

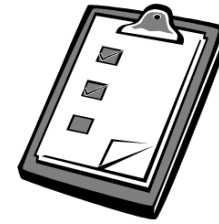
3. 연결 리스트

1) 연결 리스트

리스트

■ 리스트(list)란?

- 순서를 가진 데이터의 모임. 목록.
- (e.g.) TO DO 리스트, 요일(일요일, 월요일, ..., 토요일), etc.



■ 리스트의 주요 연산

- 원소의 참조, 삽입(insert), 삭제(remove), 검색(search), etc.

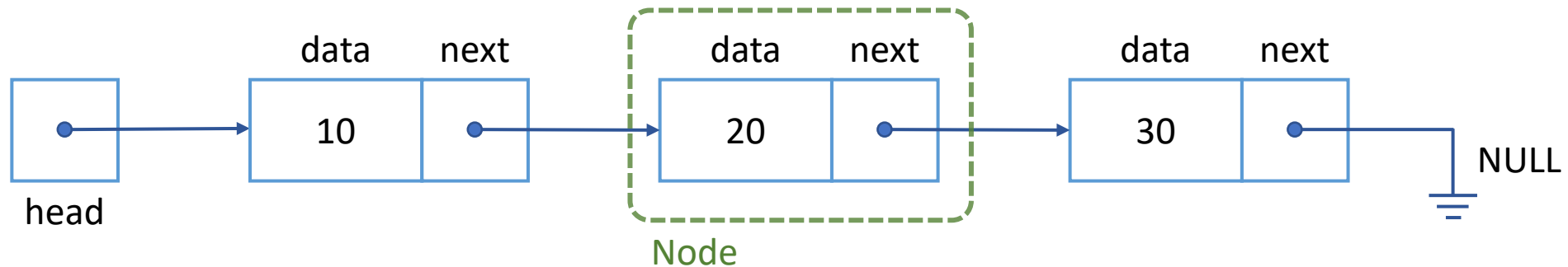
■ 대표적인 리스트 구현 방법

	배열	연결 리스트
저장 공간	연속된 메모리 공간	임의의 메모리 공간
원소의 삽입&삭제	비효율적	효율적
구현	쉬움	어려움

연결 리스트

■ 연결 리스트(linked list)란?

- 데이터와 링크로 구성된 노드(node)가 연결되어 있는 자료 구조
 - 데이터(data): 정수, 문자열, 복합 자료형 등
 - 링크(link, next): 다음 노드를 가리키는 포인터
 - 노드(node): 데이터와 링크로 이루어진 연결 리스트 구성 단위



```
Node* head;
```

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

연결 리스트 구현하기

■ 연결 리스트 클래스

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

```
class LinkedList
{
private:
    Node* head;

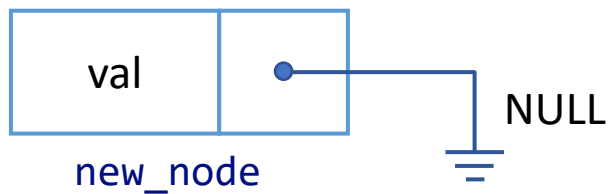
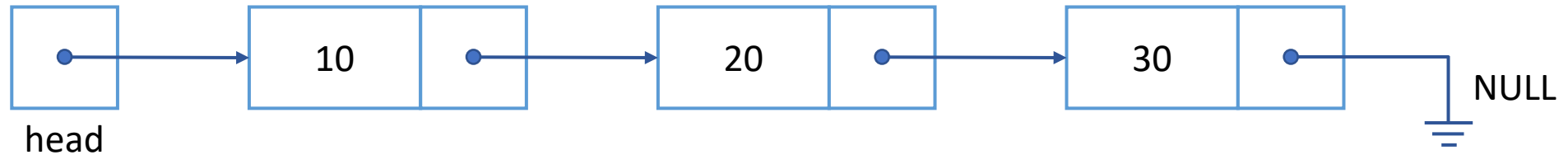
public:
    LinkedList() : head(NULL) {}
    ~LinkedList();

    void push_front(int val);
    void pop_front();

    void print_all();
    bool empty();
    ...
};
```

연결 리스트 구현하기

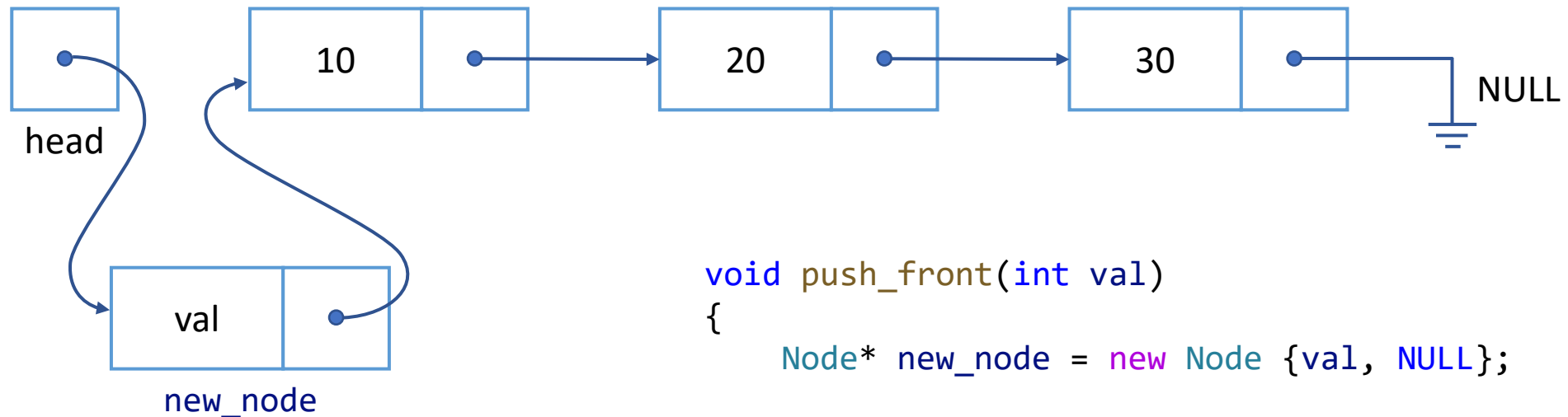
- 연결 리스트 맨 앞에 노드 삽입하기



```
void push_front(int val)
{
    Node* new_node = new Node {val, NULL};
```

연결 리스트 구현하기

■ 연결 리스트 맨 앞에 노드 삽입하기



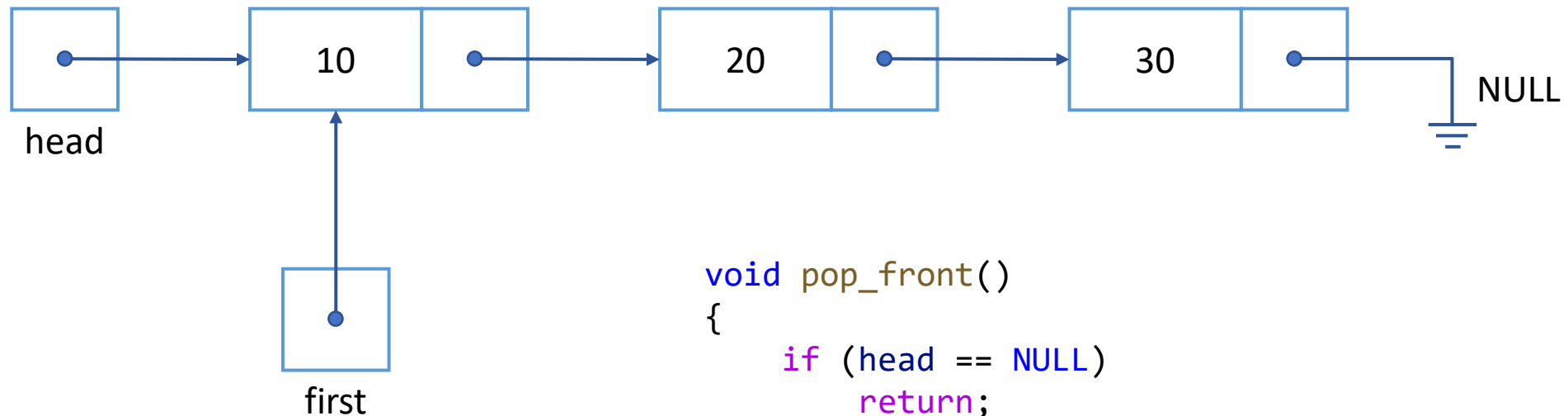
```
void push_front(int val)
{
    Node* new_node = new Node {val, NULL};

    if (head != NULL)
        new_node->next = head;

    head = new_node;
}
```

연결 리스트 구현하기

- 연결 리스트 맨 앞 노드 삭제하기

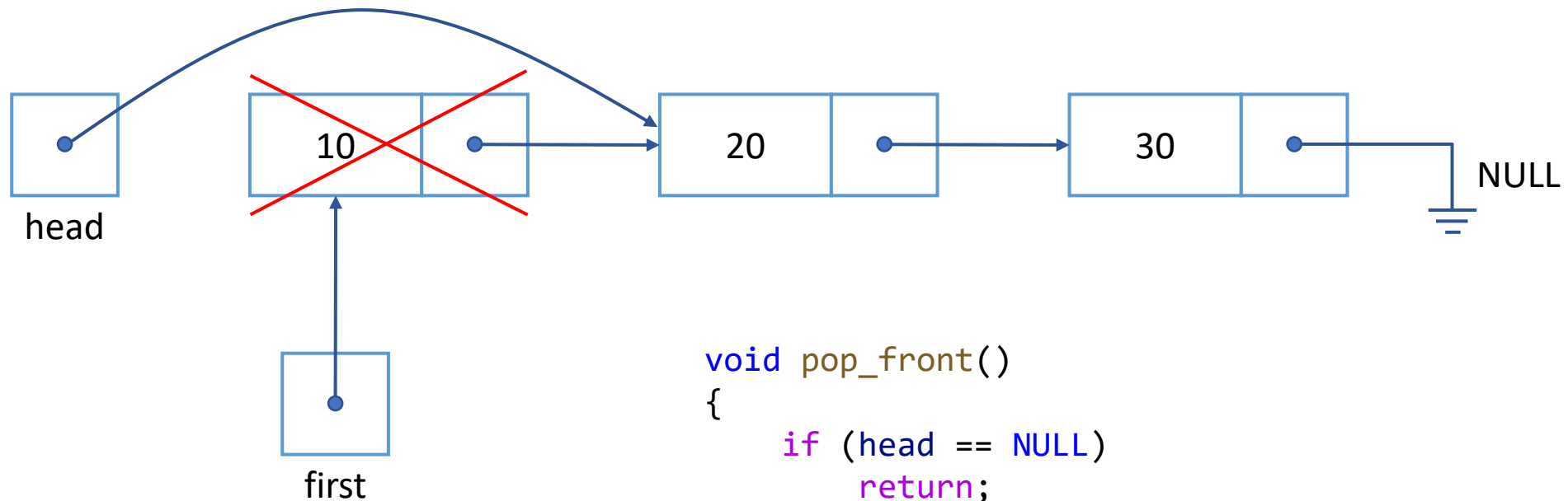


```
void pop_front()
{
    if (head == NULL)
        return;

    Node* first = head;
```

연결 리스트 구현하기

■ 연결 리스트 맨 앞 노드 삭제하기

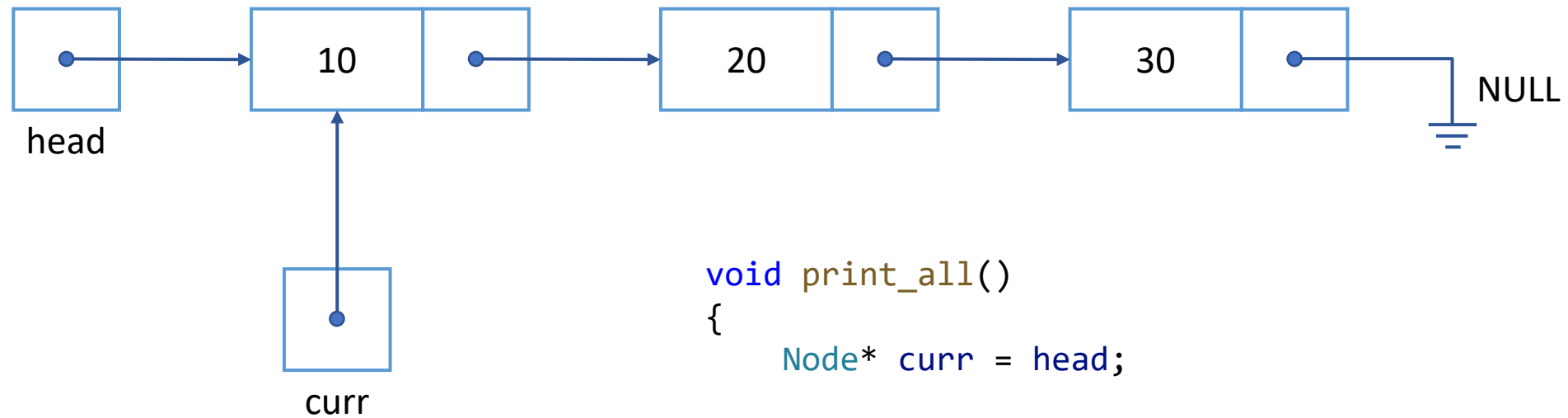


```
void pop_front()
{
    if (head == NULL)
        return;

    Node* first = head;
    head = head->next;
    delete first;
}
```


연결 리스트 구현하기

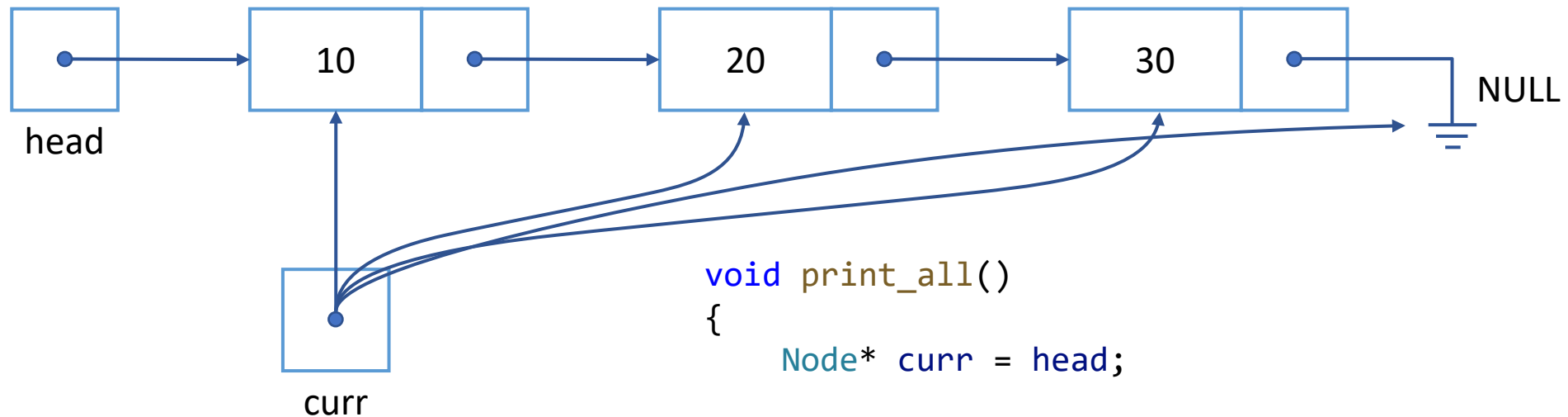
- 연결 리스트 전체 데이터 출력하기



```
void print_all()
{
    Node* curr = head;
```

연결 리스트 구현하기

■ 연결 리스트 전체 데이터 출력하기



```
void print_all()
{
    Node* curr = head;

    while (curr != NULL) {
        std::cout << curr->data << ", ";
        curr = curr->next;
    }

    std::cout << std::endl;
}
```

연결 리스트 구현하기

- 연결 리스트가 비어 있는지 확인

```
bool empty() const
{
    return head == NULL;
}
```

- 연결 리스트 제거

```
~LinkedList()
{
    while (!empty()) {
        pop_front();
    }
}
```

연결 리스트 구현하기

- 구현한 연결 리스트 클래스 동작 확인

```
int main()
{
    LinkedList ll;
    ll.push_front(10);
    ll.push_front(20);
    ll.push_front(30);
    ll.print_all();

    ll.pop_front();
    ll.print_all();

    ll.push_front(40);
    ll.print_all();
}
```



```
cmd 명령 프롬프트
30, 20, 10,
20, 10,
40, 20, 10,
```

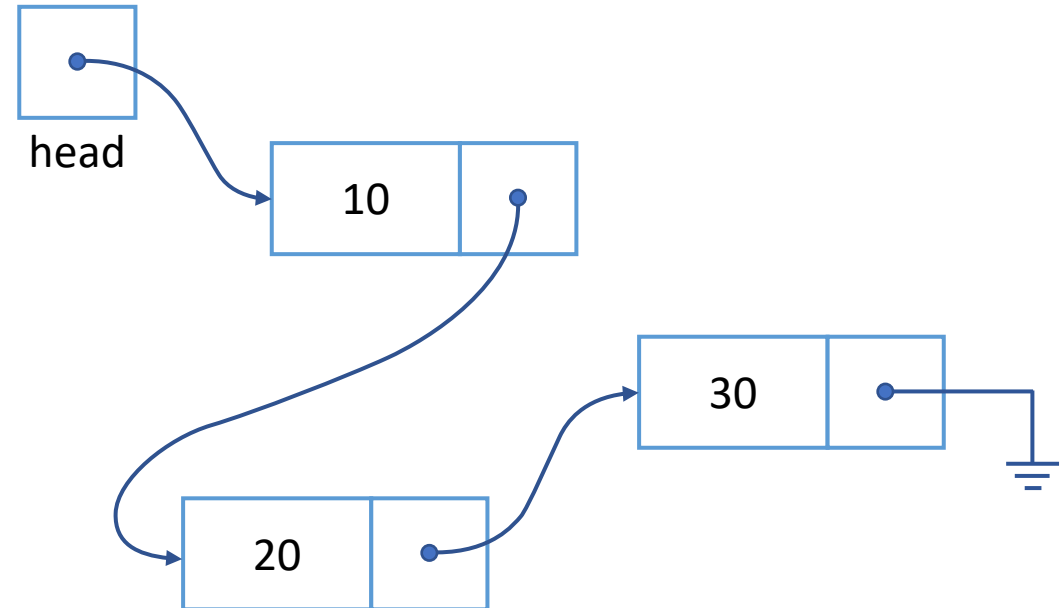
연결 리스트의 장단점

■ 연결 리스트의 장점

- 임의의 위치에 원소의 삽입&삭제가 효율적: $O(1)$
- 크기 제한이 없음

■ 연결 리스트의 단점

- 임의의 원소 접근이 비효율적: $O(N)$
- 링크를 위한 여분의 공간 사용
- 구현이 복잡



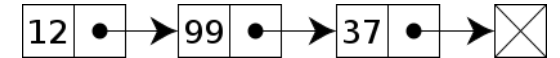
3. 연결 리스트

2) 이중 연결 리스트

연결 리스트 종류

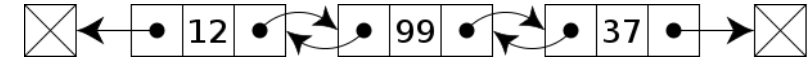
■ 단순 연결 리스트(singly linked list)

- 다음 노드에 대한 링크만 가지고 있는 연결 리스트
- 한쪽 방향으로만 순회(traverse)가 가능 (단방향 연결 리스트)



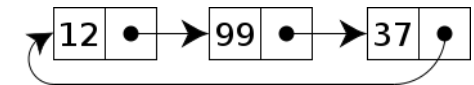
✓ ■ 이중 연결 리스트(doubly linked list)

- 이전 노드와 다음 노드에 대한 링크를 가지고 있는 연결 리스트
- 양방향 순회가 가능 (양방향 연결 리스트)



■ 원형 연결 리스트(circular linked list)

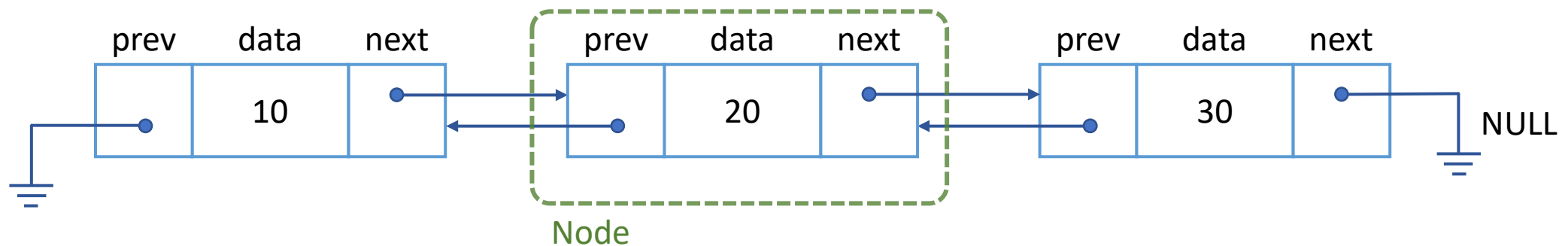
- 일반적인 연결 리스트의 마지막 노드 링크가 처음 노드를 가리키도록 구성된 자료 구조
- 처음 노드가 다시 나타나면 순회를 멈춤



Images from https://en.wikipedia.org/wiki/Linked_list

이중 연결 리스트

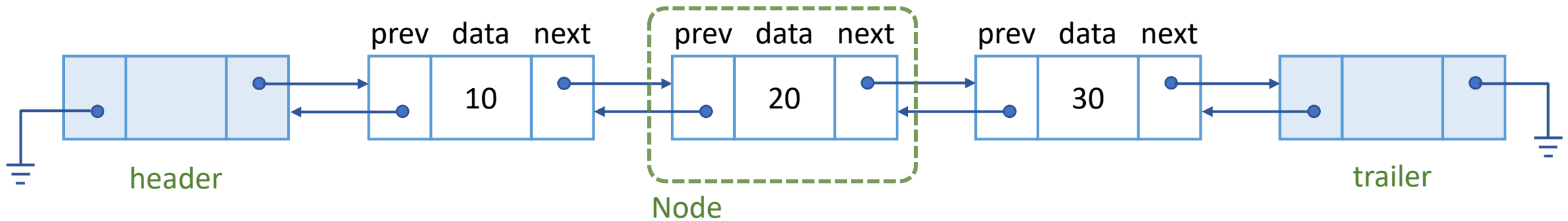
■ 이중 연결 리스트의 구조



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


이중 연결 리스트

■ 이중 연결 리스트의 구조



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

```
class DoublyLinkedList
{
    int count;
    Node* header;
    Node* trailer;
    ...
};
```

이중 연결 리스트 구현하기

■ 이중 연결 리스트 클래스

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

```
class DoublyLinkedList
{
private:
    int count;
    Node* header;
    Node* trailer;


public:
    DoublyLinkedList();
    ~DoublyLinkedList();

    void insert(Node* p, int val);
    void erase(Node* p);

    void print_all();
    void print_reverse();
    ...
};
```

```
void push_front(int val);
void push_back(int val);

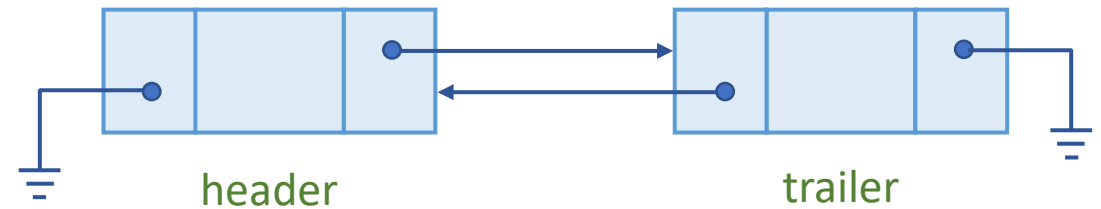
void pop_front();
void pop_back();
```



이중 연결 리스트 구현하기

■ 이중 연결 리스트 생성

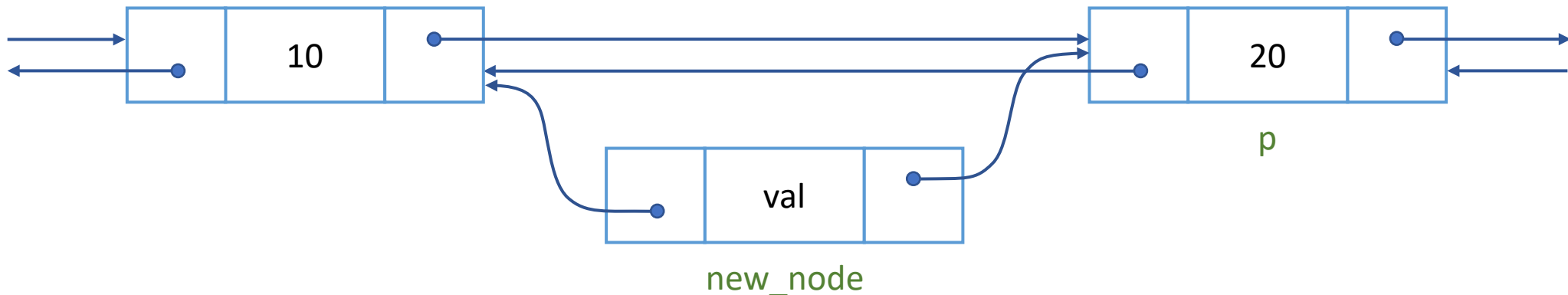
```
DoublyLinkedList()
{
    count = 0;
    header = new Node {0, NULL, NULL};
    trailer = new Node {0, NULL, NULL};
    header->next = trailer;
    trailer->prev = header;
}
```



이중 연결 리스트 구현하기

- 이중 연결 리스트에 원소 삽입

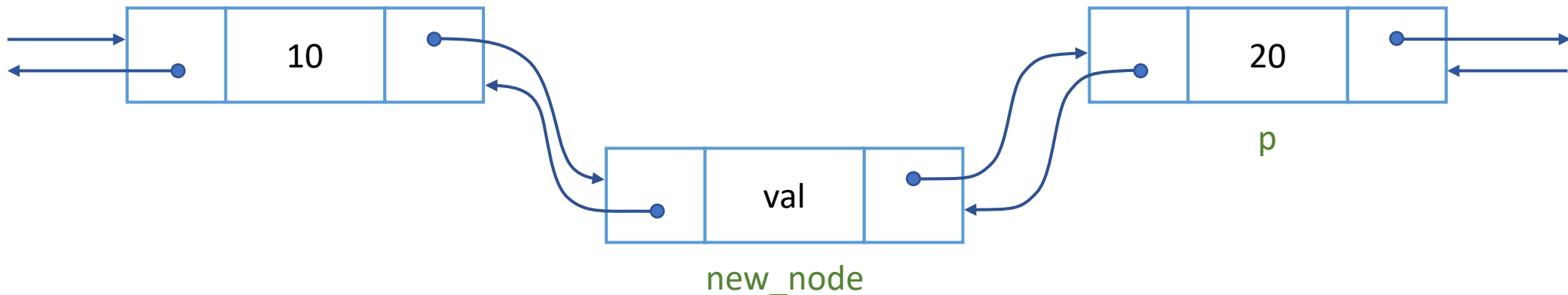
```
void insert(Node* p, int val)
{
    Node* new_node = new Node {val, p->prev, p};
```



이중 연결 리스트 구현하기

■ 이중 연결 리스트에 원소 삽입

```
void insert(Node* p, int val)
{
    Node* new_node = new Node {val, p->prev, p};
    new_node->prev->next = new_node;
    new_node->next->prev = new_node;
    count++;
}
```



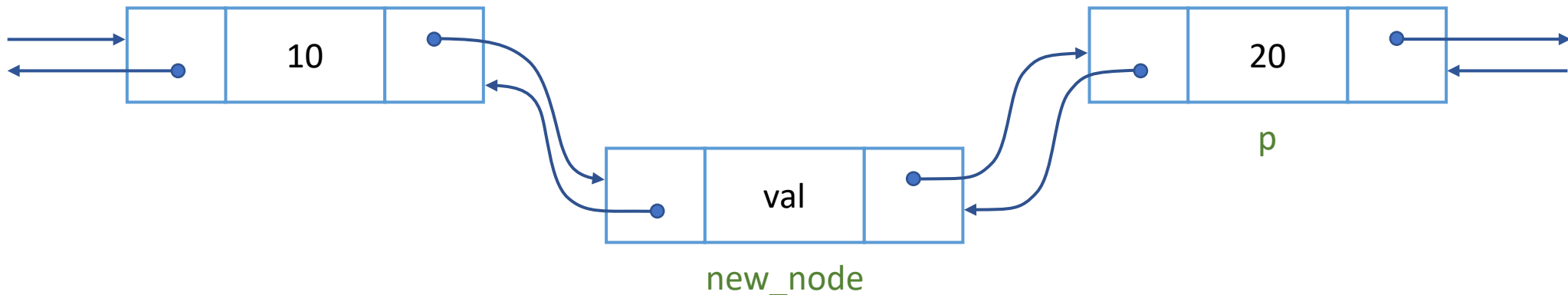
이중 연결 리스트 구현하기

■ 이중 연결 리스트에 원소 삽입

```
void insert(Node* p, int val)
{
    Node* new_node = new Node {val, p->prev, p};
    new_node->prev->next = new_node;
    new_node->next->prev = new_node;
    count++;
}
```

```
void push_front(int val)
{
    insert(header->next, val);
}

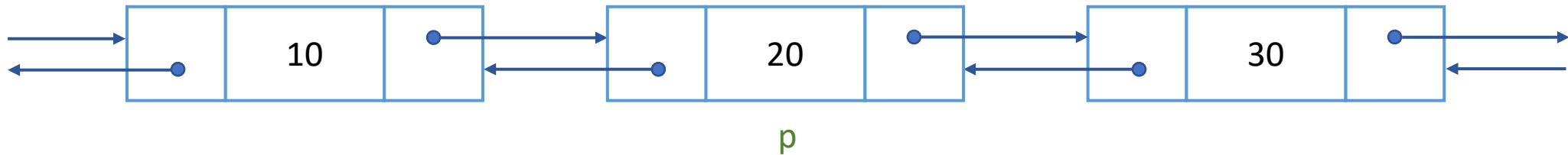
void push_back(int val)
{
    insert(trailer, val);
}
```



이중 연결 리스트 구현하기

■ 이중 연결 리스트에서 노드 삭제하기

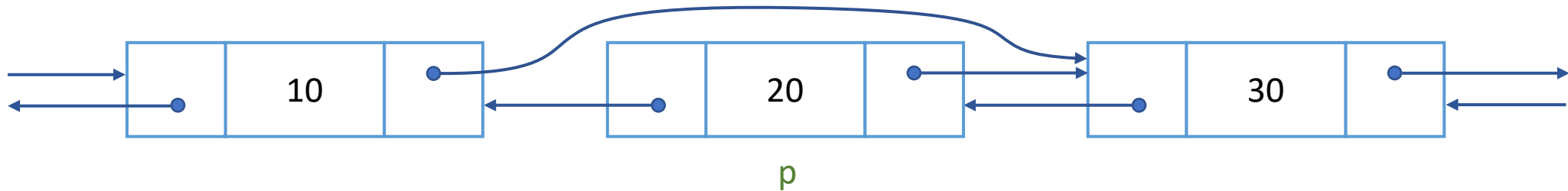
```
void erase(Node* p)
{
    p->prev->next = p->next;
```



이중 연결 리스트 구현하기

■ 이중 연결 리스트에서 노드 삭제하기

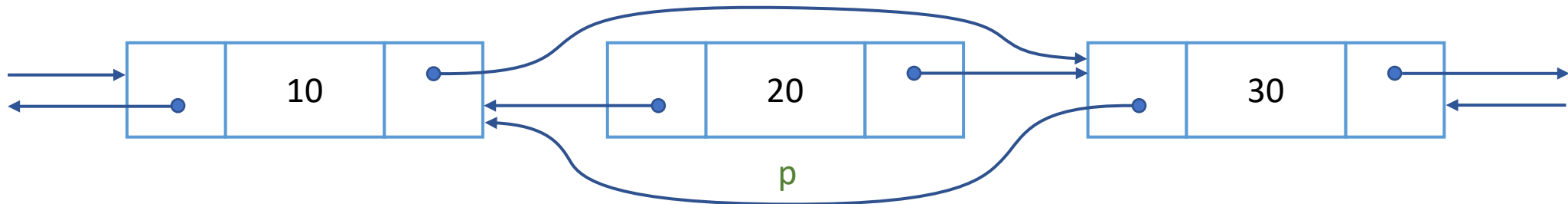
```
void erase(Node* p)
{
    p->prev->next = p->next;
}
```



이중 연결 리스트 구현하기

■ 이중 연결 리스트에서 노드 삭제하기

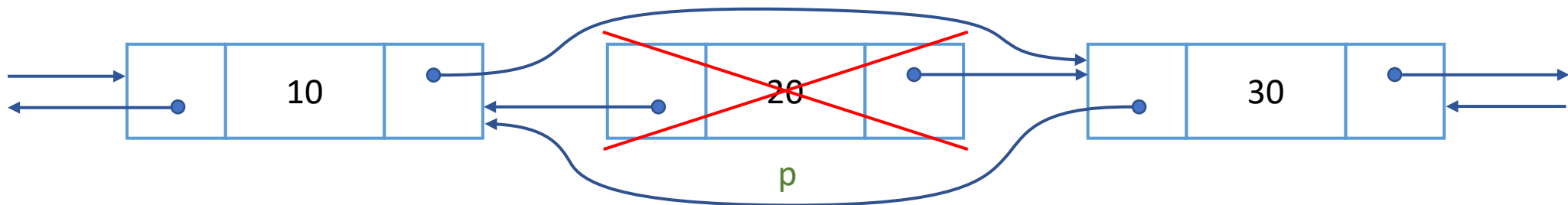
```
void erase(Node* p)
{
    p->prev->next = p->next;
    p->next->prev = p->prev;
}
```



이중 연결 리스트 구현하기

■ 이중 연결 리스트에서 노드 삭제하기

```
void erase(Node* p)
{
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    count--;
}
```



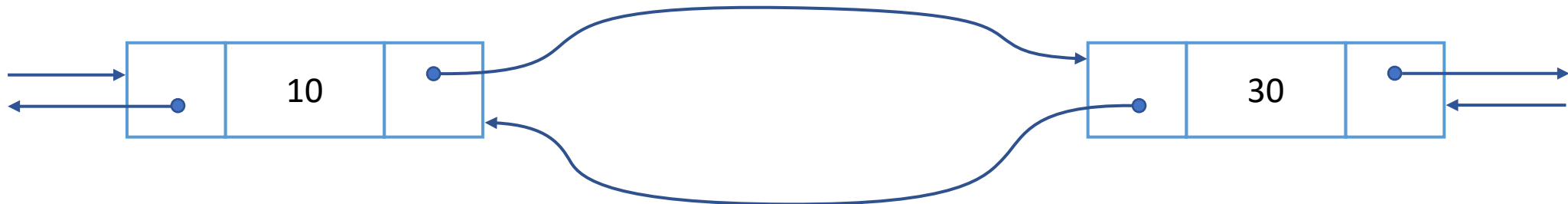
이중 연결 리스트 구현하기

■ 이중 연결 리스트에서 노드 삭제하기

```
void erase(Node* p)
{
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    count--;
}
```

```
void pop_front()
{
    if (!empty())
        erase(header->next);
}

void pop_back()
{
    if (!empty())
        erase(trailer->prev);
}
```



이중 연결 리스트 구현하기

- 이중 연결 리스트 전체 데이터 출력하기

```
void print_all()
{
    Node* curr = header->next;

    while (curr != trailer) {
        std::cout << curr->data << ", ";
        curr = curr->next;
    }

    std::cout << std::endl;
}
```

- 이중 연결 리스트 전체 데이터를 역순으로 출력하기

```
~DoublyLinkedList()
{
    while (!empty()) {
        pop_front();
    }

    delete header;
    delete trailer;
}
```

이중 연결 리스트 구현하기

■ 기타 멤버 함수

```
bool empty() const
{
    return count == 0;
}

int size() const
{
    return count;
}
```

■ 이중 연결 리스트 제거

```
~DoublyLinkedList()
{
    while (!empty()) {
        pop_front();
    }

    delete header;
    delete trailer;
}
```

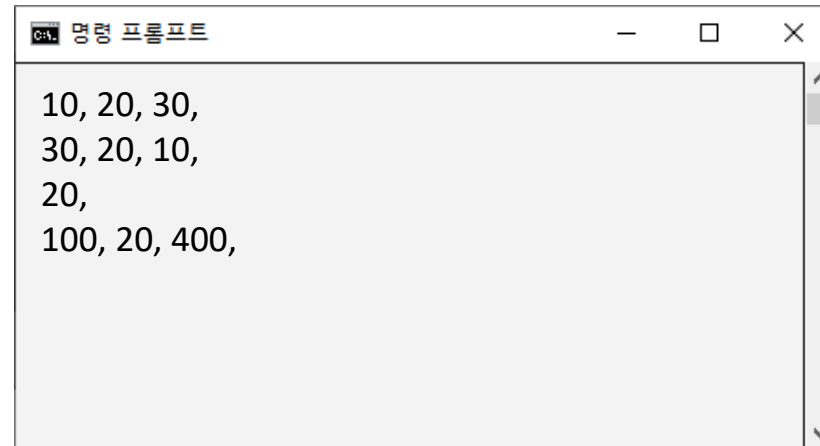
이중 연결 리스트 구현하기

- 구현한 이중 연결 리스트 클래스 동작 확인

```
int main()
{
    DoublyLinkedList ll;
    ll.push_back(10);
    ll.push_back(20);
    ll.push_back(30);
    ll.print_all();
    ll.print_reverse();

    ll.pop_front();
    ll.pop_back();
    ll.print_all();

    ll.push_front(100);
    ll.push_back(400);
    ll.print_all();
}
```



cmd 명령 프롬프트

10, 20, 30,
30, 20, 10,
20,
100, 20, 400,

이중 연결 리스트의 장단점

- (단순 연결 리스트 대비) 이중 연결 리스트의 장점
 - 링크가 양방향이므로 양방향 검색이 가능
- (단순 연결 리스트 대비) 이중 연결 리스트의 단점
 - 이전 노드 링크를 위한 여분의 공간 사용
 - 데이터의 삽입과 삭제 구현이 더 복잡

3. 연결 리스트

3) 향상된 이중 연결 리스트 클래스

향상된 이중 연결 리스트 클래스

- DoublyLinkedList 클래스에 추가할 기능
 - 반복자(iterator) 지원
 - 데이터 검색 기능
 - 범용 데이터 저장을 위한 클래스 템플릿 작성

반복자 지원

- 반복자 클래스를 중첩 클래스 형태로 정의

```
class iterator
{
private:
    Node* ptr;

public:
    iterator() : ptr(NULL) {}
    iterator(Node* p) : ptr(p) {}

    T& operator*() { return ptr->data; }

    iterator& operator++() // ++it
    {
        ptr = ptr->next;
        return *this;
    }
}
```

```
iterator& operator--() // --it
{
    ptr = ptr->prev;
    return *this;
}

bool operator==(const iterator& it)
{
    return ptr == it.ptr;
}

bool operator!=(const iterator& it)
{
    return ptr != it.ptr;
}

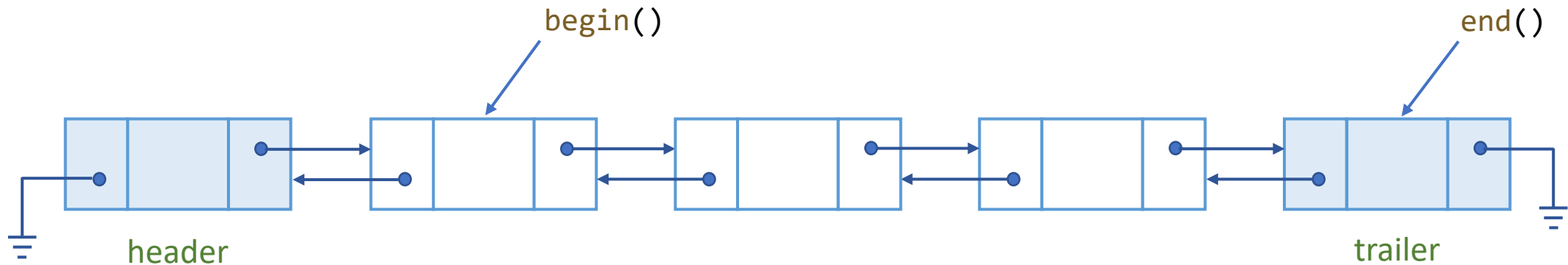
friend class DoublyLinkedList;
};
```

반복자 지원

- DoublyLinkedList 클래스에 begin() 과 end() 멤버 함수 추가

```
iterator begin() const
{
    return iterator(header->next);
}
```

```
iterator end() const
{
    return iterator(trailer);
}
```



반복자 지원

- DoublyLinkedList 클래스에서 insert(), push_front(), push_back() 함수 수정

```
void insert(const iterator& pos, const int val)
{
    Node* curr = pos.ptr;
    Node* new_node = new Node {val, curr->prev, curr};
    new_node->prev->next = new_node;
    new_node->next->prev = new_node;
    count++;
}

void push_front(const T& data)
{
    insert(begin(), data);
}

void push_back(const T& data)
{
    insert(end(), data);
}
```

반복자 지원

- DoublyLinkedList 클래스에서 erase(), pop_front(), pop_back() 함수 수정

```
void erase(const iterator& pos)
{
    Node<T>* curr = pos.ptr;
    curr->prev->next = curr->next;
    curr->next->prev = curr->prev;
    delete curr;
    count--;
}

void pop_front()
{
    if (!empty())    erase(begin());
}

void pop_back()
{
    if (!empty())    erase(--end());
}
```

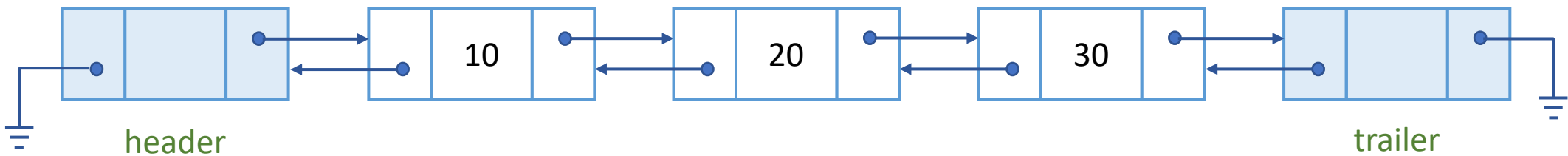
데이터 검색 기능

- DoublyLinkedList 클래스에 find() 멤버 함수 추가

```
iterator find(const int val)
{
    Node* curr = header->next;

    while (curr->data != val && curr != trailer)
        curr = curr->next;

    return iterator(curr);
}
```



범용 데이터 저장을 위한 클래스 템플릿 작성

- DoublyLinkedList 클래스를 클래스 템플릿으로 변환하기
 - Node와 DoublyLinkedList 클래스를 모두 **typename T** 를 사용하는 클래스 템플릿으로 변경
 - Node와 DoublyLinkedList 클래스에서 데이터를 표현하는 int를 T로 바꾸고, Node를 Node<T>로 변경

```
template <typename T>
struct Node
{
    T data;
    Node* prev;
    Node* next;
};
```

```
template <typename T>
class DoublyLinkedList
{
private:
    int count;
    Node<T>* header;
    Node<T>* trailer;

    ...
};
```

향상된 이중 연결 리스트 클래스

■ 향상된 이중 연결 리스트 클래스 동작 확인

```
int main()
{
    DoublyLinkedList<int> ll;
    ll.push_back(10);
    ll.push_back(20);
    ll.push_back(30);

    auto it = ll.find(20);
    if (it != ll.end())
        ll.insert(it, 50);

    // ll: header -> 30 -> 50 -> 20 -> 10 -> trailer

    for (const auto& a : ll)
        cout << a << ", ";
    cout << endl;
}
```



3. 연결 리스트

4) `std::list`와 `std::forward_list`

std::list와 std::forward_list

■ std::list

- 이중 연결 리스트를 구현한 컨테이너

```
template <class T, class Allocator = std::allocator<T>>  
class list;
```

- 어느 위치라도 원소의 삽입 또는 삭제를 상수 시간으로 처리: $O(1)$
- 그러나 특정 위치에 곧바로 접근할 수 없음
 - std::vector처럼 [] 연산자를 이용한 랜덤 액세스는 지원 안 함
 - begin(), end() 등의 함수로 얻은 (양방향) 반복자와 ++, -- 연산자로 위치 이동
- <list>에 정의되어 있음

std::list와 std::forward_list

■ std::list 주요 연산

함수 이름	설명
front(), back()	리스트의 맨 앞 또는 맨 마지막 원소 참조
begin(), cbegin(), rbegin(), crbegin(), end(), cend(), rend(), crend()	(양방향) 반복자 반환
insert(), push_front(), push_back(), emplace(), emplace_front(), emplace_back()	리스트에 원소 삽입
clear(), erase(), pop_front(), pop_back()	리스트에서 원소 삭제
splice()	다른 리스트의 원소를 이동
remove(), remove_if()	특정 조건을 만족하는 원소를 삭제
reverse()	원소 순서를 역순으로 변경
unique()	연속적으로 나타나는 중복 원소를 삭제
sort()	정렬

<https://en.cppreference.com/w/cpp/container/list>

std::list와 std::forward_list

■ std::list 예제 코드

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> l1;
    l1.push_back(10); // 10
    l1.push_back(20); // 10, 20

    list<int> l2 {10, 20, 30, 40};

    l2.splice(l2.end(), l1); // 10, 20, 30, 40, 10, 20

    l2.sort(); // 10, 10, 20, 20, 30, 40

    l2.unique(); // 10, 20, 30, 40
}
```

std::list와 std::forward_list

■ std::forward_list

- 단순 연결 리스트를 구현한 컨테이너

```
template <class T, class Allocator = std::allocator<T>>  
class forward_list;
```

- begin() 함수로 (순방향) 반복자를 얻을 수 있고, 오직 ++ 연산만 사용 가능
- std::list보다 빠르게 동작하고, 적은 메모리를 사용
- C++11부터 지원
- <forward_list>에 정의되어 있음

std::list와 std::forward_list

■ std::list와 std::forward_list 차이점

std::list	std::forward_list	설명
front(), back()	front()	forward_list는 front() 함수만 제공
begin(), end()	before_begin(), begin(), end()	forward_list는 before_begin() 함수를 추가로 제공하고, 순방향 반복자만 얻을 수 있음
insert(), emplace(), erase()	insert_after(), emplace_after(), erase_after()	forward_list는 특정 위치 다음에 원소를 삽입하거나 삭제하는 함수를 제공
push_front(), push_back(), emplace_front(), emplace_back(), pop_front(), pop_back()	push_front(), emplace_front(), pop_front()	forward_list는 연결 리스트 맨 앞에서만 원소를 삽입하거나 삭제할 수 있음
size()	N/A	forward_list는 std::distance() 함수를 활용하여 원소 개수를 알 수 있음

std::list와 std::forward_list

■ std::forward_list 예제 코드

```
#include <iostream>
#include <forward_list>

using namespace std;

int main()
{
    forward_list<int> l1 {10, 20, 30, 40};
    l1.push_front(40); // 40, 10, 20, 30, 40
    l1.push_front(30); // 30, 40, 10, 20, 30, 40

    int cnt = distance(l1.begin(), l1.end()); // 6

    l1.remove(40); // 30, 10, 20, 30
    l1.remove_if([](int n) { return n > 20; }); // 10, 20

    for (auto a : l1)
        cout << a << ", ";
    cout << endl;
}
```