Data Structures and Algorithms

Linked List

Linked List

- **node.h**
  - Node<T>
  - Iterator<T>
    - operator!=(), operator==(), operator*(), operator++()
  - functions
    - preorder(), postorder(), postorder2(), inorder()

- **list.h**
  - List<T>
    - clear(), push_front(), push_back(), pop_front(), pop_back()
    - front(), back(), empty(), size()
  - Stack<T>
    - clear(), push(), pop(), front(), empty(), size()
  - Queue<T>
    - clear(), push(), pop(), front(), empty(), size()

- **array_list.h**
  - ArrayList<T>
    - clear(), push_front(), push_back(), pop_front(), pop_back()
    - front(), back(), empty(), size()
  - ArrayStack<T>
    - clear(), push(), pop(), front(), empty(), size()
  - ArrayQueue<T>
    - clear(), push(), pop(), front(), empty(), size()

- **linked_list.h**
  - LinkedListIterative<T>
    - clear(), find(), insert(), remove()
  - LinkedListRecursive<T>
    - clear(), find(), insert(), remove()

- **ordered_linked_list.h**
  - OrderedLinkedListIterative<T>
    - clear(), find(), insert(), remove()
  - OrderedLinkedListRecursive<T>
    - clear(), find(), insert(), remove()

Linked List – Node / Iterator / Traversal

▪ Node

• Singley Linked List

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

• Examples

```
Node<int>* head;

Node<int>* new_node(int data, Node<int>* next = nullptr) {
    Node<int>* node = new Node<int>{ data, next };
    return node;
}
```

▪ Iterator

• C++ STL Iterator

```
template<typename T>
struct Iterator {
    Node<T>* ptr;

    Iterator(Node<T>* ptr = nullptr) { this->ptr = ptr; }
    bool operator!=(const Iterator& iter) const { return ptr != iter.ptr; }
    bool operator==(const Iterator& iter) const { return ptr == iter.ptr; }
    T& operator*() { return ptr->data; }                        // *iter
    Iterator& operator++() { ptr = ptr->next; return *this; }   // ++iter
    Iterator operator++(int) { Iterator iter = *this; ++(*this);  return iter; }
};
```

• begin() / end()

```
Iterator<int> begin() { return Iterator<int>(head); }
Iterator<int> end() { return Iterator<int>(nullptr); }
```

▪ Traversal

• preorder()

```
void preorder(Node<int>* ptr) {
    if (ptr == nullptr) { return; }
    printf("[%d]->", ptr->data);
    preorder(ptr->next);
}
```

• postorder()

```
void postorder(Node<int>* ptr) {
    if (ptr == nullptr) { return; }
    postorder(ptr->next);
    printf("[%d]->", ptr->data);
}
```

• postorder()

```
void preorder2(Node<int>* ptr) {
    std::stack<Node<int>*> S;
    S.push(ptr);

    while (!S.empty()) {
        ptr = S.top(); S.pop();
        printf("[%d]->", ptr->data);
        if (ptr->next != nullptr) { S.push(ptr->next); }
    }
}
```

• for / while Loop

```
    for (auto ptr = head; ptr; ptr = ptr->next) {
        printf("[%d]->", ptr->data);
    }
    for (auto iter = begin(); iter != end(); iter++) {
        printf("[%d]->", *iter);
    }
    auto ptr = head;
    while (ptr != nullptr) {
        printf("[%d]->", ptr->data);
        ptr = ptr->next;
    }
```

List – Linked List / Array

▪ List

- head / tail 대상으로 노드 추가 / 삭제

```
template<typename T>
struct List {
    Node<T>* head = nullptr;
    Node<T>* tail = nullptr;
    int cnt = 0;

    void clear() { while (!empty()) pop_front(); }
    void push_front(const T& data) {
        Node<T>* node = new Node<T>{ data, nullptr };
        if (head == nullptr) { head = tail = node; }
        else { node->next = head; head = node; }
        cnt++;
    }
    void push_back(const T& data) {
        Node<T>* node = new Node<T>{ data, nullptr };
        if (head == nullptr) { head = tail = node; }
        else { tail->next = node; tail = node; }
        cnt++;
    }
    void pop_front() {
        Node<T>* temp = head;
        head = head->next;
        delete temp; cnt--;
        if (head == nullptr) { tail = nullptr; }
    }
    void pop_back() {
        Node<T>* prev = nullptr;
        Node<T>* cur = head;
        while (cur->next != nullptr) { prev = cur; cur = cur->next; }
        tail = prev;
        if (prev == nullptr) { head = nullptr; }
        else { prev->next = nullptr; }
        delete cur; cnt--;
    }
    bool empty() { return head == nullptr; }
    T front() { return head->data; }
    T back() { return tail->data; }
    int size() { return cnt; }
};
```

▪ Stack

- head 노드 기준 push / pop, tail 노드 기준 push / pop

```
template<typename Type, int max_size>
struct ArrayList {
    Type arr[max_size * 2];
    int head = max_size - 1;
    int tail = max_size - 1;
    int cnt = 0;

    void clear() { head = tail = max_size - 1; cnt = 0; }
    void push_back(const Type& data) {
        if (cnt == max_size) return;
        if (head == -1) { head++; }
        arr[++tail] = data; cnt++;
    }
    void push_front(const Type& data) {
        if (cnt == max_size) return;
        if (head == max_size - 1) { tail--; }
        arr[--head] = data; cnt++;
    }
    void pop_back() {
        tail--; cnt--;
        if (tail < head) { head = tail = max_size - 1; }
    }
    void pop_front() {
        head++; cnt--;
        if (tail < head) { head = tail = max_size - 1; }
    }
    bool empty() { return cnt == 0; }  // head == max_size - 1
    Type front() { return arr[head]; }
    Type back() { return arr[tail]; }
    int size() { return cnt; }
};
```

Stack – Linked List / Array

▪ Stack – Linked List

- head 노드 기준 push front / pop front

```
template<typename T>
struct Stack {
    Node<T>* head = nullptr;
    Node<T>* tail = nullptr;
    int cnt = 0;

    void clear() { while (!empty()) pop(); }
    void push(const T& data) {    // push_front()
        Node<T>* node = new Node<T>{ data, nullptr };
        if (head == nullptr) { head = tail = node; }
        else { node->next = head; head = node; }
        cnt++;
    }
    void pop() {                  // pop_front()
        Node<T>* temp = head;
        head = head->next;
        delete temp; cnt--;
        if (head == nullptr) { tail = nullptr; }
    }
    bool empty() { return head == nullptr; }
    T top() { return head->data; }
    int size() { return cnt; }
};
```

▪ Stack – Array

- tail 노드 증가하면서 push / tail 노드 감소하면서 pop

```
template<typename Type, int max_size>
struct ArrayStack {
    Type arr[max_size];
    int head = -1;
    int tail = -1;
    int cnt = 0;

    void clear() { head = tail = -1; cnt = 0; }
    void push(const Type& data) {
        if (cnt == max_size) return;
        if (head == -1) { head++; }
        arr[++tail] = data; cnt++;
    }
    void pop() {
        tail--; cnt--;
        if (tail < head) { head = tail = -1; }
    }
    bool empty() { return cnt == 0; }  // head == - 1
    Type top() { return arr[tail]; }
    int size() { return cnt; }
};
```

Queue – Linked List / Array

▪ Queue – Linked List

- head 노드 기준 push back / pop front

```
template<typename T>
struct Queue {
    Node<T>* head = nullptr;
    Node<T>* tail = nullptr;
    int cnt = 0;

    void clear() { while (!empty()) pop(); }
    void push(const T& data) {    // push_back()
        Node<T>* node = new Node<T>{ data, nullptr };
        if (head == nullptr) { head = tail = node; }
        else { tail->next = node; tail = node; }
        cnt++;
    }
    void pop() {                  // pop_front()
        Node<T>* temp = head;
        head = head->next;
        delete temp; cnt--;
        if (head == nullptr) { tail = nullptr; }
    }
    bool empty() { return head == nullptr; }
    T front() { return head->data; }
    int size() { return cnt; }
};
```

▪ Queue – Array

- head 증가하면서 push / head 감소하면서 pop

```
template<typename Type, int max_size>
struct ArrayQueue {
    Type arr[max_size];
    int head = -1;
    int tail = -1;
    int cnt = 0;

    void clear() { head = tail = -1; cnt = 0; }
    void push(const Type& data) {
        if (cnt == max_size) return;
        if (head == -1) { head++; }
        arr[++tail] = data; cnt++;
    }
    void pop() {
        head++; cnt--;
        if (head > tail) { head = tail = -1; }
    }
    bool empty() { return cnt == 0; }  // head == - 1
    Type front() { return arr[head]; }
    int size() { return cnt; }
};
```

Linked List – Iterative

- **Linked List**
  - **head 노드 / 맨 뒤에 원소 추가**

```
template<typename T>
struct LinkedListIterative{
    Node<T>* head = nullptr;

    void clear();
    Node<T>* find(const T& data);
    void insert(const T& data);
    void remove(const T& data);

    Iterator<T> begin() { return Iterator<T>(head); }
    Iterator<T> end() { return Iterator<T>(nullptr); }
};
```

  - **clear(): 원소 전체 삭제**

```
    void clear() {
        Node<T>* cur = head;
        while (cur != nullptr) {
            Node<T>* temp = cur;
            cur = cur->next;
            delete temp;
        }
        head = nullptr;
    }
```

  - **find(): 원소 검색**

```
    Node<T>* find(const T& data) {
        Node<T>* ptr = head;
        while (ptr != nullptr) {
            if (ptr->data == data) { return ptr; }
            ptr = ptr->next;
        }
        return nullptr;
    }
```

  - **insert(): 원소 추가**

```
    void insert(const T& data) {
        Node<T>* prev = nullptr;
        Node<T>* ptr = head;
        while (ptr != nullptr) {
            if (ptr->data == data) { return; }
            prev = ptr;
            ptr = ptr->next;
        }
        Node<T>* node = new Node<T>{ data, nullptr };
        if (prev != nullptr) { prev->next = node; }
        else { head = node; }
    }
```

  - **remove(): 원소 삭제**

```
    void remove(const T& data) {
        Node<T>* prev = nullptr;
        Node<T>* ptr = head;
        while (ptr != nullptr) {
            if (ptr->data == data) { break; }
            prev = ptr;
            ptr = ptr->next;
        }
        if (ptr == nullptr) { return; }

        if (ptr->next == nullptr) {
            if (prev == nullptr) { head = nullptr; }
            else { prev->next = nullptr; }
        }
        else {
            if (prev == nullptr) { head = ptr->next; }
            else { prev->next = ptr->next; }
        }
        delete ptr;
    }
```

Linked List – Recursive

- **Linked List**
  - head 노드 / 맨 뒤에 원소 추가

```
template<typename T>
struct LinkedListRecursive {
    Node<T>* head = nullptr;

    Node<T>* clear(Node<T>* ptr);
    Node<T>* find(Node<T>* ptr, const T& data);
    Node<T>* insert(Node<T>* ptr, const T& data);
    Node<T>* remove(Node<T>* ptr, const T& data);

    Iterator<T> begin() { return Iterator<T>(head); }
    Iterator<T> end() { return Iterator<T>(nullptr); }
};
```

  - clear(): 원소 전체 삭제

```
    Node<T>* clear(Node<T>* ptr) {
        if (ptr == nullptr) { return nullptr; }
        ptr->next = clear(ptr->next);
        delete ptr;
        return nullptr;
    }
```

  - find(): 원소 검색

```
    Node<T>* find(const T& data) {
        Node<T>* ptr = head;
        while (ptr != nullptr) {
            if (ptr->data == data) { return ptr; }
            ptr = ptr->next;
        }
        return nullptr;
    }
```

  - insert(): 원소 추가

```
    Node<T>* insert(Node<T>* ptr, const T& data) {
        if (ptr == nullptr) {
            Node<T>* node = new Node<T>{ data, nullptr };
            return node;
        }
        if (ptr->data == data) { return ptr; }
        ptr->next = insert(ptr->next, data);
        return ptr;
    }
```

  - remove(): 원소 삭제

```
    Node<T>* remove(Node<T>* ptr, const T& data) {
        if (ptr == nullptr) { return nullptr; }
        if (ptr->data == data) {
            Node<T>* temp = ptr;
            if (ptr->next == nullptr) { ptr = nullptr; }
            else { ptr = ptr->next; }
            delete temp;
            return ptr;
        }
        ptr->next = remove(ptr->next, data);
        return ptr;
    }
```

Ordered Linked List – Iterative

- **Linked List**
  - head 노드 / 우선순위 낮은 원소를 뒤에 추가 (내림차순)

```
template<typename T>
struct LinkedListIterative{
    Node<T>* head = nullptr;

    void clear();
    Node<T>* find(const T& data);
    void insert(const T& data);
    void remove(const T& data);

    Iterator<T> begin() { return Iterator<T>(head); }
    Iterator<T> end() { return Iterator<T>(nullptr); }
};
```

  - clear(): 원소 전체 삭제

```
    void clear() {
        Node<T>* cur = head;
        while (cur != nullptr) {
            Node<T>* temp = cur;
            cur = cur->next;
            delete temp;
        }
        head = nullptr;
    }
```

  - find(): 원소 검색

```
    Node<T>* find(const T& data) {
        Node<T>* ptr = head;
        while (ptr != nullptr) {
            if (ptr->data == data) { return ptr; }
            if (data < ptr->data) { ptr = ptr->next; }
            else { break; }
        }
        return nullptr;
    }
```

  - insert(): 원소 추가

```
    void insert(const T& data) {
        Node<T>* prev = nullptr;
        Node<T>* ptr = head;
        while (ptr != nullptr) {
            if (ptr->data == data) { return; }
            if (data < ptr->data) { prev = ptr; ptr = ptr->next; }
            else { break; }
        }
        Node<T>* node = new Node<T>{ data, nullptr };
        if (prev != nullptr) { prev->next = node; node->next = ptr; }
        else { node->next = head; head = node; }
    }
```

  - remove(): 원소 삭제

```
    void remove(const T& data) {
        Node<T>* prev = nullptr;
        Node<T>* ptr = head;
        while (ptr != nullptr) {
            if (ptr->data == data) { break; }
            if (data < ptr->data) { prev = ptr; ptr = ptr->next; }
            else { break; }
        }
        if (ptr == nullptr) { return; }
        if (ptr->data != data) { return; }

        if (ptr->next == nullptr) {
            if (prev == nullptr) { head = nullptr; }
            else { prev->next = nullptr; }
        }
        else {
            if (prev == nullptr) { head = ptr->next; }
            else { prev->next = ptr->next; }
        }
        delete ptr;
    }
```

Ordered Linked List – Recursive

- **Linked List**
  - head 노드 / 우선순위 낮은 원소를 뒤에 추가 (내림차순)

```
template<typename T>
struct LinkedListIterative{
    Node<T>* head = nullptr;

    void clear();
    Node<T>* find(const T& data);
    void insert(const T& data);
    void remove(const T& data);

    Iterator<T> begin() { return Iterator<T>(head); }
    Iterator<T> end() { return Iterator<T>(nullptr); }
};
```

  - clear(): 원소 전체 삭제

```
    Node<T>* clear(Node<T>* ptr) {
        if (ptr == nullptr) { return nullptr; }
        ptr->next = clear(ptr->next);
        delete ptr;
        return nullptr;
    }
```

  - find(): 원소 검색

```
    Node<T>* find(Node<T>* ptr, const T& data) {
        if (ptr == nullptr) { return nullptr; }
        if (ptr->data == data) { return ptr; }
        if (data < ptr->data) { return find(ptr->next, data); }
        return nullptr;
    }
```

  - insert(): 원소 추가

```
    Node<T>* insert(Node<T>* ptr, const T& data) {
        if (ptr == nullptr) {
            Node<T>* node = new Node<T>{ data, nullptr };
            return node;
        }
        if (ptr->data == data) { return ptr; }
        if (data < ptr->data) { ptr->next = insert(ptr->next, data); }
        else if (ptr->data < data) {
            Node<T>* node = new Node<T>{ data, nullptr };
            node->next = ptr;
            ptr = node;
        }
        return ptr;
    }
```

  - remove(): 원소 삭제

```
    Node<T>* remove(Node<T>* ptr, const T& data) {
        if (ptr == nullptr) { return nullptr; }
        if (ptr->data == data) {
            Node<T>* temp = ptr;
            if (ptr->next == nullptr) { ptr = nullptr; }
            else { ptr = ptr->next; }
            delete temp;
            return ptr;
        }
        if (data < ptr->data) { ptr->next = remove(ptr->next, data); }
        return ptr;
    }
```