Лабораторная работа №7

Задача: Необходимо реализовать динамическую структуру данных – "Хранилище объектов"и алгоритм работы с ней. "Хранилище объектов"представляет собой контейнер стек. Каждым элементом контейнера является динамическая структура список. Таким образом, у нас получается контейнер в контейнере. Элементов второго контейнера является объект-фигура, определенная вариантом задания.
При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше 5. Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня.
Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта. При удалении объектов должно выполянться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалиться.

Фигуры: треугольник, квадрат, прямоугольник.
Контейнер 1-ого уровня: связный список.
Контейнер 2-ого уровня: бинарное дерево.

1 Введение

Каждый раз, когда в программе возникает необходимость оперировать множеством элементов, в дело вступают контейнеры. Контейнер – это практическая реализация функциональности некоторой структуры данных. Шаблоны (Templates) в C++ предоставляют замечательную возможность реализовать контейнер один раз, формализовать его внешние интерфейсы, дать асимптотические оценки времени выполнения каждой из операций, а после этого просто пользоваться подобным контейнером с любым типом данных.
В данной лабораторной работе мне надо было реализовать контейнер в контейнере.

2 Код программы

tbinarytree.cpp

```
#include <iostream>
#include <cstdlib>
#include <memory>

#include "figure.h"
#include "tbinary_tree.h"
#include "tbinary_tree_item.h"

template <class T>
TBinaryTree<T>::TBinaryTree() {
root = nullptr;
count = 0;
}

template <class T>
void TBinaryTree<T>::Insert(std::shared_ptr<T>val) {
if (root == nullptr) {
root = TreeNodePtr<T>(new TreeNode<T>(val));
}
else {
TreeNodePtr<T>curNode = root;
TreeNodePtr<T>parNode = nullptr;
while (curNode != nullptr) {
parNode = curNode;
if (val->SquareLess(curNode->data))
```

```
curNode = curNode->left;
else
curNode = curNode->right;
}
if (val->SquareLess(parNode->data))
parNode->left = TreeNodePtr<T>(new TreeNode<T>(val,parNode));
else
parNode->right = TreeNodePtr<T>(new TreeNode<T>(val,parNode));
}
count++;
}


template <class T>
std::shared_ptr<T>TBinaryTree<T>::Find(std::shared_ptr<T>key)
{
if (root == nullptr)
return nullptr;

TreeNodePtr<T>curNode = root;
TreeNodePtr<T>parNode = nullptr;
while (curNode != nullptr) {
parNode = curNode;
if (key->TypedEquals(curNode->data))
return curNode->data;
if (key->SquareLess(curNode->data))
curNode = curNode->left;
else
curNode = curNode->right;
}
return nullptr;
}

template<class T>
size_t TBinaryTree<T>::GetCount()
{
return count;
}

template<class T>
TBinaryTreeIterator<T>TBinaryTree<T>::begin()
```

```
{
return TBinaryTreeIterator<T>(root);
}

template<class T>
TBinaryTreeIterator<T>TBinaryTree<T>::end()
{
return TBinaryTreeIterator<T>(nullptr);
}


template<class T>
TreeNodePtr<T>TBinaryTree<T>::MinValueTreeNode(TreeNodePtr<T>node)
{
TreeNodePtr<T>current = node;

while (current->left != nullptr)
current = current->left;

return current;
}

template<class T>
TreeNodePtr<T>TBinaryTree<T>::deleteTreeNode(TreeNodePtr<T>_root,std::shared_ptr<T>key
{
if (_root == nullptr) return _root;
if (key->TypedEquals(_root->data)) {
if (_root->left == nullptr) {
TreeNodePtr<T>temp = _root->right;
return temp;
}
else if (_root->right == nullptr) {
TreeNodePtr<T>temp = _root->left;
return temp;
}

TreeNodePtr<T>temp = MinValueTreeNode(_root->right);
_root->data = temp->data;
_root->right = deleteTreeNode(_root->right,temp->data);
}
else if (key->SquareLess(_root->data))
```

```
_root->left = deleteTreeNode(_root->left,key);
else
_root->right = deleteTreeNode(_root->right,key);

return _root;
}

template<class T>
std::ostream & TBinaryTree<T>::InOrderPrint(std::ostream & os,TreeNodePtr<T>node,size_
level) const
{
if (node != nullptr) {
InOrderPrint(os,node->left,level + 1);
for (size_t i = 0; i <level; i++)
os <<'\t';
node->data->print(os);
InOrderPrint(os,node->right,level + 1);
}
return os;
}

template <class T>
void TBinaryTree<T>::Delete(std::shared_ptr<T>key)
{
if (key == nullptr)
return;
root = deleteTreeNode(root,key);
count--;
}

template <class A>
std::ostream& operator<<(std::ostream& os,TBinaryTree<A>& bintree) {
if (bintree.IsEmpty())
os <<"Empty tree\n";
else
bintree.InOrderPrint(os,bintree.root,0);
return os;
}

template <class T>
bool TBinaryTree<T>::IsEmpty() const {
```

```
return (root == nullptr);
}

template <class T>
TBinaryTree<T>::~TBinaryTree() {
root = nullptr;
#ifdef DEBUG
std::cout <<"bintree dtr\n";
#endif // DEBUG
}

template class TBinaryTree<Figure>;
template std::ostream& operator<<(std::ostream& os,TBinaryTree<Figure>& tree);
```

tbinarytree.h

```
#ifndef TBINARY_TREE_H
#define TBINARY_TREE_H

#include "tbinary_tree_item.h"
#include "TAllocationBlock.h"
#include "tbinarytree_iterator.h"

template <class T>
class TBinaryTree
{
private:
size_t count;
TreeNodePtr<T>root;
TreeNodePtr<T>MinValueTreeNode(TreeNodePtr<T>node);
TreeNodePtr<T>deleteTreeNode(TreeNodePtr<T>_root,std::shared_ptr<T>key);
std::ostream& InOrderPrint(std::ostream& os,TreeNodePtr<T>node,size_t level)
const;
public:
TBinaryTree();
void Insert(std::shared_ptr<T>figure);
bool IsEmpty() const;
void Delete(std::shared_ptr<T>key);
std::shared_ptr<T>Find(std::shared_ptr<T>key);
size_t GetCount();

TBinaryTreeIterator<T>begin();
```

```
TBinaryTreeIterator<T>end();

template <class A>friend std::ostream& operator<<(std::ostream& os,TBinaryTree<A>&
bintree);
virtual ~TBinaryTree();
};

#endif /* TBINARY_TREE_H */
```

storage.h

```
#ifndef STORAGE_H
#define STORAGE_H

#include "tlist.h"
#include "tbinary_tree.h"
#include "Criterion.h"


template <class T>
class TStorage
{
private:
TList<TBinaryTree<T>>storage;

public:
TStorage() {
}

~TStorage() {
}

void Insert(std::shared_ptr<T>item) {
if (storage.IsEmpty())
{
TBinaryTree<T>tree;
tree.Insert(item);
storage.Push(tree);
}
else
{
TBinaryTree<T>& top = storage.Top();
```

```
if (top.GetCount() <5)
{
top.Insert(item);
}
else
{
TBinaryTree<T>tree;
tree.Insert(item);
storage.Push(tree);
}
}
TBinaryTree<T>& top = storage.Top();

}

void DeleteByCriteria(IRemoveCriteria<T>&crit) {
auto it_list = storage.begin();
while (it_list != storage.end()) {
TList<std::shared_ptr<T>>figuresToDelete;
for (auto it_tree = (*it_list)->begin(); it_tree != (*it_list)->end(); it_tree++)
{
if (crit(*it_tree))
figuresToDelete.Push(*it_tree);
}
bool needToDeleteTree = (figuresToDelete.GetLength() == (*it_list)->GetCount());
for (auto it_figlist = figuresToDelete.begin(); it_figlist != figuresToDelete.end();
it_figlist++) {
(*it_list)->Delete(**it_figlist);
}
if (needToDeleteTree) {
auto it_tmp = it_list;
it_list++;
storage.Delete(it_tmp);
}
else
it_list++;
}
}


friend std::ostream & operator<<(std::ostream & os,TStorage<T>& stor) {
```

```
size_t i = stor.storage.GetLength() -1;
if (i == -1) {
std::cout <<"Пустой контейнер";
}
for (auto it_list = stor.storage.begin(); it_list != stor.storage.end(); it_list++)
{
std::cout <<"Контейнер(" <<i--<<")"<<std::endl;
std::cout <<**it_list;
}
return os;
}
};


#endif /* STORAGE_H */
```

Criterion.h

```
#ifndef IREMOVECRITERIA_H
#define IREMOVECRITERIA_H

#include "figure.h"
#include <memory>
#include <typeindex>

template <class T>
class IRemoveCriteria
{
public:
virtual bool operator()(std::shared_ptr<T>value) = 0;
};


class RemoveCriteriaByMaxSquare : public IRemoveCriteria<Figure>
{
public:
RemoveCriteriaByMaxSquare(double value)
{
_MaxSquareValue = value;
}
bool operator()(std::shared_ptr<Figure>value) override
{
```

```
return value->Square() <_MaxSquareValue;
}
private:
double _MaxSquareValue;
};


class RemoveCriteriaByFigureType : public IRemoveCriteria<Figure>
{
public:
RemoveCriteriaByFigureType(const char * value)
{
_TypeName = new char[strlen(value) + 1];
strcpy(_TypeName,value);
}

bool operator()(std::shared_ptr<Figure>value) override
{
return strcmp(typeid(*value).name() + 6,_TypeName) == 0;
}

~RemoveCriteriaByFigureType()
{
delete _TypeName;
}
private:
char * _TypeName;
};


#endif
```

tbinarytreeitem.cpp

```
#include <iostream>
#include <cstdlib>
#include <memory>

#include "tbinary_tree.h"

template <class T>TAllocationBlock
TreeNode<T>::allocator(sizeof(TreeNode<T>),MAX_TREE_CAPACITY);
```

```
template <class T>
TreeNode<T>::TreeNode() {
left = nullptr;
right = nullptr;
parent = nullptr;
}

template<class T>
TreeNode<T>::TreeNode(std::shared_ptr<T>data,TreeNodePtr<T>parent)
{
left = nullptr;
right = nullptr;
this->parent = parent;
this->data = data;
}

template<class T>
std::shared_ptr<T>TreeNode<T>::GetPtr()
{
return data;
}

template <class T>
void *TreeNode<T>::operator new(size_t size)
{
return allocator.Allocate();
}

template <class T>
void TreeNode<T>::operator delete(void *ptr)
{
allocator.Deallocate(ptr);
}

template class TreeNode<Figure>;
```

tbinarytreeitem.h

```
#ifndef TBINARY_TREE_ITEM_H
#define TBINARY_TREE_ITEM_H
```

```
#include "figure.h"
#include "tbinarytree_iterator.h"
#include "TAllocationBlock.h"
#include "tbinary_tree.h"
#include <memory>

template<class T>
class TreeNode;

template<class T>
class TBinaryTreeIterator;

template <class T>
using TreeNodePtr = std::shared_ptr<TreeNode<T>>;

#define MAX_TREE_CAPACITY 100

template<class T>
class TreeNode {
public:
TreeNode();
TreeNode(std::shared_ptr<T>data,TreeNodePtr<T>parent = nullptr);
std::shared_ptr<T>GetPtr();

void *operator new(size_t size);
void operator delete(void *ptr);

friend class TBinaryTree<T>;
friend class TBinaryTreeIterator<T>;
private:
TreeNodePtr<T>left;
TreeNodePtr<T>right;
TreeNodePtr<T>parent;
std::shared_ptr<T>data;

static TAllocationBlock allocator;
};

#endif /* TBINARY_TREE_ITEM_H */
```

3 Вывод программы:

```
------------------------------
-------------МЕНЮ-------------
1-Добавить треугольник
2-Добавить прямоугольник
3-Добавить квадрат
4-Удалить фигуру по площади
5-Удалить трегольники
6-Удалить прямоугольники
7-Удалить квадраты
8-Расспечатать
0-Выход
Выберете действие:
1
Введите значение a:3
Введите значение b:4
Введите значение c:5
Выберете действие:
2
Введите значение a:2
Введите значение b:2
Выберете действие:
3
Введите значение a:3
Выберете действие:
2
Введите значение a:1
Введите значение b:4
Выберете действие:
3
Введите значение a:4
Выберете действие:
2
Введите значение a:1
Введите значение b:7
Выберете действие:
8
Контейнер(1)
Прямоугольник со сторонами [1,7],Площадь = 7
Контейнер(0)
```

```
Прямоугольник со сторонами [2,2],Площадь = 4
Прямоугольник со сторонами [1,4],Площадь = 4
Треугольник со сторонами [3,4,5],Площадь = 6
Квадрат со стороной [3],Площадь = 9
Квадрат со стороной [4],Площадь = 16

Выберете действие:
7
Выберете действие:
8
Контейнер(1)
Прямоугольник со сторонами [1,7],Площадь = 7
Контейнер(0)
Прямоугольник со сторонами [2,2],Площадь = 4
Прямоугольник со сторонами [1,4],Площадь = 4
Треугольник со сторонами [3,4,5],Площадь = 6

Выберете действие:
3
Введите значение a:1
Выберете действие:
8
Контейнер(1)
Квадрат со стороной [1],Площадь = 1
Прямоугольник со сторонами [1,7],Площадь = 7
Контейнер(0)
Прямоугольник со сторонами [2,2],Площадь = 4
Прямоугольник со сторонами [1,4],Площадь = 4
Треугольник со сторонами [3,4,5],Площадь = 6

Выберете действие:
4
Удалить все объекты с площадью меньше,чем: 5
Выберете действие:
8
Контейнер(1)
Прямоугольник со сторонами [1,7],Площадь = 7
Контейнер(0)
Треугольник со сторонами [3,4,5],Площадь = 6

Выберете действие:
```

```
9
Неверный ввод. Попробуйте снова
Выберете действие:
0
```