

# array, vector, list

2025.03.11 이바다

# 들어가며

- 시간 복잡도
  - 특정 작업을 수행하는 데 걸리는 시간을 데이터 크기에 대한 수식으로 표현하는 방식.
  - 데이터 크기가 변경되면 연산 시간이 어떻게 변하는지 보여줌.
  - 빅오표기법으로 나타냄. (ex:  $O(1)$ )
- 연속된 자료구조 : 배열
- 연결된 자료구조 : 연결 리스트

# std::array

```
#include <array>
```

- 메모리를 자동으로 할당 및 해제.
- 매개변수 : 원소의 타입, 배열 크기.
- 원소 접근
  - [] 연산자 : 전달된 인덱스 값이 배열의 크기보다 작은지 검사 X.
  - **at()** : 전달된 인덱스 값이 유효하지 않으면 예외 발생 -> []연산자보다 느림.
  - **front()** : 첫 번째 원소에 대한 참조 반환.
  - **back()** : 마지막 원소에 대한 참조 반환.
  - **data()** : 배열 객체 내부에서 실제 데이터 메모리 버퍼를 가리키는 포인터 반환. 포인터를 함수의 인자로 받는 예전 스타일의 함수 사용할 때 유용.
  - 반복자(**iterator**) + 범위 기반 **for**문 : 소스 코드의 재사용성, 유지 보수, 가독성 이점.
    - **array::begin()** : 첫 번째 원소.
    - **array::end()** : 마지막 원소 다음.
    - **const\_iterator** : **const**로 선언된 배열에 대한 **begin()**, **end()**.
    - **reverse\_iterator** : 역방향으로 이동.

std::array

```
#include <array>
```

```
int main()
{
    std::array<int, 3> arr = { 1, 2, 3 };    // 크기가 3인 int 타입 배열 선언

    std::cout << arr[0] << std::endl;
    std::cout << arr.at(2) << std::endl;
}
```

C:\>

Microsc

1  
3

std::array

```
#include <array>
```

```
int main()
{
    std::array<int, 3> arr = { 1, 2, 3 };    // 크기가 3인 int 타입 배열 선언

    try
    {
        std::cout << arr.at(2) << std::endl;    // 정상 출력
        std::cout << arr.at(4) << std::endl;
    }
    catch (const std::out_of_range& ex)
    {
        std::cout << "예외 발생" << std::endl;
    }
}
```

c:\ Microsoft Visu

3  
예외 발생

std::array

```
#include <array>
```

```
int main()
{
    std::array<int, 3> arr = { 1, 2, 3 };    // 크기가 3인 int 타입 배열 선언

    for (auto a : arr)
    {
        std::cout << a << std::endl;
    }
}
```

Microsoft Visual Stud

1  
2  
3

# std::array

```
#include <array>
```

- C 스타일 배열과의 유사성을 유지하면서 더 빠르고 메모리 효율적으로 동작.
- 단점
  - 크기 고정.
  - 원소 추가 및 삭제 불가.
  - 메모리 할당 방법 변경 불가. 항상 스택 메모리 사용.

# std::vector

```
#include <vector>
```

- std::array의 고정 크기 문제 해결.
- 컴파일러 구현 방법에 따른 용량 (capacity) 있음.
- 벡터의 크기 : 실제로 저장된 원소의 개수. != 용량 (capacity)
- 추가
  - push\_back() : 맨 마지막에 새로운 원소 추가.  $O(1)$
  - insert() : 원하는 위치에 원소 추가.  $O(n)$
  - push\_front()는 지원하지 않음.
  - emplace\_back() / emplace() : 새로운 원소가 추가될 위치에서 해당 원소를 생성.
- 제거
  - pop\_back() : 맨 마지막 원소 제거.  $O(1)$
  - erase() : 반복자 하나를 인자로 받아 해당 위치 원소 제거 / 범위의 시작과 끝을 나타내는 반복자를 받아 시작부터 끝 바로 앞 원소까지 제거.  $O(n)$



std::vector

```
#include <vector>
```

```
std::vector<int> vec0;    // 크기가 0인 벡터 선언  
std::vector<int> vec1 = { 1, 2, 3 };    // 지정한 초기값으로 이루어진 크기가 3인 벡터 선언  
std::vector<int> vec2(5);    // 크기가 5인 벡터 선언  
std::vector<int> vec3(10, 1);    // 크기가 10이고 모든 원소가 1로 초기화된 벡터 선언
```

# std::vector

```
#include <vector>
```

```
int main()
{
    std::vector<int> vec0;    // 크기가 0인 벡터 선언

    vec0.push_back(1);
    vec0.emplace_back(3);

    vec0.insert(vec0.begin(), 0);
    vec0.emplace(vec0.begin() + 2, 2);

    for (auto v : vec0)
    {
        std::cout << v << std::endl;
    }
}
```

Microsoft Visual

0  
1  
2  
3

# std::vector

```
#include <vector>
```

```
int main()
{
    std::vector<int> vec0;    // 크기가 0인 벡터 선언

    vec0.push_back(1);
    vec0.emplace_back(3);

    vec0.insert(vec0.begin(), 0);
    vec0.emplace(vec0.begin() + 2, 2);

    vec0.pop_back();

    for (auto v : vec0)
    {
        std::cout << v << std::endl;
    }
}
```

Microsoft V

0  
1  
2

# std::vector

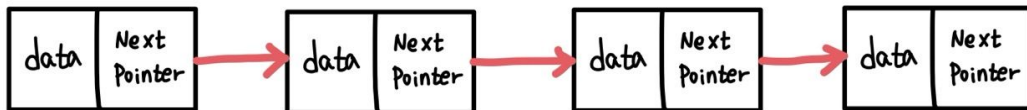
```
#include <vector>
```

- 유용한 멤버 함수
  - `clear()` : 모든 원소 제거.
  - `reserve(capacity)` : 벡터에서 사용할 용량 지정. 벡터의 크기 변경 XXX
  - `shrink_to_fit()` : 여분의 메모리 공간을 해제하는 용도로 사용. 용량이 크기와 같게 설정됨.
- 매개변수에서 데이터 타입 다음에 할당자 전달 가능 : `vector<데이터 타입, 할당자>`
- 할당자
  - 초기화 되지 않은 메모리 공간에 객체를 직접 할당할 수 없지만, 할당자 클래스의 멤버 함수 혹은 관련 함수가 초기화 되지 않은 공간에 객체를 저장할 수 있도록 함.
  - `allocate()`, `deallocate()`, `construct()`, `destroy()` 함수 필요.
  - 자세한 사항은 찾아보세요~!

# Singly Linked List / 단일연결리스트

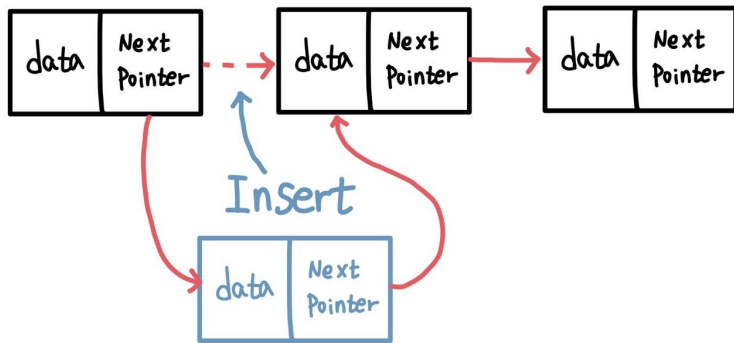
- 배열과 벡터 같은 연속된 자료 구조는 데이터 중간에 자료를 추가하거나 삭제하는 작업 비효율적.
- **Singly Linked List / 단일연결리스트**
  - 각 노드가 데이터와 포인터를 가지고 한 줄로 연결되어 있는 방식.
  - 추가 / 삭제 시간 복잡도  $O(1)$ .
  - 특정 위치의 데이터 검색 시간 복잡도  $O(n)$ .

Node



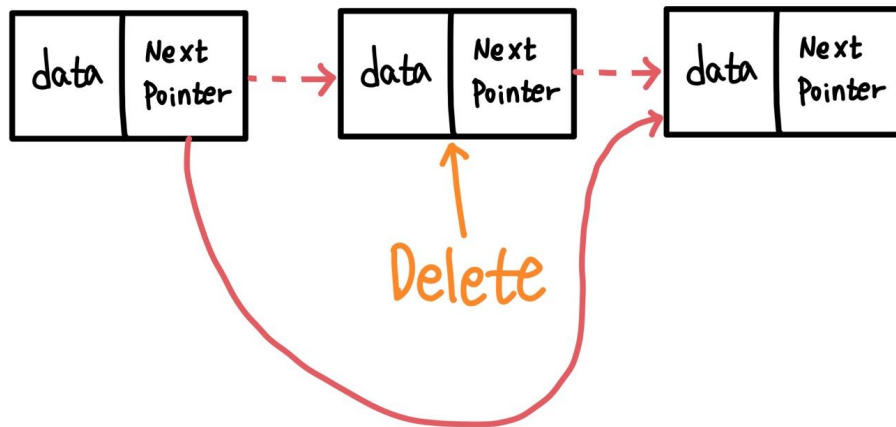
# Singly Linked List / 단일 연결리스트

- 데이터 추가
  - 새로운 데이터 노드 생성.
  - 새로 생성한 노드의 포인터에 데이터를 추가할 위치 다음 노드 연결.
  - 데이터를 추가할 위치 앞 노드의 포인터에 새로 생성한 노드 연결.



# Singly Linked List / 단일 연결리스트

- 데이터 삭제
  - 삭제할 노드 앞 노드의 포인터에 삭제할 노드 뒤 노드 연결.
  - 삭제할 노드 삭제.



# Singly Linked List / 단일 연결리스트

```
#include <iostream>
using namespace std;

template <typename T>
class List
{
private:
    struct Node
    {
        T data;    // 데이터 / 자료

        Node* next;    // 다음 노드를 가리키는 포인터

        Node(T data) : data(data), next(nullptr) {}
    };
```

```
    Node* head;    // 리스트의 첫 번째 노드

public:
    List() { head = nullptr; }    // 생성자
```



# Singly Linked List / 단일 연결리스트

```
// 맨 앞에 추가
void push_front(T data)
{
    Node* node = new Node(data);    // 새로운 노드 생성

    if (head == nullptr)    // 리스트가 비어있다면
    {
        head = node;        // 새로운 노드를 리스트의 첫 번째 노드로 설정
    }
    else
    {
        // 리스트가 비어있지 않으면
        node->next = head;    // 새로운 노드의 포인터를 리스트의 첫 번째 노드로 연결
        head = node;        // 리스트의 첫 번째 노드를 새로운 노드로 설정
    }
}
```

# Singly Linked List / 단일 연결리스트

// 특정 위치 뒤에 추가

```
void insert_after(Node* prev, T data)
```

```
{
```

```
    Node* node = new Node(data);
```

// 새로운 노드 생성

```
    node->next = prev->next;
```

// 새로운 노드의 포인터를 추가하려는 위치 다음 노드와 연결

```
    prev->next = node;
```

// 추가하려는 위치 앞 노드의 포인터를 새로운 노드로 연결

```
}
```

# Singly Linked List / 단일 연결 리스트

```
// 맨 뒤에 추가
void append(T data)
{
    Node* node = new Node(data);    // 새로운 노드 생성

    if (head == nullptr)            // 리스트가 비어있다면
    {
        head = node;                // 새로운 노드를 첫 번째 노드로 설정
        return;
    }

    // 리스트가 비어있지 않다면
    Node* current = head;           // 현재 노드를 가리키는 포인터 변수
    while (current->next != nullptr) // 현재 리스트에서 맨 뒤에 있는 노드 찾기
    {
        current = current->next;
    }
    current->next = node;            // 맨 뒤에 있는 노드의 포인터에 새로운 노드 연결
}
```

## Singly Linked List / 단일 연결리스트

```
// 맨 앞 삭제
void pop_front()
{
    if (head == nullptr) return;

    Node* temp = head;           // 삭제할 노드
    head = head->next;           // 리스트의 첫 번째 노드를 그 다음 노드로 설정
    delete temp;                // 노드 삭제
}
```

# Singly Linked List / 단일 연결리스트

```
// 특정 위치 뒤 삭제
void erase_after(Node* prev)
{
    Node* temp = prev->next;           // 특정 위치 뒤에 있는 노드가 삭제할 노드
    prev->next = prev->next->next;      // 특정 위치 노드의 포인터를 삭제할 노드 포인터로 연결
    delete temp;                      // 노드 삭제
}
```

# Singly Linked List / 단일 연결리스트

```
// data 찾아서 삭제
void remove(T data)
{
    Node* current = head;      // 현재 가리키고 있는 노드
    while (current->next != nullptr && current->next->data != data)    // data를 가지고 있는 노드의 앞 노드 찾기
    {
        current = current->next;
    }

    if (current->next != nullptr)
    {
        Node* temp = current->next;    // 삭제할 노드는 현재 노드의 다음 노드
        current->next = current->next->next;    // 현재 노드의 포인터는 삭제할 노드의 포인터와 연결
        delete temp;    // 노드 삭제
    }
}
```

# Singly Linked List / 단일 연결리스트

```
// 특정 노드 찾기
Node* findNode(T data)
{
    Node* current = head;    // 현재 가리키고 있는 노드
    while (current != nullptr && current->data != data)    // 데이터를 가진 노드 찾기
    {
        current = current->next;
    }

    return current;
}
```

## Singly Linked List / 단일 연결리스트

```
// 리스트 출력
void print()
{
    Node* current = head;    // 현재 노드
    while (current != nullptr) // 리스트의 마지막 노드까지 반복
    {
        cout << current->data << " ";    // 데이터 출력
        current = current->next;
    }
    cout << endl;
}
```



# Singly Linked List / 단일연결리스트

```
int main()
{
    List<int> list;

    list.push_front(0);
    list.print();

    list.append(2);
    list.print();

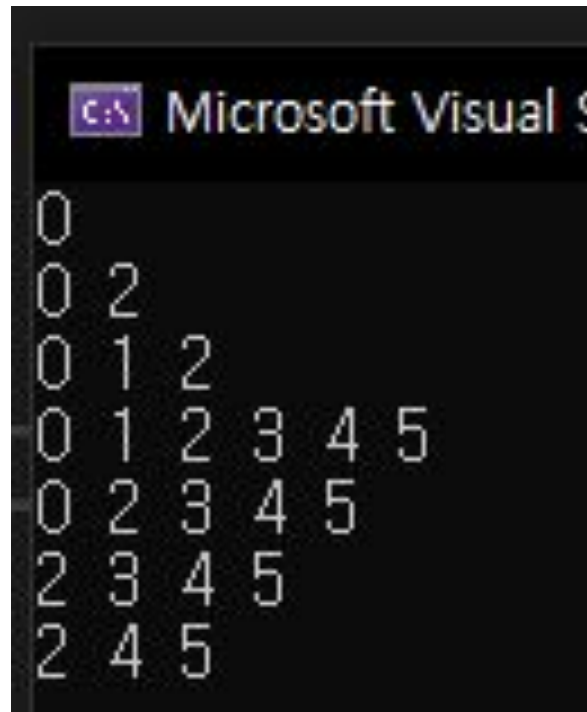
    list.insert_after(list.findNode(0), 1);
    list.print();

    list.append(3);
    list.append(4);
    list.append(5);
    list.print();

    list.erase_after(list.findNode(0));
    list.print();

    list.pop_front();
    list.print();

    list.remove(3);
    list.print();
}
```



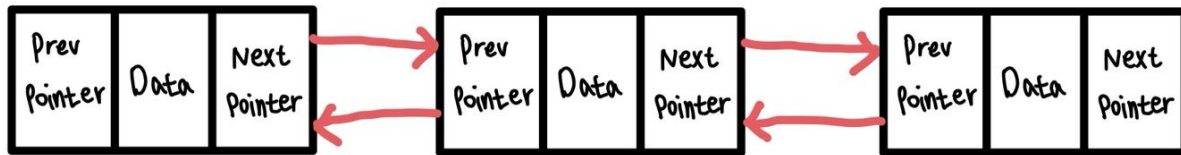
The screenshot shows the output of the program in a console window. The output consists of eight lines of numbers, each representing the state of the singly linked list after a specific operation. The numbers are displayed in a staggered, triangular format to represent the sequence of nodes in the list.

```
C:\> Microsoft Visual S
0
0 2
0 1 2
0 1 2 3 4 5
0 2 3 4 5
2 3 4 5
2 4 5
```

# Doubly Linked List / 이중연결리스트

- 양쪽 방향으로 연결된 리스트.
- 역방향으로 이동 가능.

Node



# 과제 1

- 자연수 뒤집어 배열로 만들기.

## 문제 설명

자연수  $n$ 을 뒤집어 각 자리 숫자를 원소로 가지는 배열 형태로 리턴해주세요. 예를들어  $n$ 이 12345이면 [5,4,3,2,1]을 리턴합니다.

## 제한 조건

- $n$ 은 10,000,000,000이하인 자연수입니다.

## 입출력 예

n	return
12345	[5,4,3,2,1]

## 과제2

- 단일연결리스트 구현.
- 이중연결리스트 구현.