**자료구조 과제3**

20202863 신지환

**R-3.7** Give an algorithm for finding the penultimate (second to last) node in a singly linked list where the last element is indicated by a null next link.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

위 캡쳐본은 뒤에서 두번째 노드를 찾는 알고리즘을 작성한 소스코드이다. SLinkedList.cpp파일에서 SNode<E>\* findPenultimate() const를 선언하고 if(head == NULL || head->next == NULL) 즉, 뒤에서 두번째 노드가 존재하지 않을때는 NULL을 반환하도록하고 마지막노드의 next값이 null일떄까지 while문을 순회하면서 찾고 해당 노드를 반환한다.

텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명스크린샷이(가) 표시된 사진

자동 생성된 설명

**해당 알고리즘이 잘 작동하게 하는지 확인하기 위한 Main()함수이다. 그 결과 1,2,3순서로 addFront()를 작동시키고 확인해보니 잘 작동하는 것을 알 수 있다.**

**R-3.8** Give a fully generic implementation of the doubly linked list data structure of Section 3.3.3 by using a templated class.

#include <iostream>

#include <string>

using namespace std;

template <typename T>

class DLinkedList;

template <typename T>

class DNode {

private:

T elem;

DNode<T>\* prev;

DNode<T>\* next;

friend class DLinkedList<T>;

};

template <typename T>

class DLinkedList {

public:

DLinkedList();

~DLinkedList();

bool empty() const;

const T& front() const;

const T& back() const;

void addFront(const T& e);

void addBack(const T& e);

void removeFront();

void removeBack();

void display() const;

private:

DNode<T>\* header;

DNode<T>\* trailer;

protected:

void add(DNode<T>\* v, const T& e);

void remove(DNode<T>\* v);

};

// 생성자

template <typename T>

DLinkedList<T>::DLinkedList() {

header = new DNode<T>;

trailer = new DNode<T>;

header->next = trailer;

trailer->prev = header;

}

// 소멸자

template <typename T>

DLinkedList<T>::~DLinkedList() {

while (!empty()) removeFront();

delete header;

delete trailer;

}

// 비었는지 확인

template <typename T>

bool DLinkedList<T>::empty() const {

return (header->next == trailer);

}

template <typename T>

const T& DLinkedList<T>::front() const {

return header->next->elem;

}

template <typename T>

const T& DLinkedList<T>::back() const {

return trailer->prev->elem;

}

template <typename T>

void DLinkedList<T>::add(DNode<T>\* v, const T& e) {

DNode<T>\* u = new DNode<T>;

u->elem = e;

u->next = v;

u->prev = v->prev;

v->prev->next = u;

v->prev = u;

}

template <typename T>

void DLinkedList<T>::addFront(const T& e) {

add(header->next, e);

}

template <typename T>

void DLinkedList<T>::addBack(const T& e) {

add(trailer, e);

}

template <typename T>

void DLinkedList<T>::remove(DNode<T>\* v) {

DNode<T>\* u = v->prev;

DNode<T>\* w = v->next;

u->next = w;

w->prev = u;

delete v;

}

template <typename T>

void DLinkedList<T>::removeFront() {

remove(header->next);

}

template <typename T>

void DLinkedList<T>::removeBack() {

remove(trailer->prev);

}

template <typename T>

void DLinkedList<T>::display() const {

DNode<T>\* v = header->next;

cout << "Output: ";

while (v != trailer) {

cout << v->elem << " ";

v = v->next;

}

cout << endl;

}

// 리스트를 역순으로 만드는 함수

template <typename T>

void listReverse(DLinkedList<T>& L) {

DLinkedList<T> Tlist;

while (!L.empty()) {

T elem = L.front();

L.removeFront();

Tlist.addFront(elem);

}

while (!Tlist.empty()) {

T elem = Tlist.front();

Tlist.removeFront();

L.addBack(elem);

}

}

// main 함수

int main() {

DLinkedList<string> a;

a.addFront("Korea");

a.display();

a.addBack("Japan");

a.display();

a.addBack("China");

a.display();

a.addFront("UK");

a.display();

a.addFront("France");

a.display();

a.addFront("Spain");

a.display();

a.removeBack();

a.display();

a.removeFront();

a.display();

listReverse(a);

a.display();

DLinkedList<int> b;

b.addFront(1);

b.addBack(2);

b.addBack(3);

b.display();

return 0;

}

기존 typedef string Elem;으로 선언해 string형의 데이터만 처리했던 것을 template <typename T>를 각 함수에 선언하고 해당 제네릭 데이터타입에 맞게 리팩토링해주었다. 이를 Main()함수를 통해서 확인해보면 string타입으로 선언해 사용할 수도, int형타입으로 선언해 doubly linkedlist를 사용할 수 도있게 해주었다.

**R-3.10** Describe a nonrecursive function for finding, by link hopping, the middle node of a doubly linked list with header and trailer sentinels. (Note: This function must only use link hopping; it cannot use a counter.) What is the running time of this function?

#include <iostream>

#include <string>

using namespace std;

template <typename T>

class DLinkedList;

template <typename T>

class DNode {

public:

T elem;

DNode<T>\* prev;

DNode<T>\* next;

friend class DLinkedList<T>;

};

template <typename T>

class DLinkedList {

public:

DLinkedList();

~DLinkedList();

bool empty() const;

const T& front() const;

const T& back() const;

void addFront(const T& e);

void addBack(const T& e);

void removeFront();

void removeBack();

void display() const;

DNode<T>\* findCenter(); // 중앙 노드 찾기

private:

DNode<T>\* header;

DNode<T>\* trailer;

protected:

void add(DNode<T>\* v, const T& e);

void remove(DNode<T>\* v);

};

// 생성자

template <typename T>

DLinkedList<T>::DLinkedList() {

header = new DNode<T>;

trailer = new DNode<T>;

header->next = trailer;

trailer->prev = header;

}

// 소멸자

template <typename T>

DLinkedList<T>::~DLinkedList() {

while (!empty()) removeFront();

delete header;

delete trailer;

}

// 리스트 비었는지 확인

template <typename T>

bool DLinkedList<T>::empty() const {

return (header->next == trailer);

}

// 첫 번째 요소

template <typename T>

const T& DLinkedList<T>::front() const {

return header->next->elem;

}

// 마지막 요소

template <typename T>

const T& DLinkedList<T>::back() const {

return trailer->prev->elem;

}

// 노드 추가 (기본 유틸)

template <typename T>

void DLinkedList<T>::add(DNode<T>\* v, const T& e) {

DNode<T>\* u = new DNode<T>;

u->elem = e;

u->next = v;

u->prev = v->prev;

v->prev->next = u;

v->prev = u;

}

// 앞에 추가

template <typename T>

void DLinkedList<T>::addFront(const T& e) {

add(header->next, e);

}

// 뒤에 추가

template <typename T>

void DLinkedList<T>::addBack(const T& e) {

add(trailer, e);

}

// 노드 삭제 (기본 유틸)

template <typename T>

void DLinkedList<T>::remove(DNode<T>\* v) {

DNode<T>\* u = v->prev;

DNode<T>\* w = v->next;

u->next = w;

w->prev = u;

delete v;

}

// 앞 제거

template <typename T>

void DLinkedList<T>::removeFront() {

remove(header->next);

}

// 뒤 제거

template <typename T>

void DLinkedList<T>::removeBack() {

remove(trailer->prev);

}

// 출력

template <typename T>

void DLinkedList<T>::display() const {

DNode<T>\* v = header->next;

cout << "Output: ";

while (v != trailer) {

cout << v->elem << " ";

v = v->next;

}

cout << endl;

}

// 중앙 노드 찾기 (링크 따라가며)

template <typename T>

DNode<T>\* DLinkedList<T>::findCenter() {

DNode<T>\* forward = header->next;

DNode<T>\* backward = trailer->prev;

while (forward != backward && forward->next != backward) {

forward = forward->next;

backward = backward->prev;

}

return forward;

}

// 메인 함수

int main() {

DLinkedList<string> list1;

list1.addBack("A");

list1.addBack("B");

list1.addBack("C");

list1.addBack("D");

list1.addBack("E");

list1.display();

DNode<string>\* center1 = list1.findCenter();

cout << "[홀수] 중앙 노드: " << center1->elem << endl;

DLinkedList<string> list2;

list2.addBack("1");

list2.addBack("2");

list2.addBack("3");

list2.addBack("4");

list2.display();

DNode<string>\* center2 = list2.findCenter();

cout << "[짝수] 중앙 노드: " << center2->elem << endl;

return 0;

}

header와 trailer 센티넬 노드가 포함된 이중 연결 리스트에서 재귀나 카운터 없이 링크만을 따라가며 중앙 노드를 찾는 비재귀적 알고리즘을 구현하기 위해 findCenter()를 구현했고 추가로 header와 trailer에 쉽게 접근하기 위해 public으로 접근지정자를 변경했다. 함수는 리스트의 앞쪽에서 시작하는 forward 포인터와 뒤쪽에서 시작하는 backward 포인터를 사용한다. forward는 header->next에서 시작하고, backward는 trailer->prev에서 시작한다. 두 포인터는 각각 next와 prev 방향으로 한 칸씩 이동하며, 서로 만날 때까지 반복한다. 포인터가 같은 노드를 가리키거나, 서로 교차하기 직전에 반복을 종료하며, 그때의 forward가 리스트의 중앙 노드를 가리키게 된다. 이 과정은 재귀 호출을 사용하지 않고, 노드 간의 링크만을 이용해 리스트를 순회한다는 점에서 문제의 조건을 충족한다. 또한, 외부 변수를 통한 카운팅을 하지 않으며 오직 포인터 이동만으로 가운데 노드를 찾는다.

리스트의 길이를 n이라고 할 때, 두 포인터는 각기 O(n/2) 거리만큼 이동하므로 전체 수행 시간은 O(n)에 해당한다. 이는 리스트 전체를 한 번 순회하는 것과 동일한 시간 복잡도를 갖는다.

따라서 해당 알고리즘은 비재귀적이며, 링크 순회를 통해 중앙 노드를 정확하게 찾는 효율적인 방법이라 할 수 있다.

**R-3.15** Give a fully generic implementation of the circularly linked list data structure of Section 3.4.1 by using a templated class.

#include <iostream>

using namespace std;

template <typename T>

class CircleList; // forward declaration

template <typename T>

class CNode { // circularly linked list node

private:

T elem; // linked list element value

CNode<T>\* next; // next item in the list

friend class CircleList<T>; // provide CircleList access

};

template <typename T>

class CircleList { // a circularly linked list

public:

CircleList(); // constructor

~CircleList(); // destructor

bool empty() const; // is list empty?

const T& front() const; // element at cursor

const T& back() const; // element following cursor

void advance(); // advance cursor

void add(const T& e); // add after cursor

void remove(); // remove node after cursor

void display() const;

private:

CNode<T>\* cursor; // the cursor

};

template <typename T>

CircleList<T>::CircleList() // constructor

: cursor(NULL) { }

template <typename T>

CircleList<T>::~CircleList() // destructor

{ while (!empty()) remove(); }

template <typename T>

bool CircleList<T>::empty() const // is list empty?

{ return cursor == NULL; }

template <typename T>

const T& CircleList<T>::back() const // element at cursor

{ return cursor->elem; }

template <typename T>

const T& CircleList<T>::front() const // element following cursor

{ return cursor->next->elem; }

template <typename T>

void CircleList<T>::advance() // advance cursor

{ cursor = cursor->next; }

template <typename T>

void CircleList<T>::add(const T& e) { // add after cursor

CNode<T>\* v = new CNode<T>; // create a new node

v->elem = e;

if (cursor == NULL) { // list is empty?

v->next = v; // v points to itself

cursor = v; // cursor points to v

}

else { // list is nonempty?

v->next = cursor->next; // link in v after cursor

cursor->next = v;

// cursor = cursor->next; // add a statement \*\*\* important

}

}

template <typename T>

void CircleList<T>::remove() { // remove node after cursor

CNode<T>\* old = cursor->next; // the node being removed

if (old == cursor) // removing the only node?

cursor = NULL; // list is now empty

else

cursor->next = old->next; // link out the old node

delete old; // delete the old node

}

template <typename T>

void CircleList<T>::display() const {

if (cursor == NULL) {

cout << "Output : []" << endl;

return;

}

CNode<T>\* v = cursor->next;

cout << "Output : [";

while ( v != NULL ) {

cout << v->elem;

if (v == cursor) break;

cout << ", ";

v = v->next;

}

cout << "\*]" << endl;

}

// Example usage

int main() {

CircleList<string> playList; // []

playList.add("Stayin Alive"); //[Stayin Alive\*]

playList.display();

playList.add("Le Freak"); //[Le Freak, Stayin Alive\*]

playList.display();

playList.add("Jive Talkin"); //[Jive Talkin, Le Freak, Stayin Alive\*]

playList.display();

playList.advance(); //[Le Freak, Stayin Alive, Jive Talkin\*]

playList.display();

playList.advance(); //[Stayin Alive, Jive Talkin, Le Freak\*]

playList.display();

playList.remove(); //[Jive Talkin, Le Freak\*]

playList.display();

playList.add("Disco Inferno"); //[Disco Inferno, Jive Talkin, Le Freak\*]

playList.display();

CircleList<string> a;

a.add("Korea");

a.display();

a.add("Japan");

a.display();

a.add("USA");

a.display();

a.add("Austrailia");

a.display();

a.add("German");

a.display();

a.add("Norway");

a.display();

a.advance();

a.display();

a.advance();

a.display();

a.remove();

a.display();

a.remove();

a.display();

return EXIT\_SUCCESS;

}

template <typename T>를 사용해 기존 로직은 그대로 유지하면서 템플릯 기반으로 일반화 시켰고 string형태의 테스트 코드에서 잘 동작하는 것을 확인할 수 있다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**C-3.2** Give C++ code for performing add(e) and remove(i) functions for game entries stored in an array a, as in class Scores in Section 3.1.1, except this time, don’t maintain the game entries in order. Assume that we still need to keep n entries stored in indices 0 to n−1. Try to implement the add and remove functions without using any loops, so that the number of steps they perform does not depend on n.

#include <iostream>

using namespace std;

class IndexOutOfBounds { // IndexOutOfBounds exception

public:

IndexOutOfBounds(const string& err) // constructor

: errMsg(err) { }

string getError() { return errMsg; } // access error message

private:

string errMsg; // error message

};

class GameEntry { // a game score entry

public:

GameEntry(const string& n="", int s=0); // constructor

string getName() const; // get player name

int getScore() const; // get score

private:

string name; // player's name

int score; // player's score

};

GameEntry::GameEntry(const string& n, int s) // constructor

: name(n), score(s) { }

// accessors

string GameEntry::getName() const { return name; }

int GameEntry::getScore() const { return score; }

class Scores { // stores game high scores

public:

Scores(int maxEnt = 10); // constructor

~Scores(); // destructor

void add(const GameEntry& e); // add a game entry

GameEntry remove(int i); // remove the ith entry

void display() const;

private:

int maxEntries; // maximum number of entries

int numEntries; // actual number of entries

GameEntry\* entries; // array of game entries

};

Scores::Scores(int maxEnt) { // constructor

maxEntries = maxEnt; // save the max size

entries = new GameEntry[maxEntries]; // allocate array storage

numEntries = 0; // initially no elements

}

Scores::~Scores() { // destructor

delete[] entries;

}

void Scores::add(const GameEntry& e) {

if (numEntries == maxEntries) return; // 더 이상 공간이 없으면 무시

entries[numEntries++] = e; // 마지막 자리에 삽입

}

GameEntry Scores::remove(int i) {

if (i < 0 || i >= numEntries)

throw IndexOutOfBounds("Invalid index");

GameEntry removed = entries[i];

entries[i] = entries[numEntries - 1]; // 마지막 항목을 삭제된 자리로 복사

numEntries--; // 마지막 항목 제거

return removed;

}

void Scores::display() const {

cout << "Game Entry : " ;

for ( int i = 0; i < numEntries; i++) {

cout << "(" << entries[i].getName() << "," << entries[i].getScore() << ")";

}

cout << endl;

}

int main() {

Scores s;

s.add(GameEntry("Anna", 660));

s.display();

s.add(GameEntry("Jack", 510));

s.display();

s.add(GameEntry("Mike", 1105));

s.display();

s.add(GameEntry("Paul", 720));

s.display();

s.add(GameEntry("Rob", 750));

s.display();

s.add(GameEntry("Rose", 590));

s.display();

// Add new object (Jill, 740) to the entries array

s.add(GameEntry("Jill", 740));

s.display();

// Removal of the entry at index 3 (Anna, 660)

try {

s.remove(3);

s.display();

s.remove(7); // index invalid exception

s.display();

} catch (IndexOutOfBounds &e) {

cerr << "Exception caught : " << e.getError() << endl;

} catch (...) {

cerr << "Unknown exception error " << endl;

}

return EXIT\_SUCCESS;

}

수정한 소스코드

void Scores::add(const GameEntry& e) {

if (numEntries == maxEntries) return; // 더 이상 공간이 없으면 무시

entries[numEntries++] = e; // 마지막 자리에 삽입

}

GameEntry Scores::remove(int i) {

if (i < 0 || i >= numEntries)

throw IndexOutOfBounds("Invalid index");

GameEntry removed = entries[i];

entries[i] = entries[numEntries - 1]; // 마지막 항목을 삭제된 자리로 복사

numEntries--; // 마지막 항목 제거

return removed;

}

기존의 Scores 클래스에서는 정렬된 순서를 유지하기 위해 add 함수에서 삽입 위치를 찾기 위해 루프를 돌고, remove 함수에서는 삭제 후 요소들을 앞으로 당기는 과정을 수행하는데, 이러한 구현은 입력 크기 n에 따라 시간이 길어지므로 조건을 만족하지 않는다.

따라서 수정된 add 함수는 새 게임 점수를 단순히 배열의 마지막 위치에 추가하고, remove(i) 함수는 배열의 마지막 요소를 제거할 인덱스 i에 덮어쓰는 방식으로 구현하였다. 이 방식은 점수들의 순서를 유지하지 않는 대신 루프 없이 단 한 번의 대입 연산으로 삭제가 이루어지기 때문에 항상 일정한 시간 안에 처리가 가능하다. 이렇게 구현하면 add와 remove 모두 O(1)의 시간 복잡도를 가진다.

**C-3.12** In the **Towers of Hanoi**puzzle, we are given a platform with three pegs, a, b, and c, sticking out of it. On peg a is a stack of n disks, each larger than the next, so that the smallest is on the top and the largest is on the bottom. The puzzle is to move all the disks from peg a to peg c, moving one disk at a time, so that we never place a larger disk on top of a smaller one. Describe a recursive algorithm for solving the Towers of Hanoi puzzle for arbitrary n.

(Hint: Consider first the subproblem of moving all but the nth disk from peg a to another peg using the third as “temporary storage.”)

#include <iostream>

using namespace std;

void hanoi(int n, char from, char temp, char to) {

if (n == 1) {

cout << "Move disk 1 from " << from << " to " << to << endl;

return;

}

hanoi(n - 1, from, to, temp);

cout << "Move disk " << n << " from " << from << " to " << to << endl;

hanoi(n - 1, temp, from, to);

}

int main() {

int n = 3;

hanoi(n, 'A', 'B', 'C');

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

이 문제를 해결하기 위해 재귀적인 방법을 사용할 수 있다. 재귀 알고리즘의 기본 아이디어는 n개의 원판을 한 번에 모두 옮기는 것이 아니라, 문제를 더 작은 부분 문제로 나누는 것이다. 먼저, 가장 아래의 가장 큰 원판(n번째 원판)을 제외한 n-1개의 원판을 시작 기둥(from)에서 보조 기둥(temp)으로 옮긴다. 그런 다음, 가장 큰 원판 하나를 목적지 기둥(to)으로 직접 이동시킨다. 그 후, 보조 기둥에 옮겨놓은 n-1개의 원판을 다시 목적지 기둥으로 옮긴다. 이 과정을 반복하면 결국 모든 원판이 목적지 기둥으로 옮겨지게 된다.

이 알고리즘의 시간 복잡도는 O(2^n)으로, 원판의 수가 증가할수록 필요한 이동 횟수는 기하급수적으로 늘어난다.