Лабораторна робота №2 + Проект

Тема: Операції над бінарними деревами з використанням паттернів проектування

Виконав: Писаренков Тимофій, ІПС-22

Використані паттерни:

* Composite – структура бінарного дерева, розділ вузлів дерева на листки і нелистки
* Singleton/Factory Method – створення бінарного дерева
* Iterator – симетричний, прямий та обернений обходи бінарного дерева
* Strategy – вибір методу обходу бінарного дерева і запуск обходу
* Command – операції зі збереження/скасування дії бінарного дерева
* Memento – операція скасування дії над бінарним деревом
* Facade – спрощення виконання дій над бінарним деревом у результаті використання тільки одного класу у клієнтському коді
* Template Method – виведення деяких деталей про бінарне дерево, зокрема висоти, макс./мін. значень, суми усіх ключів або найдовшого ключа
* Adapter – перетворення дерева на структуру, яку зручно зберігати у файл: перетворення усіх ключів на рядки, збереження типу даних ключів, типу дерева; а також навпаки: перетворення вищевказаної структури на дерево заданого типу із ключами заданого типу даних.

Приклади використання вищевказаних патернів із коду:

Composite: template<typename T>

class Node{

protected:

//! Key of the node, can be of any type (int/double/char/std::string)

T key\_;

//TypeNode type\_;

//! Parent of the node

Node\* parent\_;

public:

Node() : key\_(T(Converter<int>(0))), parent\_(nullptr){}

Node(T key) : key\_(key), parent\_(nullptr){}

Node(T key, Node\* parent) : key\_(key), parent\_(parent){}

virtual ~Node(){}

virtual void Add(T key){}

virtual void AddToStringedNode(std::string key){}

virtual Node\* AddDummy() { return this; }

virtual Node\* Search(T key) { return nullptr; }

virtual Node<T>\* Remove(T key) { return this; }

virtual Node\* RemoveDummy() { return this; }

virtual void setLeft(Node\* n){}

virtual void setRight(Node\* n){}

virtual Node\* getLeft() { return nullptr; }

virtual Node\* getRight() { return nullptr; }

void setParent(Node\* p){

}

Node\* setDummyParent() {

}

Node\* RemoveDummyParent() {

}

Node\* getParent() {

}

T getKey() {

}

virtual Node\* getSuccessor() { return nullptr; }

virtual Node\* getMin() { return nullptr; }

virtual Node\* getMax() { return nullptr; }

virtual bool isLeaf(){

return false;

}

virtual void setChild() {

}

TreeIterator<T> \*CreateInOrderIterator(){

}

TreeIterator<T> \*CreatePreOrderIterator(){

}

TreeIterator<T>\* CreatePostOrderIterator() {

}

int getHeight(int depth = 1) {

}

std::string getLongestWord() { return ""; }

T getSumOfAllNodes() {

}

};

template<typename T>

class NodeCompos : virtual public Node<T>{

protected:

Node<T>\* left\_;

Node<T>\* right\_;

public:

NodeCompos() : left\_(nullptr), right\_(nullptr){}

NodeCompos(Node<T>\* left, Node<T>\* right) : left\_(left), right\_(right){}

void setLeft(Node<T>\* n) override {

this->left\_ = n;

}

void setRight(Node<T>\* n) override {

this->right\_ = n;

}

Node<T>\* getLeft() override {

return this->left\_;

}

Node<T>\* getRight() override {

return this->right\_;

}

};

template<typename T>

class NodeLeaf : virtual public Node<T>{

public:

NodeLeaf(){}

bool isLeaf() override{

return true;

}

};

//! Class for Binary Search Tree nodes

template<typename T>

class BinaryNode : virtual public Node<T>{

public:

BinaryNode(){

}

};

template<typename T>

class BinaryNodeLeaf;

//! Class for composite Binary Search Tree nodes

template<typename T>

class BinaryNodeCompos : public BinaryNode<T>, public NodeCompos<T>{

public:

BinaryNodeCompos() : Node<T>(), NodeCompos<T>(){}

BinaryNodeCompos(T key) : Node<T>(key), NodeCompos<T>(){}

BinaryNodeCompos(BinaryNodeLeaf<T> \*leaf) : Node<T>(leaf->getKey(), leaf->getParent()), NodeCompos<T>(){

this->setChild();

}

void Add(T key) override{

if(key < this->key\_)

…

}

Node<T>\* Remove(T key) override {

if (key < this->key\_)

…

};

Node<T>\* AddDummy() override {

}

Node<T>\* RemoveDummy() override {

}

Node<T>\* getSuccessor() override {

}

Node<T>\* getMin() override {

}

Node<T>\* getMax() override {

}

void setChild() {

}

void setSNChild() ;

};

Singleton/Factory:

class TreeFactory{

private:

static TreeFactory\* treefac;

protected:

TreeFactory() {};

~TreeFactory() {};

public:

TreeFactory(TreeFactory &other) = delete;

void operator=(const TreeFactory &) = delete;

static TreeFactory\* getTreeFac();

Node<int>\* createTree(TreeType type, int rootkey){

…

}

Node<double>\* createTree(TreeType type, double rootkey){

…

}

Node<std::string>\* createTree(TreeType type, std::string rootkey){

…

}

};

TreeFactory\* TreeFactory::treefac{nullptr};

TreeFactory\* TreeFactory::getTreeFac(){

if(!treefac)

{

treefac = new TreeFactory();

}

return treefac;

}

Iterator:

template <typename KeyType>

class TreeIterator{

protected:

Node<KeyType>\* First\_;

Node<KeyType>\* Last\_;

Node<KeyType>\* iter\_;

Node<KeyType>\* max\_;

public:

TreeIterator() : First\_(nullptr), Last\_(nullptr), iter\_(nullptr), max\_(nullptr){}

virtual ~TreeIterator() {}

virtual void First(){ iter\_ = First\_; }

virtual void Next(){}

virtual bool isDone(){

return (iter\_ == Last\_);

}

virtual Node<KeyType>\* Current(){

return iter\_;

}

};

template<typename KeyType>

class InOrderIterator : public TreeIterator<KeyType>{

public:

InOrderIterator() {}

InOrderIterator(Node<KeyType>\* root) {

this->First\_ = root->getMin();

this->iter\_ = this->First\_;

this->max\_ = root->getMax()->AddDummy();

this->Last\_ = this->max\_->getRight();

}

~InOrderIterator() {

this->max\_ = this->max\_->RemoveDummy();

}

void Next() override{

if (this->iter\_->getRight()) {

this->iter\_ = this->iter\_->getSuccessor();

}

else {

Node<KeyType>\* parent = this->iter\_->getParent();

while (/\*parent->getRight() ||\*/ parent->getRight() == this->iter\_) {

this->iter\_ = parent;

parent = parent->getParent();

}

this->iter\_ = parent;

}

}

};

template<typename KeyType>

class PreOrderIterator : public TreeIterator<KeyType> {

public:

PreOrderIterator(){}

PreOrderIterator(Node<KeyType>\* root){

this->First\_ = root;

this->iter\_ = this->First\_;

this->max\_ = root->getMax()->AddDummy();

this->Last\_ = this->max\_->getRight();

}

~PreOrderIterator() {

this->max\_ = this->max\_->RemoveDummy();

}

void Next() override {

if (this->iter\_->getLeft()) {

this->iter\_ = this->iter\_->getLeft();

}

else if (this->iter\_->getRight())

{

this->iter\_ = this->iter\_->getRight();

}

else {

Node<KeyType>\* parent = this->iter\_->getParent();

while (!parent->getRight() || parent->getRight() == this->iter\_)

{

this->iter\_ = parent;

parent = parent->getParent();

}

this->iter\_ = parent->getRight();

}

}

};

template<typename KeyType>

class PostOrderIterator : public TreeIterator<KeyType> {

public:

PostOrderIterator() {}

PostOrderIterator(Node<KeyType>\* root) {

this->First\_ = root->getMin();

this->iter\_ = this->First\_;

this->max\_ = root;

this->Last\_ = root->setDummyParent();

}

~PostOrderIterator() {

this->max\_ = this->max\_->RemoveDummyParent();

}

void Next() override {

if (this->iter\_->getParent()->getRight() && this->iter\_->getParent()->getRight() != this->iter\_) {

this->iter\_ = this->iter\_->getParent()->getRight();

while (this->iter\_->getLeft() || this->iter\_->getRight()) {

if (this->iter\_->getLeft()) {

this->iter\_ = this->iter\_->getLeft();

}

else

this->iter\_ = this->iter\_->getRight();

}

}

else {

this->iter\_ = this->iter\_->getParent();

}

}

};

Strategy:

template <typename T>

class TreeIterStrategy{

public:

virtual ~TreeIterStrategy() = default;

virtual std::string doTraverse(Node<T>\* root){

return "";

}

};

template <typename T>

class InOrderStrategy : public TreeIterStrategy<T>{

public:

std::string doTraverse(Node<T>\* root) override{

…

}

};

template <typename T>

class PreOrderStrategy : public TreeIterStrategy<T> {

std::string doTraverse(Node<T>\* root) override {

…

}

};

template <typename T>

class PostOrderStrategy : public TreeIterStrategy<T> {

std::string doTraverse(Node<T>\* root) override {

…

}

};

template <typename T>

class Traverser{

private:

std::unique\_ptr<TreeIterStrategy<T>> strategy\_;

public:

Traverser(){}

Traverser(TravType type){

this->set\_strategy(type);

}

void set\_strategy(TravType type){

switch(type){

case InOrder:

{strategy\_ = std::make\_unique<InOrderStrategy<T>>();break;}

case PreOrder:

{strategy\_ = std::make\_unique<PreOrderStrategy<T>>();break;}

case PostOrder:

{strategy\_ = std::make\_unique<PostOrderStrategy<T>>();break;}

}

}

std::string Iterate(Node<T>\* root){

if(strategy\_){

return strategy\_->doTraverse(root);

}

else{

return "No strategy set\n";

}

}

};

Template Method:

template <typename KeyType>

class TreeAnalyzer {

protected:

Node<KeyType>\* root\_;

public:

TreeAnalyzer(){}

virtual ~TreeAnalyzer(){}

TreeAnalyzer(Node<KeyType>\* root) : root\_(root){}

std::string Analyze() {

if (!root\_)

{

return "The tree is not set or is null\n";

}

std::string ret = "";

int h = this->findHeight();

ret += "Height: " + std::to\_string(h) + "\n";

std::string min = this->findMin();

ret += min + "\n";

std::string max = this->findMax();

ret += max;

std::string lword = this->findLongestWord();

if (lword != "")

ret += "\n" + lword;

std::string sum = this->findSumOfAllNodes();

if (sum != "")

ret += "\n" + sum;

return ret;

}

int findHeight() {

return root\_->getHeight();

}

virtual std::string findMin(){

return "Least element: ";

}

virtual std::string findMax() {

return "Greatest element: ";

}

virtual std::string findLongestWord() { return ""; }

virtual std::string findSumOfAllNodes() { return ""; }

};

template <typename Number>

class NumberTreeAnalyzer : public TreeAnalyzer<Number> {

public:

NumberTreeAnalyzer(Node<Number>\* root) : TreeAnalyzer<Number>(root) {}

std::string findMin() override {

return TreeAnalyzer<Number>::findMin() + std::to\_string(this->root\_->getMin()->getKey());

}

std::string findMax() override {

return TreeAnalyzer<Number>::findMax() + std::to\_string(this->root\_->getMax()->getKey());

}

std::string findSumOfAllNodes() override {

return "Sum of all keys in the tree: " + std::to\_string(this->root\_->getSumOfAllNodes());

}

};

class StringTreeAnalyzer : public TreeAnalyzer<std::string> {

public:

StringTreeAnalyzer(Node<std::string>\* root) : TreeAnalyzer(root) {}

std::string findMin() override {

return TreeAnalyzer::findMin() + this->root\_->getMin()->getKey();

}

std::string findMax() override {

return TreeAnalyzer::findMax() + this->root\_->getMax()->getKey();

}

std::string findLongestWord() override {

std::string theword = this->root\_->getLongestWord();

return "Longest word: " + theword + ", of length: " + std::to\_string(theword.length());

}

};

Command/Memento

class Snapshot {

private:

StringedNode\* root\_;

std::string tt\_;

std::string descr\_;

std::string date\_;

public:

template<typename T>

Snapshot(Node<T>\* root, TypeNode type, TreeType tt, std::string descr) : descr\_(descr) {

TNodeAdapter<T>\* adapter = new TNodeAdapter<T>(root, type);

this->root\_ = adapter->getStrNode();

delete adapter;

std::time\_t now = std::time(0);

switch (tt) {

case BinSearchTree:

tt\_ = "BST";

default:

break;

}

this->date\_ = std::ctime(&now)/\*.substr(0, 19)\*/;

};

template<typename T>

Node<T>\* getTree() {

StringedNodeAdapter<T>\* adapter = new StringedNodeAdapter<T>(this->root\_);

Node<T>\* n = adapter->getNode();

delete adapter;

return n;

}

StringedNode\* getStrNode() {

return this->root\_;

}

TreeType getTreeType() {

if (tt\_ == "BST")

return BinSearchTree;

}

std::string getTreeTypeStr() {

return tt\_;

}

std::string getName() {

return "(" + this->date\_ + ") " + this->descr\_;

}

std::string getDate() {

return this->date\_;

}

};

template<typename T>

class Command {

protected:

std::vector<Snapshot\*> snaps;

TreeFacade<T>\* fac;

//StringedNode\* root;

public:

Command() {}

Command(TreeFacade<T>\* facc) : fac(facc) {};

virtual void execute() {};

void makeBackup() {

if (this->snaps.size() <= 50)

this->snaps.push\_back(fac->getSnapshot());

}

std::vector<Snapshot\*> getSnaps() {

return snaps;

}

void setSnaps(std::vector<Snapshot\*> s) {

this->snaps = s;

}

};

template<typename T>

class SaveCommand : public Command<T> {

public:

SaveCommand() : Command<T>() {};

SaveCommand(TreeFacade<T>\* facc) : Command<T>(facc) {};

void execute() override {

std::ofstream output("tree.txt");

Snapshot\* snap = this->snaps.back();

StringedNode\* sn = snap->getStrNode();

output << sn->getTypeStr() << "\n" << snap->getTreeTypeStr() << "\n";

Node<std::string>\* n = sn->getTree();

Traverser<std::string>\* trav = new Traverser<std::string>(PreOrder);

std::string out = trav->Iterate(n);

output << out;

output.close();

}

};

template<typename T>

class UndoCommand : public Command<T> {

public:

UndoCommand() : Command<T>() {};

UndoCommand(TreeFacade<T>\* facc) : Command<T>(facc) {};

void execute() override {

if (!(this->snaps.size() - 1))

return;

this->snaps.pop\_back();

Snapshot\* snap = this->snaps.back();

this->fac->restore(snap);

}

};