Map STL 구현

우리는 Map STL을 코딩하면서 정말 많이 사용한다.

Pair로 저장할 수 있고 ,

자동으로 정렬이 되고 ,

삽입 / 삭제가 빠르고, …

장점이 되게 많기 때문이다.

오늘은 Map STL을 C++로 구현 해보고 Map에서 사용 된 기능 , 그 구조에 대해서 알아볼 것이다. 한 마디로 Map STL을 완전 뜯어 볼 생각이다. 많은 공부가 될 것이다.

요구 되는 기능

1. RB Tree

Map STL은 RB Tree를 기반으로 만들어 졌다.

Rb Tree는 삽입 , 삭제 , 탐색 모두 O(log n) 으로 할 수 있기 때문에 아주 중요한 자료 구조이다.

Map STL을 구현 하면서 Rb Tree의 삽입 ( Insertion ), 삭제 ( Deletion ) 을 구현하여 사용하였다.

1. Template

template <typename T1, typename T2>

Vector , list STL들처럼 Map STL 에도 다양한 type들을 저장하기 위해서는 Template을 사용해야 했다. Map STL의 경우 key와 value를 가지기 때문에 template 속 타입을 두 개로 지정하여 사용하였다.

1. Iterator

STL 마다 그 STL 만의 iterator가 정의 되어 있다.

내가 구현한 Map STL에서도 나만의 iterator를 정의하여 Map의 모든 값들을 편리하게 접근하기 위해서 iterator를 정의 하고 그에 알맞게 연산자 오버로딩을 통해서 iterator가 제 기능을 하게끔 해주었다.

1. 인덱스 연산자 오버로딩

m[key] = value

맵에서 흔히 사용 되는 insert , modify 방식이다.

Map STL을 사용할 때는 당연하게 사용해 왔지만 구현을 통해 사용하려면 인덱스 연산자 오버로딩 ( operator [] )를 반드시 해주어야 이 기능을 사용할 수 있다.

구현

구조

Node Struct

// Node Structure   ---------------------------------------------------------------------------------------------------------------------

// {key, value} + color + {L,R,P}

template <typename T1, typename T2>     // Template for key, value

struct Tree\_node{

    T1 key;         // Map key

    T2 value;       // Map value

    char color;     // Red-Black Color

    Tree\_node<T1,T2>\* left;         // pointer Left

    Tree\_node<T1,T2>\* right;        // pointer Right

    Tree\_node<T1,T2>\* parent;       // pointer Parent

    // Node Constructor

    Tree\_node(){color='R';left = nullptr;right=nullptr;parent=nullptr;}

    Tree\_node(T1 k , T2 v , char c , Tree\_node\* l , Tree\_node\* r , Tree\_node\* p){key = k;value = v;color = c; left =l; right =r;parent=p;}

    // Node Destructor

    ~Tree\_node(){}

};

Map의 Rbtree의 노드 부분이다.

Key와 value를 사용하기 때문에 노드 속에 key , value 두 가지의 변수가 있어야 한다.

Rbtree는 Left child , Right child 노드 포인터 뿐만 아니라 Parent 노드 포인터도 가지고 있다.

Iterator Implementation Struct

// Iterator Class Implementation for map -------------------------------------------------------------------------------------------------

// operator overloading : ++(prefix) , \* , == , !=

template<typename T1, typename T2>

class MyIterator{

private:

    Tree\_node<T1,T2>\* cur;

public:

    // MyIterator Constructor

    MyIterator(Tree\_node<T1,T2>\*p = nullptr)

        :cur(p){}

    // operator++ overloading

    // Finding node having next min key

    MyIterator& operator++(){

        // If it has right child

        if(cur->right){

            Tree\_node<T1,T2>\* search = cur->right;

            while(search -> left) search = search ->left; // find minimum in right subtree.

            cur = search;

        }

        // No Right child

        else{

            // cur == root + No right child (no parent)

            if(!cur->parent) cur = nullptr;

            // cur has parent

            else{

                Tree\_node<T1,T2>\* search= cur->parent;

                // Checking cur is either left child or right child

                while(cur == search->right){

                    cur = search;

                    if(search->parent) search = search->parent;

                    else{

                        cur = nullptr;

                        return \*this;

                    }

                }

                cur = search;

            }

        }

        return \*this;   // return iterator

    }

    // Operator '\*' overloading for Get cur

    Tree\_node<T1,T2>\* operator\*(){

        return cur;

    }

    // Operator '==' overloading for comparation.

    bool operator==(const MyIterator &ref){

        return cur == ref.cur;

    }

    //Operator '!=' overloading for comparation.

    bool operator!=(const MyIterator &ref){

        return cur != ref.cur;

    }

};

Iterator class 안에 노드 포인터를 Map 탐색을 위해 노드 포인터를 넣어 두었다.

Iterator에 사용되는 연산자 ++ , \* , == , != 들을 오버로딩 하였다.

기본적으로 Map은 key 기준 오름차순으로 정렬이 되어 있다.

그렇기 때문에 연산자 ++ 는 현재 iterator가 가리키고 있는 노드의 그 다음으로 큰 노드를 가리키게 끔 설정 해주었다. RbTree도 (왼쪽 서브트리가 루트 로드 보다 더 작고 오른쪽 서브트리가 루트 로드보다 큰) 이진탐색트리의 성질을 가지고 있다. 그러므로 iterator가 inorder 순서로 접근할 수 있게 구현 해주면 된다.

연산자 \* , != , == 는 간단하므로 생략하겠다.

Mymap Class ( map implementation )

// MyMap Class ---------------------------------------------------------------------------------------------------------------------------

// STL Map implementation using Red - Black Tree

template <typename T1, typename T2>

class MyMap{

private:

    Tree\_node<T1,T2>\* root;     // RB Tree

    int num;                    // size of tree

public:

    MyMap(){root= new Tree\_node<T1,T2>(); num=0;};          // Constructor

    ~MyMap(){};                                             // Destructor

    // Operator '[]' overloading for index access

    // m[key] = value   (==)    m.insert(make\_pair(key,value))

    T2& operator[](T1 key){

        MyMap<T1,T2>::iterator iter;    // using iterator

        iter = find(key);

        T2 value;                       // garbage value

        // There isn't Key in map , then insert

        if(iter == end()){

            insert(make\_pair(key,value));

            iter = find(key);

        }

        return (\*iter)->value;  // return iter's value for modification.

    }

    // Map Functions

    void insert(const pair<T1,T2> & in);

    void erase(T1 out);

    void modify(Tree\_node<T1,T2>\* modf);

    void modify\_erase(Tree\_node<T1,T2>\* p, char LR);

    // Iterator for MyMap and it's Functions.

    // Iterator typedef

    typedef MyIterator<T1,T2> iterator;

    // Functions of Iterator

    iterator begin(){

        Tree\_node<T1,T2>\* search = root;

        while(search->left) search = search->left;

        return iterator(search);

    }

    iterator end(){

        return iterator(nullptr);

    }

    iterator find(T1 key){

        MyMap<string, int>::iterator iter;

        for(iter = begin(); iter!=end(); ++iter){

            if((\*iter)->key == key) return iterator(\*iter);

        }

        return end();

    }

    // Print for Checking

    // void print();                            // Level Order(BFS) using Queue

    // void print(Tree\_node<T1,T2>\* temp);      // Inorder

};

Root 노드 포인터를 가지고 있다.

클래스 내에 인덱스 연산자 [] 오버로딩을 해주었다.

m[key] = value 의 형태로 insert , modify가 가능해졌다.

클래스 내에 iterator를 typedef로 재정의 함으로써 map만의 iterator를 만들었다.

그리고 begin , end , find 등 iterator의 함수 또한 정의해 두었다.

함수

RR rotation , RL rotation , LL rotation , LR rotation

Tree의 Balance 를 맞추어 주는 함수들이다.

Rb Tree의 삽입 ( Insertion ) , 삭제 ( Deletion )에 사용된다.

Recoloring은 rotation 함수 내에 포함되지 않았다.

삽입 / 삭제의 각 케이스 마다 recoloring이 다르기 때문이다.

// << Rotation >> ------------------------------------------------------------------------------------------------------------------------

// LR\_Rotation for modify function

template<typename T1, typename T2>

void LR\_Rotation(Tree\_node<T1,T2>\* node){

    // node == parent

    // node\_r == parent->right child

    Tree\_node<T1,T2>\* node\_r = node->right;

    //grand parent change

    node->parent->left = node\_r;

    // parent->left child change

    node\_r->parent = node->parent;

    // parent change

    node-> parent = node\_r;

    // parent->left child -> right child change

    node->right = node\_r->left;

    if(node->right) node->right->parent = node;

    // parent->left child change

    node\_r->left = node;

}

// LL\_Rotation for modify function

template<typename T1 , typename T2>

void LL\_Rotation(Tree\_node<T1,T2>\* node){

    // node is parent

    //grand parent change

    node->parent->left = node ->right;

    // parent->right change

    if(node-> right) node->right->parent = node->parent;

    // parent change

    node->right = node->parent;

    node-> parent = node->parent->parent;

    if(node->parent){

        if(node->parent->right == node->right) node->parent->right = node;

        if(node->parent->left == node->right) node->parent->left = node;

    }

    //grand parent change(rest part)

    node->right->parent = node;

}

// RL\_Rotation Function for modify function

template<typename T1, typename T2>

void RL\_Rotation(Tree\_node<T1,T2>\* node){

    // node == parent

    // node\_l == parent->left child

    Tree\_node<T1,T2>\* node\_l = node->left;

    //grand parent change

    node->parent->right = node\_l;

    // parent->left child change

    node\_l->parent = node->parent;

    // parent change

    node-> parent = node\_l;

    // parent->left child -> right child change

    node->left = node\_l->right;

    if(node->left) node->left->parent = node;

    // parent->left child change

    node\_l->right = node;

}

// RR\_Rotation for modify function

template<typename T1 , typename T2>

void RR\_Rotation(Tree\_node<T1,T2>\* node){

    // node is parent

    //grand parent change

    node->parent->right = node ->left;

    // node->parent->color = 'R';

    // parent->right change

    if(node->left) node->left->parent = node->parent;

    // parent change

    // node->color = 'B';

    node->left = node->parent;

    node-> parent = node->parent->parent;

    if(node->parent){

        if(node->parent->right == node->left) node->parent->right = node;

        if(node->parent->left == node->left) node->parent->left = node;

    }

    //grand parent change(rest part)

    node->left->parent = node;

}

// ---------------------------------------------------------------------------------------------------------------------------------------------------

이제 제일 중요한 Insertion / Deletion을 알아보자

Recoloring 과 Parent 노드 포인터 연결이 올바르게 적용하지 않으면 오류가 생기니 반드시 알맞게 수정 해주어야 한다.

Insertion

Insertion의 규칙

삽입된 새로운 노드는 반드시 빨간색으로 설정

부모 노드가 검정색이면 modify 안함

Case들

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 삽입된 노드 | 부모 노드 | 삼촌 노드 | 할아버지 노드와  부모 노드 관계 | 삽입된 노드와  부모노드 관계 | modify |
| R | R | R | Left child | 상관 X | Recoloring |
| R | R | B | Left child | Left child | LL\_rotation + Recoloring |
| R | R | B | Left child | Right child | LR\_rotation + LL\_rotation  Recoloring |
| R | R | R | Right child | 상관 X | Recoloring |
| R | R | B | Right child | Right child | RR\_rotation + Recoloring |
| R | R | B | Right child | Left child | RL\_rotation + RR\_rotation + Recoloring |

Code

// << Insertion >> -----------------------------------------------------------------------------------------------------------------------------------

// Insertion Function

// It is supported by Modify Function , RR\_rotation Function , LL\_rotation Function.

// Insertion -> Modification -> Rotation

template<typename T1 , typename T2>

void MyMap<T1,T2>::insert (const pair<T1,T2> & in){

    // First Node

    if(num == 0){

        root->key = in.first; root->value = in.second;

        num++;

        modify(root);

    }

    // Else Nodes

    else{

        // Finding location + Insertion

        Tree\_node<T1,T2>\* search = root;

        while(search){

        if(search->key == in.first) return;

        else if( search->key < in.first){

            if(!search->right){ search->right = new Tree\_node<T1,T2>(in.first,in.second,'R',nullptr,nullptr,search); break;}

            search = search-> right;

        }

        else if(search->key > in.first){

            if(!search->left){ search->left = new Tree\_node<T1,T2> (in.first,in.second,'R',nullptr,nullptr,search); break;}

            search = search->left;

        }

        }

        // Num Increase

        num++;

        // Modification

        if(search->left && in.first == search->left->key) modify(search->left);

        else if(search->right && in.first == search->right->key) modify(search->right);

    }

}

// Modify Function for Insert Function

// Part of Insertion Function

template<typename T1, typename T2>

void MyMap<T1,T2>::modify(Tree\_node<T1,T2>\* modf){

    // If root -> color = 'Black'

    if(root == modf) modf->color = 'B';

    // Else cases

    else{

        Tree\_node<T1,T2>\* parent = modf->parent;    // Parent

        if(parent->color =='B') return;             // If parent is Black Node

        Tree\_node<T1,T2>\* gr\_parent = parent->parent;   // Grand Parent

        // If parent is Left child of Grand Parent

        if(parent == gr\_parent->left){

            // If Uncle is Red Node

            // Color change + recursive

            if(gr\_parent->right && gr\_parent->right->color=='R'){

                gr\_parent->right->color = 'B';

                gr\_parent->color = 'R';

                parent->color = 'B';

                modify(gr\_parent);

            }

            // If Uncle is Black Node

            else{

                // LL Case

                if(modf == parent->left){

                    if(gr\_parent == root) root = parent;

                    parent->parent->color = 'R';

                    parent->color = 'B';

                    LL\_Rotation(parent);

                    }

                // LR Case

                else if(modf == parent->right){

                    LR\_Rotation(parent);

                    if(gr\_parent == root) root = modf;

                    modf->parent->color='R';

                    modf->color='B';

                    LL\_Rotation(modf);

                }

            }

        }

        // If parent is Right child of Grand Parent

        else if(parent == gr\_parent ->right){

            // If Uncle is Red Node

            // Color change + recursive

            if(gr\_parent->left && gr\_parent->left->color =='R'){

                gr\_parent->left->color = 'B';

                gr\_parent->color = 'R';

                parent->color = 'B';

                modify(gr\_parent);

            }

            // If Uncle is Black Node

            else{

                // RR Case

                if(modf == parent->right) {

                    if(gr\_parent == root) root = parent;

                    parent->parent->color = 'R';

                    parent->color = 'B';

                    RR\_Rotation(parent);

                }

                // RL Case

                else if(modf == parent->left){

                    RL\_Rotation(parent);

                    if(gr\_parent == root) root = modf;

                    modf->parent->color = 'R';

                    modf->color = 'B';

                    RR\_Rotation(modf);

                }

            }

        }

    }

}

// ---------------------------------------------------------------------------------------------------------------------------------------

Deletion ( = Erase )

규칙 :

지우려는 노드의 색이 Red 이면 삭제만 진행

지우려는 노드의 색이 Black 이고 대체되는 노드가 NIL 이거나 Black 이면 삭제 + Recoloring

지우려는 노드의 색이 Black 이고 대체되는 노드가 NIL 이거나 Black 이면 삭제 + modify

Modify는 5개의 Case로 나누어 진행한다.

코드에 자세히 적어 놓았으니 코드를 보면 Case 5개를 알 수 있다..!

Code

// << Erase >> ---------------------------------------------------------------------------------------------------------------------------

// Erase Function

// [ modify\_erase , RR\_Rotation , LL\_Rotation ] functions are used.

// Erase -> modify -> rotate

template<typename T1 , typename T2>

void MyMap<T1,T2>::erase (T1 out){

    Tree\_node<T1,T2>\* search = root;    // pointer searching delete node (having key 'out')

    Tree\_node<T1,T2>\* replace;          // pointing replace node

    Tree\_node<T1,T2>\* del\_parent;       // parent of search pointer

    char del\_color;                     // Color of delete node

    char LR;                            // Whether the delete node is left child or right child

    while(search){

        // Finding delete node

        if(out > search->key) search = search->right;

        else if(out < search->key) search = search->left;

        // Found

        else{

        // Delete

        // Case 1 : NO Child

            if(!search->left && !search->right){

                num--;

                del\_color = search->color; // color information

                if(search==root) delete search; // if root

                else{

                    del\_parent = search->parent;    // parent of delete node

                    // LR information

                    if(del\_parent->right == search) { del\_parent->right = NULL;  LR = 'R';}

                    else if(del\_parent->left == search){ del\_parent->left = NULL;  LR = 'L';}

                    delete search;

                }

                // if delete node is Red -> Pass

                // if delete node is black -> Modify\_erase   (Because replace node is NIL -> color is black)

                if(del\_color == 'B') modify\_erase(del\_parent , LR);

                break;

            }

        // Case 2-1 : 1 Child(left)

            else if(search->left && !search->right){

                num--;

                replace = search->left;     // replace node

                del\_color = search->color;  // color information

                // if root

                if(search==root){

                    root = replace;

                    root->parent = nullptr;

                    delete search;

                }

                // Else

                else{

                    del\_parent = search->parent;    // parent of delete node

                    // LR information

                    if(del\_parent->left == search) {del\_parent->left = replace; LR = 'L';}

                    else if(del\_parent->right == search) {del\_parent-> right = replace;  LR ='R';}

                    replace->parent = search->parent;

                    delete search;

                }

                // if color of delete node is Red -> Pass

                // if color of delete node is Black

                if(del\_color == 'B'){

                    // if replace node color is Red -> change to black

                    if(replace->color =='R') replace->color = 'B';

                    // if replace node color is Black -> modify\_erase

                    else if(replace->color == 'B') modify\_erase(del\_parent , LR);

                }

                break;

            }

        // Case 2-2 : 1 Child(right)

            else if(!search->left && search->right){

                num--;

                replace = search->right;        // replace of delete node

                del\_color = search->color;      // color information

                // if root

                if(search==root){

                    root = replace;

                    root->parent = nullptr;

                    delete search;

                }

                // Else

                else{

                    del\_parent = search->parent;    // Parent of delete node

                    // LR information

                    if(del\_parent->left == search){ del\_parent->left = replace;  LR = 'L';}

                    else if(del\_parent->right == search){ del\_parent->right = replace; LR = 'R';}

                    replace->parent = search->parent;

                    delete search;

                }

                // if color of delete node is Red -> Pass

                // if color of delete node is Black

                if(del\_color == 'B'){

                    // if replace node color is Red -> change to black

                    if(replace->color =='R') replace->color ='B';

                    // if replace node color is Black -> modify\_erase

                    else if(replace ->color == 'B') modify\_erase(del\_parent , LR);

                }

                break;

            }

        // Case 3 : 2 Child (left & right)

            else{

                // replace node is maximum of left subtree

                replace = search->right;

                while(replace->left) replace = replace->left;

                T1 key\_temp = replace->key;

                T2 value\_temp = replace->value;

                erase(replace->key);        // recursive erase

                // change delete node to replace node

                search->key = key\_temp;

                search->value = value\_temp;

            }

        }

    }

}

// Isblack Function

// For modify\_erase function

// NIL or node->color =='B' -> True

template<typename T1, typename T2>

bool isblack(Tree\_node<T1,T2>\* node){

    return (!node || (node && node->color =='B') );

}

// Modify\_Erase Function

// recoloring Nodes and modifying Tree After Delete

// p : parent of delete node

// s : sibling of delete node

// s\_l : left child of sibling

// s\_r : right child of sibling

// if LR is 'L' -> There are 5 cases

// if LR is 'R' -> There are 5 cases (Symmetric to the case above)

template<typename T1 , typename T2>

void MyMap<T1,T2>::modify\_erase(Tree\_node<T1,T2>\* p, char LR){

// Delete node is left child of p

    if(LR =='L'){

// Case 1 :

    // p is Red

    // s is black

    // s\_l & s\_r is black

    // << Solution >>

    // Recoloring

        if(p->color == 'R' && (p->right && isblack(p->right)) && isblack(p->right->left) && isblack(p->right->right) ){

            p->color = 'B';

            p->right->color = 'R';

        }

// Case 2 :

    // p is black

    // s is black

    // s\_l & s\_r & black

    // << Solution >>

    // Recoloring

    // recursive

        else if(p->color =='B' && (p->right && isblack(p->right)) && isblack(p->right->left) && isblack(p->right->right) ){

            p->right->color = 'R';

            if(p->parent){

                if(p->parent->left && p->parent->left == p) modify\_erase(p->parent, 'L');

                else if(p->parent->right && p->parent->right == p) modify\_erase(p->parent, 'R');

            }

        }

// Case 3 :

    // s is black

    // s\_r is Red

    // << Solution >>

    // RR rotation + Recoloring

        else if( (p->right && isblack(p->right)) && (p->right->right && !isblack(p->right->right)) ){

            Tree\_node<T1,T2>\* s = p->right;

            if(p == root) root = s; // if p is root , root change

            s->color = p->color;

            p->color = 'B';

            s->right->color = 'B';

            RR\_Rotation(s);

        }

// Case 4 :

    // s is Black

    // s\_l is Red

    // << Solution >>

    // RL rotation + Recoloring

        else if( (p->right && isblack(p->right)) && (p->right->left && p->right->left->color =='R') ){

            Tree\_node<T1,T2>\* s = p->right;

            Tree\_node<T1,T2>\* s\_l = s->left;

            if(p == root) root = s\_l; // if p is root , root change

            // Step 1 : rotate to right

            s\_l->color = 'B';

            s->color = 'R';

            RL\_Rotation(s);

            // Step 2 : RR rotation(Case 3)

            s = s\_l;

            s->color = p->color;

            p->color = 'B';

            s->right->color = 'B';

            RR\_Rotation(s);

        }

// Case 5 :

    // s is Red

    // << Solution >>

    // RR rotation + Recursive

        else if( (p->right && p->right->color=='R') ){

            if(p == root) root = p->right;

            p->color = 'R';

            p->right->color = 'B';

            RR\_Rotation(p->right);

            modify\_erase(p,'L');

        }

    }

// Delete node is right child of p

    else if(LR = 'R'){

// Case 1 : Recoloring

        if(p->color == 'R' && (p->left && isblack(p->left)) && isblack(p->left->left) && isblack(p->left->right) ){

            p->color = 'B';

            p->left->color = 'R';

        }

// Case 2 : Recoloring + recursive

        else if(p->color =='B' && (p->left && isblack(p->left)) && isblack(p->left->left) && isblack(p->left->right) ){

            p->left->color = 'R';

            if(p->parent){

                if(p->parent->left && p->parent->left == p) modify\_erase(p->parent, 'L');

                else if(p->parent->right && p->parent->right == p) modify\_erase(p->parent, 'R');

            }

        }

// Case 3 : LL rotation

        else if( (p->left && isblack(p->left)) && (p->left->left && p->left->left->color =='R') ){

            Tree\_node<T1,T2>\* s = p->left;

            if(p==root) root = s;

            s->color = p->color;

            p->color = 'B';

            s->left->color = 'B';

            LL\_Rotation(s);

        }

// Case 4 : LR rotation

        else if( (p->left && isblack(p->left)) && (p->left->right && p->left->right->color =='R') ){

            Tree\_node<T1,T2>\* s = p->left;

            Tree\_node<T1,T2>\* s\_r = s->right;

            if(p == root) root = s\_r;

            s\_r->color = 'B';

            s->color = 'R';

            LR\_Rotation(s);

        // Step 2 : LL rotate(Case 3)

            s = s\_r;

            // recoloring

            s->color = p->color;

            p->color = 'B';

            s->left->color = 'B';

            LL\_Rotation(s);

        }

// Case 5 : LL rotation + recursive

        else if( (p->left && p->left->color=='R') ){

            if(p == root ) root = p->left;

            p->color = 'R';

            p->left->color = 'B';

            LL\_Rotation(p->left);

            modify\_erase(p,'R');

        }

    }

}

// ---------------------------------------------------------------------------------------------------------------------------------------

코드의 길이는 주석까지 대략 1천줄 정도 나왔다.

Map이 이미 STL로 존재하기 때문에 굳이 왜 다시 구현을 하려 하는가 묻는다면

Map STL을 구현하면서 배우는 것들이 너무 많기 때문이다!!

Full code는 github에 올려 두었다.

Github 주소 : https://github.com/ohinhyuk/MCNL-Study/blob/main/%EA%B3%A0%EC%9C%A4%EB%AF%BC%EA%B5%90%EC%88%98%EB%8B%98%20%EC%8A%A4%ED%84%B0%EB%94%94/week%202/RedBlack%20Tree%20%EA%B8%B0%EB%B0%98%20Map%EA%B5%AC%ED%98%84/MyMap.cpp