МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный

электротехнический университет

«ЛЭТИ» им. В.И. Ульянова (Ленина)

Кафедра САПР

ОТЧЕТ

по лабораторной работе № 1

по дисциплине «Ассоциативный массив»

Студент гр. 8301 _____ Готовский К.В.

Преподаватель _____ Тутева А.В.

Санкт-Петербург

2020

Цель работы

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева.

Описание реализуемого класса

Класс Map состоит из вложенного класса Node с полями: `T key` (Ключ по которому хранится значение) и `T1 value` (Значение которое хранится по определённому ключу) `bool color` (цвет ячейки необходимый для дальнейшей проверки сбалансированности), `Node* parent` (указатель на родителя ячейки), `Node* left` и `Node* right` (Указатели на левые и правые ячейки (детей)), а также и собственные поля `Node* Top` (Вершина дерева), `Node* TNULL` (Обозначение пустого листа). Реализованный мною класс Map основан на такой структуре данных как красно-чёрное дерево. Выбранная структура данных является самобалансирующийся бинарное дерево, поэтому она содержит цвета (красный и чёрный) и 5 основных правил. При несоблюдении хотя бы одного из правил при вставке или удалении, будет требоваться балансировка. Заметим, что key и value являются шаблонными, что позволяет нам хранить ключ и значение любого типа данных.

Класс содержит следующие методы:

- Констуктор – реализованы конструкторы по умолчанию для вложенного класса Node и для самого класса Map.
- Деструктор – реализован деструктор, вызывающий метод clear().
- *insert(T key,T1 value)* – функция добавления элемента в дерево по ключу. Добавление происходит точно так же, как и в бинарном дереве. Дальше следует проверка на соблюдение 5 свойств, иначе происходит перебалансировка.
- *remove(T key)* – функция удаления элемента по ключу.
- *find(T key)* – функция получения значения по ключу.
- *clear()* – функция, по одному удаляющая элементы постфиксным обходом дерева.
- *get_keys()* – функция, возвращающая список ключей.
- *get_values()* – функция, возвращающая список значений.
- *print()*– функция вывода дерева.

Оценка временной сложности алгоритмов

- *remove(T key)* – O(log n)
- *insert(T key,T1 value)* – O(log n)
- *find(T key)* – O(log n)
- *clear()* – O(n)
- *get_keys()* – O(1)
- *get_values()* – O(1)
- *print()* – O(n)

Описание реализованных unit-тестов

В тестах реализованных для класса Map мы проверили функцию добавления элемента в дерево с помощью функции *insert()* и удаляем с помощью функции remove(), и проверяем их с помощью функции find() которая возвращает нам значение по ключу или бросает

исключение. Так же мои unit-тесты затрагивают такие методы как get_keys() и get_values(), которые возвращают списки ключей и значений, а также метод очистки дерева clear().

Пример работы программы

Пример

Листинг

Map.h

```
1.  #define COLOR_RED 1
2.  #define COLOR_BLACK 0
3.  #include"List.h"
4.  using namespace std;
5.  template<typename T, typename T1>
6.  class Map {
7.  public:
8.      class Node
9.      {
10.     public:
11.         Node(bool color = COLOR_RED, T key = T(), Node* parent = NULL, Node* left = NULL, Node* right = NU
12.         T key;
13.         T1 value;
14.         bool color;
15.         Node* parent;
16.         Node* left;
17.         Node* right;
18.     };
19.     ~Map() {
20.         if (this->Top != NULL)
21.             this->clear();
22.         Top = NULL;
23.         delete TNULL;
24.         TNULL = NULL;
25.     }
26.     Map(Node* Top = NULL, Node* TNULL = new Node(0)) :Top(Top), TNULL(TNULL) {}
27.
28.     void printTree()
29.     {
30.         if (Top)
31.         {
32.             print_Helper(this->Top, "", true);
33.         }
34.         else throw std::out_of_range("Tree is empty!");
35.     }
36.
37.     void  insert(T key, T1 value)
38.     {
39.
40.         if (this->Top != NULL) {
41.             Node* node = NULL;
42.             Node* parent=NULL;
43.             /* Search leaf for new element */
44.             for (node = this->Top; node != TNULL; )
45.             {
```

```
                parent = node;
                if (key < node->key)
                    node = node->left;
                else if (key >= node->key)
                    node = node->right;
            }

            node = new Node(COLOR_RED, key, TNULL, TNULL, TNULL, value);
            node->parent = parent;


            if (parent != TNULL) {
                if (key < parent->key)
                    parent->left = node;
                else
                    parent->right = node;
            }
            rbtree_fixup_add(node);
        }
        else {
            this->Top = new Node(COLOR_BLACK, key, TNULL, TNULL, TNULL, value);
        }
    }
    List<T>* get_keys() {
        List<T>* list = new List<T>();
        this->ListKeyOrValue(1,list);
        return list;
    }
    List<T1>* get_values() {
        List<T1>* list = new List<T1>();
        this->ListKeyOrValue(2, list);
        return list;
    }
    T1 find(T key) {
        Node* node = Top;

        while (node != TNULL && node->key != key) {
            if (node->key > key)
                node = node->left;
            else
                if (node->key < key)
                    node = node->right;
        }
        if (node != TNULL)
            return node->value;
        else
            throw std::out_of_range("Key is missing");
    }
    void remove(T key) {
        this->deleteNodeHelper(this->find_key(key));
    }

    void clear() {
        this->clear_tree(this->Top);
            this->Top = NULL;
        }
      private:
          Node* Top;
          Node* TNULL;

      ////////////////////////////////////////////////////////////////////////
          //delete functions
      ////////////////////////////////////////////////////////////////////////
          void deleteNodeHelper(Node* z)
          {
              Node* x, * y;
              y = z;
```

```
113.                int y_original_color = y->color;
114.                if (z->left == TNULL)
115.                {
116.                    x = z->right;
117.                    Transplant(z, z->right);
118.                }
119.                else if (z->right == TNULL)
120.                {
121.                    x = z->left;
122.                    Transplant(z, z->left);
123.                }
124.                else
125.                {
126.                    y = minimum(z->right);
127.                    y_original_color = y->color;
128.                    x = y->right;
129.                    if (y->parent == z)
130.                    {
131.                        x->parent = y;
132.                    }
133.                    else
134.                    {
135.                        Transplant(y, y->right);
136.                        y->right = z->right;
137.                        y->right->parent = y;
138.                    }
139.                    Transplant(z, y);
140.                    y->left = z->left;
141.                    y->left->parent = y;
142.                    y->color = z->color;
143.                }
144.                delete z;
145.                if (y_original_color == COLOR_BLACK)
146.                {
147.                    deleteFix(x);
148.                }
149.            }
150.            //swap links(parent and other) for rotate
151.            void Transplant(Node* cur, Node* cur1)
152.            {
153.                if (cur->parent == TNULL)
154.                {
155.                    Top = cur1;
156.                }
157.                else if (cur == cur->parent->left)
158.                {
159.                    cur->parent->left = cur1;
160.                }
161.                else
162.                {
163.                    cur->parent->right = cur1;
164.                }
165.                cur1->parent = cur->parent;
166.            }
167.
168.
169.            void deleteFix(Node* x)
170.            {
171.                Node* sibling;
172.                while (x != Top && x->color == 0)
173.                {
174.                    if (x == x->parent->left)
175.                    {
176.                        sibling = x->parent->right;
177.                        if (sibling->color == COLOR_RED)
178.                        {
179.                            sibling->color = COLOR_BLACK;
```

```
180.                            x->parent->color = COLOR_RED;
181.                            left_rotate(x->parent);
182.                            sibling = x->parent->right;
183.                        }
184.                        if (sibling->left->color == COLOR_BLACK && sibling->right->color == COLOR_BLACK)
185.                        {
186.                            sibling->color = COLOR_RED;
187.                            x = x->parent;
188.                        }
189.                        else
190.                        {
191.                            if (sibling->right->color == COLOR_BLACK)
192.                            {
193.                                sibling->left->color = COLOR_BLACK;
194.                                sibling->color = COLOR_RED;
195.                                right_rotate(sibling);
196.                                sibling = x->parent->right;
197.                            }
198.                            sibling->color = x->parent->color;
199.                            x->parent->color = COLOR_BLACK;
200.                            sibling->right->color = COLOR_BLACK;
201.                            left_rotate(x->parent);
202.                            x = Top;
203.                        }
204.                    }
205.                    else
206.                    {
207.                        sibling = x->parent->left;
208.                        if (sibling->color == COLOR_RED)
209.                        {
210.                            sibling->color = COLOR_BLACK;
211.                            x->parent->color = COLOR_RED;
212.                            right_rotate(x->parent);
213.                            sibling = x->parent->left;
214.                        }
215.                        if (sibling->right->color == COLOR_BLACK && sibling->right->color == COLOR_BLACK)
216.                        {
217.                            sibling->color = 1;
218.                            x = x->parent;
219.                        }
220.                        else
221.                        {
222.                            if (sibling->left->color == COLOR_BLACK)
223.                            {
224.                                sibling->right->color = COLOR_BLACK;
225.                                sibling->color = COLOR_RED;
226.                                left_rotate(sibling);
227.                                sibling = x->parent->left;
228.                            }
229.                            sibling->color = x->parent->color;
230.                            x->parent->color = COLOR_BLACK;
231.                            sibling->left->color = COLOR_BLACK;
232.                            right_rotate(x->parent);
233.                            x = Top;
234.                        }
235.                    }
236.                }
237.                x->color = COLOR_BLACK;
238.            }
239.            void clear_tree(Node* tree) {
240.                if (tree != TNULL) {
241.                    clear_tree(tree->left);
242.                    clear_tree(tree->right);
243.                    delete tree;
244.                }
245.            }
246.        ////////////////////////////////////////////////////////////////////////////////////////
```

```
247.            //all find functions
248.        ////////////////////////////////////////////////////////////////////////////////
249.            Node* minimum(Node* node)
250.            {
251.                while (node->left != TNULL)
252.                {
253.                    node = node->left;
254.                }
255.                return node;
256.            }
257.            Node* maximum(Node* node)
258.            {
259.                while (node->right != TNULL)
260.                {
261.                    node = node->right;
262.                }
263.                return node;
264.            }
265.            Node* grandparent(Node* cur)
266.            {
267.                if ((cur != TNULL) && (cur->parent != TNULL))
268.                    return cur->parent->parent;
269.                else
270.                    return TNULL;
271.            }
272.            Node* uncle(Node* cur)
273.            {
274.                Node* cur1 = grandparent(cur);
275.                if (cur1 == TNULL)
276.                    return TNULL; // No grandparent means no uncle
277.                if (cur->parent == cur1->left)
278.                    return cur1->right;
279.                else
280.                    return cur1->left;
281.            }
282.            Node* sibling(Node* n)
283.            {
284.                if (n == n->parent->left)
285.                    return n->parent->right;
286.                else
287.                    return n->parent->left;
288.            }
289.            Node* find_key(T key) {
290.                Node* node = this->Top;
291.                while (node != TNULL && node->key != key) {
292.                    if (node->key > key)
293.                        node = node->left;
294.                    else
295.                        if (node->key < key)
296.                            node = node->right;
297.                }
298.                if (node != TNULL)
299.                    return node;
300.                else
301.                    throw std::out_of_range("Key is missing");
302.            }
303.            ////////////////////////////////////////////////////////////////////////////////
304.              //all print function
305.            ////////////////////////////////////////////////////////////////////////////////
306.            void print_Helper(Node* root, string indent, bool last)
307.            {
308.                if (root != TNULL)
309.                {
310.                    cout << indent;
311.                    if (last)
312.                    {
313.                        cout << "R----";
```

```
314.                     indent += "   ";
315.                 }
316.                 else
317.                 {
318.                     cout << "L----";
319.                     indent += "|  ";
320.                 }
321.                 string sColor = !root->color ? "BLACK" : "RED";
322.                 cout << root->key << "(" << sColor << ")" << endl;
323.                 print_Helper(root->left, indent, false);
324.                 print_Helper(root->right, indent, true);
325.             }
326.         }
327.         void ListKeyOrValue(int mode,List<T>*list) {
328.             if (this->Top != TNULL)
329.                 this->KeyOrValue(Top,list, mode);
330.             else
331.                 throw std::out_of_range("Tree empty!");
332.         }
333.         void KeyOrValue(Node* tree,List<T>*list, int mode) {
334.
335.             if (tree != TNULL) {
336.                 KeyOrValue(tree->left,list, mode);
337.                 if (mode == 1)
338.                     list->push_back(tree->key);
339.                 else
340.                     list->push_back(tree->value);
341.                 KeyOrValue(tree->right,list, mode);
342.             }
343.         }
344.         ////////////////////////////////////////////////////////////////////////////////
345.           //fix before add
346.         ////////////////////////////////////////////////////////////////////////////////////////
347.         void rbtree_fixup_add(Node* node)
348.         {
349.             Node* NullNode = new Node(COLOR_BLACK);
350.             Node* uncle;
351.             /* Current node is COLOR_RED */
352.             while (node != this->Top && node->parent->color == COLOR_RED)//
353.             {
354.                 /* node in left tree of grandfather */
355.                 if (node->parent == this->grandparent(node)->left)//
356.                 {
357.                     /* node in left tree of grandfather */
358.                     uncle = this->uncle(node);
359.                     if (uncle->color == COLOR_RED) {
360.                         /* Case 1 - uncle is COLOR_RED */
361.                         node->parent->color = COLOR_BLACK;
362.                         uncle->color = COLOR_BLACK;
363.                         this->grandparent(node)->color = COLOR_RED;
364.                         node = this->grandparent(node);
365.                     }
366.                     else {
367.                         /* Cases 2 & 3 - uncle is COLOR_BLACK */
368.                         if (node == node->parent->right) {
369.                             /*Reduce case 2 to case 3 */
370.                             node = node->parent;
371.                             this->left_rotate(node);
372.                         }
373.                         /* Case 3 */
374.                         node->parent->color = COLOR_BLACK;
375.                         this->grandparent(node)->color = COLOR_RED;
376.                         this->right_rotate(this->grandparent(node));
377.                     }
378.                 }
379.                 else {
380.                     /* Node in right tree of grandfather */
```

```
                        uncle = this->uncle(node);
                        if (uncle->color == COLOR_RED) {
                            /* Uncle is COLOR_RED */
                            node->parent->color = COLOR_BLACK;
                            uncle->color = COLOR_BLACK;
                            this->grandparent(node)->color = COLOR_RED;
                            node = this->grandparent(node);
                        }
                        else {
                            /* Uncle is COLOR_BLACK */
                            if (node == node->parent->left) {
                                node = node->parent;
                                this->right_rotate(node);
                            }
                            node->parent->color = COLOR_BLACK;
                            this->grandparent(node)->color = COLOR_RED;
                            this->left_rotate(this->grandparent(node));
                        }
                    }
                }
                this->Top->color = COLOR_BLACK;
            }
            ////////////////////////////////////////////////////////////////////////////
              //Rotates
            ////////////////////////////////////////////////////////////////////////////
            //(left rotate)
            void left_rotate(Node* node)
            {
                Node* right = node->right;
                /* Create node->right link */
                node->right = right->left;
                if (right->left != TNULL)
                    right->left->parent = node;
                /* Create right->parent link */
                if (right != TNULL)
                    right->parent = node->parent;
                if (node->parent != TNULL) {
                    if (node == node->parent->left)
                        node->parent->left = right;
                    else
                        node->parent->right = right;
                }
                else {
                    this->Top = right;
                }
                right->left = node;
                if (node != TNULL)
                    node->parent = right;
            }
            //(right rotate)
            void right_rotate(Node* node)
            {
                Node* left = node->left;
                /* Create node->left link */
                node->left = left->right;
                if (left->right != TNULL)
                    left->right->parent = node;
                /* Create left->parent link */
                if (left != TNULL)
                    left->parent = node->parent;
                if (node->parent != TNULL) {
                    if (node == node->parent->right)
                        node->parent->right = left;
                    else
                        node->parent->left = left;
                }
                else {
```

```
448.                 this->Top = left;
449.             }
450.             left->right = node;
451.             if (node != TNULL)
452.                 node->parent = left;
453.         }
454.         ////////////////////////////////////////////////////////////////////////////////////////////////
455.     };
456.
```

RedBlackTreeTest.cpp

```
1.  #include "pch.h"
2.  #include "CppUnitTest.h"
3.  #include"../RedBlackTree/Map.h"
4.  using namespace Microsoft::VisualStudio::CppUnitTestFramework;
5.
6.  namespace RedBlackTreeTest
7.  {
8.      TEST_CLASS(RedBlackTreeTest)
9.      {
10.     public:
11.
12.         TEST_METHOD(TestMethodInsertAndFind)
13.         {
14.             Map<int, int>* tree = new Map<int, int>();
15.             tree->insert(10, -1);
16.             tree->insert(5, -2);
17.             tree->insert(3, -3);
18.             tree->insert(11, -4);
19.             Assert::AreEqual(tree->find(5), -2);
20.
21.         }
22.         TEST_METHOD(TestMethodFindExeption)
23.         {
24.             try {
25.                 Map<int, int>* tree = new Map<int, int>();
26.                 tree->insert(10, -1);
27.                 tree->insert(5, -2);
28.                 tree->insert(3, -3);
29.                 tree->insert(11, -4);
30.                 tree->find(29);
31.             }
32.             catch (std::out_of_range exc) {
33.                 Assert::AreEqual("Key is missing", exc.what());
34.             }
35.         }
36.         TEST_METHOD(TestMethodClear)
37.         {
38.             try {
39.                 Map<int, int>* tree = new Map<int, int>();
40.                 tree->insert(10, -1);
41.                 tree->insert(5, -2);
42.                 tree->insert(3, -3);
43.                 tree->insert(11, -4);
44.                 tree->clear();
45.                 tree->printTree();
46.             }
47.             catch (std::out_of_range exc) {
48.                 Assert::AreEqual("Tree is empty!", exc.what());
49.             }
50.         }
51.         TEST_METHOD(TestMethodGetValues)
52.         {
53.                 Map<int, int>* tree = new Map<int, int>();
54.                 tree->insert(10, -1);
```

```
                tree->insert(5, -2);
                tree->insert(3, -3);
                tree->insert(11, -4);
                List<int>* list = tree->get_values();
                int sum = 0;
                for (int i = 0; i < list->get_size(); i++)
                    sum+=list->at(i);
                Assert::AreEqual(-10,sum);
        }
        TEST_METHOD(TestMethodGetKeys)
        {
            Map<int, int>* tree = new Map<int, int>();
            tree->insert(10, -1);
            tree->insert(5, -2);
            tree->insert(3, -3);
            tree->insert(11, -4);
            List<int>* list = tree->get_keys();
            int sum = 0;
            for (int i = 0; i < list->get_size(); i++)
                sum += list->at(i);
            Assert::AreEqual(29, sum);
        }
        TEST_METHOD(TestMethodRemove)
        {
            try {
                Map<int, int>* tree = new Map<int, int>();
                tree->insert(10, -1);
                tree->insert(5, -2);
                tree->insert(3, -3);
                tree->insert(11, -4);
                tree->remove(10);
                tree->find(10);
            }
            catch (std::out_of_range exc) {
                Assert::AreEqual("Key is missing", exc.what());
            }
        }
    };
}
```

Вывод

В данной лабораторной работе я познакомился с такой структурой данных как очередь и реализовал её.