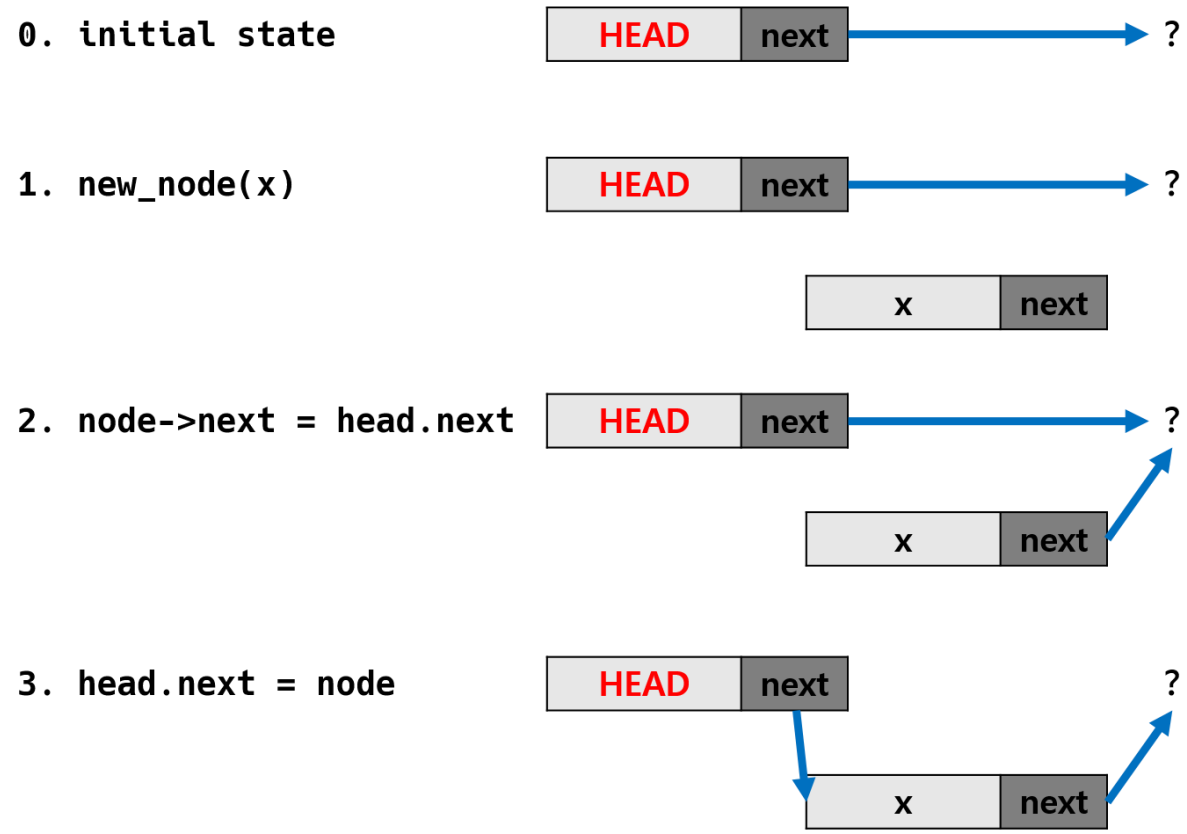
이렇게 더미 노드 head 를 사용하면 구현이 간단해집니다. 연결 리스트에 항상 원소가 1 개는 있기 때문에 삽입/삭제 과정에서 리스트가 비어있을 경우를 생각하지 않아도 됩니다.
단점도 있습니다. struct Node 를 보시면 int 형 데이터와 다음 노드를 가리키는 포인터를 들고 있는데, 우리는 head 의 포인터만 사용할 뿐 int 형 데이터는 사용하지 않습니다. 데이터 낭비가 생깁니다. int 는 고작 4B 이기 때문에 상관 없지만, 아주 큰 데이터를 연결 리스트로 저장하고자 한다면 더미 노드는 좋은 선택이 아닙니다.

3 삽입

리스트 맨 앞에 data 를 추가하는 코드입니다.

```
// O(1)
void insert(int x) {
        Node* node = new_node(x);

        node->next = head.next;
        head.next = node;
}
```

아래 그림처럼 동작합니다. head->next 가 노드를 가리키고 있는지 NULL 인지에 상관 없이 동작합니다.



4 삭제

리스트에서 data 를 찾아서 삭제하는 코드입니다. data 값이 없을 경우 아무 것도 하지 않고, data 값이 여러 개 있을 경우 첫번째 값만 삭제합니다.

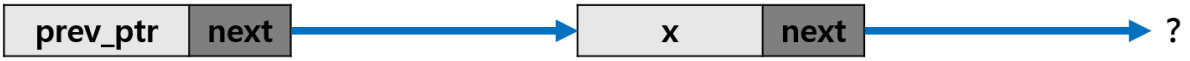
```
// O(N)
void remove(int x) {
	Node* prev_ptr = &head;
	while (prev_ptr->next != nullptr && prev_ptr->next->data != x) {
		prev_ptr = prev_ptr->next;
	}

	if (prev_ptr->next != nullptr) {
		prev_ptr->next = prev_ptr->next->next;
	}
}
```

삭제할 노드의 이전 노드를 찾는 점에 주목합시다. Singly Linked List 에서 이전 노드의 정보를 알 방법이 없습니다. 따라서 노드 연결의 수정, 삭제가 필요한 경우 해당 노드의 이전 노드에서 작업이 이뤄져야 합니다.

while 문이 끝난 후 2 가지 케이스로 나뉩니다. 이전 포인터의 다음 포인터(즉, 삭제할 노드)가 NULL 인 경우와 아닌 경우입니다.

1. prev_ptr->next == nullptr

2. prev_ptr->next != nullptr



NULL 인 경우는 삭제할 노드가 없다는 뜻이므로 아무 것도 하지 않고 return 합니다. NULL 이 아닌 경우 삭제할 노드의 이전 노드와 다음 노드를 이어서 삭제합니다.
삭제한 후의 모습은 아래와 같습니다. 삭제한 노드의 next pointer 는 유지되지만(메모리 풀의 단점 중 하나), 삭제한 노드에 접근하지 않는다면 문제될 점이 없습니다

5 탐색

리스트에 data 값이 있는지 반환하는 코드입니다.

```
// O(N)
bool find(int x) {
	Node* ptr = head.next;
	while (ptr != nullptr && ptr->data != x) {
		ptr = ptr->next;
	}

	return ptr != nullptr;
}
```

while 문이 끝난 후 2 가지 케이스로 나뉩니다. data == x 인 노드가 NULL 인 경우(즉, 리스트에 x 란 값이 없는 경우)와 NULL 이 아닌 경우(x 가 있는 경우)입니다.

6 Singly Linked List 구현

위의 함수를 종합한 결과입니다! 자유롭게 테스트해보세요.

```
#include <cstdio >

struct Node {
	int data;
	Node* next;
};

constexpr size_t MAX_NODE = 1000;

int node_count = 0;
Node node_pool[MAX_NODE];

Node* new_node(int data) {
	node_pool[node_count].data = data;
	node_pool[node_count].next = nullptr;

	return &node_pool[node_count++];
}
```

```
class SinglyLinkedList {
	Node head;

public:
	SinglyLinkedList() = default;

	void init() {
		head.next = nullptr;
		node_count = 0;
	}

	void insert(int x) {
		Node* node = new_node(x);

		node->next = head.next;
		head.next = node;
	}

	void remove(int x) {
		Node* prev_ptr = &head;
		while (prev_ptr->next != nullptr && prev_ptr->next->data != x) {
			prev_ptr = prev_ptr->next;
		}

		if (prev_ptr->next != nullptr) {
			prev_ptr->next = prev_ptr->next->next;
		}
	}

	bool find(int x) const {
		Node* ptr = head.next;
		while (ptr != nullptr && ptr->data != x) {
			ptr = ptr->next;
		}

		return ptr != nullptr;
	}

	void print() const {
```

```
		Node* ptr = head.next;
		printf("[List] ");
		while (ptr != nullptr) {
			printf("%d", ptr->data);
			if (ptr->next != nullptr) {
				printf(" -> ");
			}
			ptr = ptr->next;
		}
		putchar('₩n');
	}
};

int main() {
	SinglyLinkedList slist;
	int a, b;
	for (;;) {
		scanf("%d", &a);
		switch (a) {
		case 0:
			slist.init();
			slist.print();
			break;
		case 1:
			scanf("%d", &b);
			slist.insert(b);
			slist.print();
			break;
		case 2:
			scanf("%d", &b);
			slist.remove(b);
			slist.print();
			break;
		case 3:
			scanf("%d", &b);
			puts(slist.find(b) ? "found" : "not found");
			break;
		default:
			return puts("wrong input"), 0;
```

```
                }
        }
}
```

2.7 구현 연습

간단한 구현 연습 문제가 준비되어 있습니다. 동계 대학생 S/W 알고리즘 특강 - 기본 문제에서 아래 2 문제를 풀어주세요.

1. **기초 Single Linked List 연습**
2. **기초 Double Linked List 연습**

3. 기본 문제

- 암호문 3
- 수열 편집