**REPORT**

**[Prac 4]**



**과 목 : 알고리즘06**

**담당교수 : 주종화 교수님**

**학 과 : 컴퓨터공학과**

**학 번 : 2021111971**

**이 름 : 이재혁**

텍스트, 클립아트이(가) 표시된 사진

자동 생성된 설명

**프로그램 구현 환경**

프로그래밍언어 : C++

텍스트 에디터 : Visual Studio Code

운영체제 : MAC OS

**문제 1 : 최대 최소 값 찾기**

**프로그램 설계**

최대 최소를 찾는 Find 클래스를 정의해 Minimum, Maximum, FindMinMax 함수 메소드로 멤버 변수 배열의 최대 최소 값을 출력한다.

**전역 변수**

**num** : index에 해당하는 난수의 발생여부를 저장하는 배열

발생한 난수는 해당 index에 1의 값 저장

* 난수의 중복 방지를 위해 사용, 배열의 길이가 1000으로 크기 때문에 난수 발생 시 매번 배열을 확인하는 것은 시간이 걸린다. 추가적인 메모리를 사용해 배열설정 시간을 줄일 수 있다.

**멤버 변수**

**arr** : 랜덤 한 변수를 저장할 1000크기의 배열

**멤버 메소드**

**생성자** : 임의의 난수 1000개를 생성해 배열에 저장한다.

**Minimum()** : 현재 배열이 저장하고 있는 값 중 가장 작은 값

출력

**Maximum()** : 현재 배열이 저장하고 있는 값 중 가장 큰 값

출력

**FindMinMax ()** : 현재 배열이 저장하고 있는 값 중 가장 작은 값과 큰 값 출력

* 함수들은 모두 배열을 처음부터 끝까지 확인하며 최대, 최소 값을 저장하는 변수들과 비교한다. 값을 확인했을 때 해당 변수보다 클 때는 최댓값을 작을 때는 최솟값을 수정한다.
* FindMinMax 함수에서 최댓값보다 큰 변수를 확인했을 시 최솟값 보다 작을 수 없고 반대 경우도 마찬가지로 비교횟수를 줄일 수 있다.

**print()** : 현재 배열의 0, 999 index의 값을 출력하는 메소드

**merge(int left, int mid, int right)**

**re\_MergeSort(int left, int right)**

* **합병 정렬을 시행하는 메소드, 최솟값 최댓값을 찾았을 때 실제로 맞는지 확인하기 위해 작성**

**(실습3 의 메소드를 그대로 가져와 사용했습니다.)**

**소스코드**

// 2021111971 이재혁 문제1

#include <cstdlib> // 난수 생성을 위한 라이브러리

#include <ctime> // 난수 생성을 위한 라이브러리

#include <iostream>

using namespace std;

int num[100001] = {0};

// 난수의 발생여부를 저장할 배열

// index에 해당하는 난수가 발생했다면 1, 발생하지 않았다면 0의 값 저장

class Find {

public:

Find() { // 1000개의 고유한 난수를 생성하여 배열에 저장

int i = 0;

while (i < 1000) {

int n = rand() % 100000 + 1;

// 1 ~ 100000 사이의 난수 생성

if (num[n] == 0) { // 처음 발생한 난수라면

arr[i] = n; // 배열에 추가

num[n] = 1; // 난수가 생성됐음을 표시

i++; // 다음에 추가할 위치로 이동

}

}

}

void Minimum() { // 배열에서 최솟값을 찾아 출력하는 메소드

cout << "----- Minimum() -----" << endl;

int min = arr[0]; // 최솟값을 첫 번째 배열 값으로 초기화

for (int i = 1; i < 1000; i++) {

if (arr[i] < min) { // 더 작은 값이 발견되면 최솟값 갱신

min = arr[i];

}

}

cout << "배열의 최솟값 : " << min << endl;

}

void Maximun() { // 배열에서 최댓값을 찾아 출력하는 메소드

cout << "----- Maximum() -----" << endl;

int max = arr[0]; // 최댓값을 첫 번째 배열 값으로 초기화

for (int i = 1; i < 1000; i++) {

if (arr[i] > max) { // 더 큰 값이 발견되면 최댓값 갱신

max = arr[i];

}

}

cout << "배열의 최댓값 : " << max << endl;

}

void FindMinMax() { // 배열에서 최솟값과 최댓값을 모두 찾아 출력하는 메소드

cout << "----- FindMinMax() -----" << endl;

int max = arr[0]; // 최댓값을 첫 번째 배열 값으로 초기화

int min = arr[0]; // 최솟값을 첫 번째 배열 값으로 초기화

for (int i = 1; i < 1000; i++) {

if (arr[i] > max) { // 더 큰 값이 발견되면 최댓값 갱신

max = arr[i];

} else if (arr[i] < min) { // 더 작은 값이 발견되면 최솟값 갱신

min = arr[i];

}

// 최댓값을 찾았다면 그 값은 최솟값이 될 수 없으므로

// else if문을 사용해 실행시간 최적화

}

cout << "배열의 최솟값 : " << min << endl;

cout << "배열의 최댓값 : " << max << endl;

}

void print() {

cout << "index 0 : " << arr[0] << endl;

cout << "index 999 : " << arr[999] << endl;

}

void merge(int left, int mid, int right) {

int \*list;

list = new int[right - left + 1];

int start1 = left;

int start2 = mid + 1;

int index = 0;

while (start1 <= mid && start2 <= right) {

if (arr[start1] < arr[start2]) {

list[index++] = arr[start1++];

} else {

list[index++] = arr[start2++];

}

}

if (start1 > mid) {

for (int i = start2; i <= right; i++) {

list[index++] = arr[i];

}

} else {

for (int i = start1; i <= mid; i++) {

list[index++] = arr[i];

}

}

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

arr[left + i] = list[i];

}

delete[] list;

}

void re\_MergeSort(int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

this->re\_MergeSort(left, mid);

this->re\_MergeSort(mid + 1, right);

this->merge(left, mid, right);

}

}

private:

int arr[1000];

};

int main() {

srand((unsigned int)time(NULL));

Find f;

f.Minimum();

f.Maximun();

f.FindMinMax();

cout << "정렬 후 0, 999 번 index 확인" << endl;

f.re\_MergeSort(0, 999);

f.print();

}

**결과 분석**

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Minimum, Maximum 메소드와 FindMinMax 메소드가 같은 값을 찾은 것을 확인할 수 있다.

정렬 후 0번 index -> 최솟값

999번 index -> 최댓값

또한 같은 값인 것을 확인할 수 있다.

* 올바른 최솟값, 최댓값을 찾았다.

**문제 2 : 다단계 합병 정렬, 단계별 블록개수 계산**

마지막 단계는 런이 1개만 존재해야 한다.

역 단계 1은 1개의 빈 테이블을 제외한 모든 테이블의 런 개수가 1이다.

역 단계 1에서 T0 이 0 이므로 역 단계 2에서 T0에 있던 모든 런을 합친 것을 알 수 있다.

비어 있던 T1에 1개의 런이 들어갔기 때문에,

역 단계 1에서T0에는 1개의 런만 존재 했던 것을 알 수 있다.

T2, T3에서도 1개의 런을 꺼내 합쳤기 때문에, 역 단계 1에서 1개씩 남았다면, 합치기전인 역 단계2 에서 각각 2개의 런이 있었던 것을 알 수 있다.

|  |  |
| --- | --- |
| **텍스트, 친필, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명** | 2->1  T0에 있는 모든 런(1개)을 꺼내어 결합  -> 0개가 됨  비어 있던 T1에 저장  -> 1개가 됨  2개의 런이 들어있던 T2, T3에서 1개의 런을 꺼내어 결합  -> 각각 1개의 런이 남음 |
| **텍스트, 친필, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명** | 3->2  T1에 있는 모든 런(2개)을 꺼내어 결합  -> 0개가 됨  비어 있던 T2에 저장  -> 2개가 됨  3개의 런이 들어있던 T0에서 2개의 런을 꺼내어 합침  -> 1개가 됨  4개의 런이 들어있던 T3에서 2개의 런을 꺼내어 결합  -> 각각 2개의 런이 남음 |
| 위와 같은 방식으로 계산  **텍스트, 친필, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명** | |
| **텍스트, 친필, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명** | |
| **텍스트, 친필, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명** | |
| **텍스트, 친필, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명** | |

**문제 3 : 알고리즘의 수행시간 계산**

|  |  |
| --- | --- |
| **텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진  자동 생성된 설명** | 텍스트, 친필, 폰트, 서예이(가) 표시된 사진  자동 생성된 설명텍스트, 친필, 폰트, 서예이(가) 표시된 사진  자동 생성된 설명 |

**문제 4 : BST**

|  |  |
| --- | --- |
| **BST 생성과정** | **Delete 6** |
| **텍스트, 스크린샷, 도표, 원이(가) 표시된 사진  자동 생성된 설명**  **패턴이(가) 표시된 사진  자동 생성된 설명** | **텍스트, 도표, 라인, 패턴이(가) 표시된 사진  자동 생성된 설명그림, 스케치, 패턴이(가) 표시된 사진  자동 생성된 설명** |

**Delete 연산**

* 삭제하려는 노드의 자식 노드 개수에 따라 delete연산의 방식이 다르다.

Case 1) 자식 노드가 없는 경우 -> 현재 노드 반환

Case 2) 자식 노드가 한쪽만 있는 경우

* 현재 노드의 부모 노드가, 그 자식 노드를 가리키고 현재 노드는 반환한다.

Case 3) 자식 노드가 양쪽 다 있는 경우

* 현재 노드의 값을 오른쪽 자식 노드 중 가장 작은 값과 교환 한다.

교환된 위치는 가장 작은 값의 위치였기 때문에 왼쪽 노드가 존재하지 않는다. Case2의 delete 연산을 수행한다.

**프로그램 설계**

트리의 노드의 성질을 갖는 class를 정의한다.

노드의 추가, 삭제 메소드를 정의해 class 객체들의 연결을 관리한다.

**멤버변수**

**value** : 노드가 가지는 값

**left** : 노드의 왼쪽 자식 노드를 가리키는 포인터, 현재 노드의

value보다 작은 값을 가진다.

**right** : 노드의 오른쪽 자식 노드를 가리키는 포인터,

현재 노드의value보다 작은 값을 가진다.

**멤버 메소드**

**생성자** : 매개변수로 value를 전달받아 노드의 값을 설정한다.

생성한 노드는 아직 연결 되어있지 않으므로 left, right 포인터의 값이 NULL이다, insertNode함수로 다른 노드와 연결성이 생긴다.

**insertNode(TreeNode\* node)**:

새로운 노드를 트리에 삽입하는 메소드

현재 노드의 값과 전달받은 노드의 값을 비교해 적절한 위치의 노드와 연결한다.

**deleteNode(TreeNode\* node, int value)**:

전달받은 노드부터, value의 값을 찾아 삭제하는 메소드

node의 자식 개수에 따라 삭제 방식이 다르다.

자식 개수가 2개일 때 큰 값들 중 가장 작은 값과

교환 후 다시 deleteNode를 시행한다.

부모 노드에 삭제 후 남아 있는 트리를 연결하기 위해

삭제 후 root노드를 반환한다.

* 순환적 방식으로 삭제할 노드를 탐색하고 반환한다.

**minNode (TreeNode\* node)** :

현재 노드에서 가장 작은 값을 반환하는 메소드

**printTree ()** :

중위 순회 (root -> left -> right 순으로 출력) 방식으로

트리 출력

**소스코드**

// 2021111971 이재혁 문제 4

#include <iostream> // 표준 입출력을 위한 헤더 파일

using namespace std;

class TreeNode {

public:

TreeNode(int v) { // 노드의 값, 자식 노드를 NULL로 설정

value = v;

left = NULL;

right = NULL;

}

// 새로운 노드를 트리에 삽입하는 함수

void insertNode(TreeNode \*node) {

// 삽입할 노드의 값이 현재 노드의 값보다 작은 경우

// 현재 노드의 왼쪽에 삽입

if (this->value > node->value) {

if (this->left) { // 왼쪽 자식이 있으면 왼쪽에서 다시 확인

this->left->insertNode(node);

} else { // 왼쪽 자식이 없으면 새로운 노드를 왼쪽에 삽입

this->left = node;

}

} else { // 삽입할 노드의 값이 현재 노드의 값보다 큰 경우

if (this->right) { // 오른쪽 자식이 있으면 오른쪽에서 다시 확인

this->right->insertNode(node);

} else { // 오른쪽 자식이 없으면 새로운 노드를 오른쪽에 삽입

this->right = node;

}

}

}

// 특정 값을 가지는 노드를 삭제하는 함수

TreeNode \*deleteNode(TreeNode \*node, int value) {

if (node == NULL) // 삭제할 노드를 찾지 못한 경우

return node;

// 삭제할 값이 현재 노드의 값보다 작은 경우, 왼쪽 서브트리에서 탐색

if (value < node->value)

node->left = deleteNode(node->left, value);

// 삭제할 값이 현재 노드의 값보다 큰 경우, 오른쪽 서브트리에서 탐색

else if (value > node->value)

node->right = deleteNode(node->right, value);

else { // 삭제할 값을 찾은 경우

// 자식이 없거나 한쪽만 있는 경우

if (node->left == NULL) {

TreeNode \*tmp = node->right; // 오른쪽 자식을 임시로 저장

free(node); // 현재 노드 메모리 해제

return tmp; // 오른쪽 자식 반환 (왼쪽이 없을 경우)

}

if (node->right == NULL) {

TreeNode \*tmp = node->left; // 왼쪽 자식을 임시로 저장

free(node); // 현재 노드 메모리 해제

return tmp; // 왼쪽 자식 반환 (오른쪽이 없을 경우)

}

// 양쪽 자식이 모두 있는 경우, 오른쪽 서브트리에서 가장 작은 값을

// 찾음

TreeNode \*tmp = minNode(node->right);

node->value = tmp->value; // 해당 값을 현재 노드로 이동

node->right =

deleteNode(node->right, tmp->value); // 해당 노드를 삭제

}

return node; // 삭제 후의 노드를 반환

}

// 주어진 트리에서 가장 작은 값을 가지는 노드를 찾는 함수

TreeNode \*minNode(TreeNode \*root) {

TreeNode \*tmp = root;

while (tmp->left != NULL) // 왼쪽 자식이 없을 때까지 탐색

tmp = tmp->left;

return tmp; // 가장 작은 값을 가지는 노드 반환

}

// 트리출력 : 중위순회

void printTree() {

cout << this->value << " ";

if (this->left) { // 왼쪽 자식이 있으면 왼쪽 서브트리 출력

this->left->printTree();

}

if (this->right) { // 오른쪽 자식이 있으면 오른쪽 서브트리 출력

this->right->printTree();

}

}

// 소멸자: 트리의 모든 노드를 재귀적으로 삭제

~TreeNode() {

if (left) // 왼쪽 자식이 있으면 삭제

delete left;

if (right) // 오른쪽 자식이 있으면 삭제

delete right;

}

private:

int value; // 현재 노드의 값

TreeNode \*left; // 왼쪽 자식 노드

TreeNode \*right; // 오른쪽 자식 노드

};

int main() {

TreeNode \*root; // 트리의 루트 노드를 가리킬 포인터

cout << "10 개의 노드 값 입력 >> ";

int input;

cin >> input;

root = new TreeNode(input); // 첫 번째 입력 값으로 루트 노드 생성

for (int i = 0; i < 9; i++) { // 9개의 추가 노드를 입력받아 트리에 삽입

cin >> input;

root->insertNode(new TreeNode(input)); // 입력받은 값을 삽입

}

root->printTree(); // 현재 트리의 노드 값 출력

cout << endl;

cout << "------- 6 삭제 ---------" << endl;

root = root->deleteNode(root, 6); // 값이 6인 노드를 삭제

root->printTree(); // 노드 삭제 후 트리 출력

cout << endl;

cout << "------- 20 삭제 ---------" << endl;

root = root->deleteNode(root, 20); // 값이 20인 노드를 삭제

root->printTree(); // 노드 삭제 후 트리 출력

cout << endl;

delete root; // 트리 메모리 해제

}

// 20 6 2 4 16 10 8 12 14 9

**결과 분석**

**텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

**초기 트리**

|  |  |
| --- | --- |
| **패턴이(가) 표시된 사진  자동 생성된 설명** | **중위 순회 (Root -> Left -> Right) 출력**  **20 -> 6(Left) -> 2 -> 4 -> 16(Right) -> 10 -> 8(Left) -> 9 -> 12(Right) -> 14** |

|  |  |
| --- | --- |
| **텍스트, 도표, 라인, 패턴이(가) 표시된 사진  자동 생성된 설명그림, 스케치, 패턴이(가) 표시된 사진  자동 생성된 설명** | **중위 순회 (Root -> Left -> Right) 출력**  **20 -> 8(Left) -> 2 -> 4 -> 16(Right) -> 10 -> 9(Left) -> 12(Right) -> 14** |