

Языки программирования

Лекция 7

ПМИ Семестр 2

Демяненко Я.М.

2024

Классы для рекурсивных типов данных

Рекурсивное определение данных возникает, когда структура данных ссылается на объект такой же структуры.

При этом в случае **одной ссылки** чаще всего в алгоритмах обработки эффективней использовать **итерацию**.

Если же **ссылок больше** одной (как в **деревьях**), то в большинстве случаев применяются **рекурсивные** реализации.

Рекурсивное определение бинарного дерева

- Первый способ определения класса бинарного дерева состоит в том, чтобы определить в нём **указатели на два поддеревя** и **открытые рекурсивные функции** для работы с деревом.
- Второй способ. Рекурсивное определение бинарного дерева можно реализовать как класс, содержащий внутри себя **указатель на узел дерева** (корень) и **рекурсивное описание узла дерева**.

При этом **открытые функции-члены** класса **не будут рекурсивными**.

Пример. Реализовать класс бинарного дерева поиска, определив в нём
указатели на два поддеревя и
открытые рекурсивные функции для работы с деревом:

конструктор,
конструктор копии,
деструктор,
добавление элемента в дерево и
вывод всех элементов дерева в поток.

```
class TreeR{
private:
    int data;
    TreeR *lt;
    TreeR *rt;

public:
    TreeR (int val=0, TreeR *l = nullptr, TreeR *r = nullptr):
        data(val), lt(l), rt(r) {}

    TreeR(const TreeR * t){
        if (t->lt) lt = new TreeR(t->lt);
        data = t->data;
        if (t->rt) rt = new TreeR(t->rt);
    }

    ~TreeR(){
        if (lt != nullptr) delete lt;
        if (rt != nullptr) delete rt;
    }
}
```

```

void add(int a){
    if (data > a){
        if (lt) lt->add(a);
        else lt = new TreeR(a);
    }
    else{
        if (rt) rt->add(a);
        else rt = new TreeR(a);
    }
}

```

```

friend ostream& operator<<(ostream& os, const TreeR *t){
    if (t->lt) os << t->lt;
    os << t->data << " ";
    if (t->rt) os<< t->rt;
    return os;
}

void printRKL(){
    if (rt) rt->printRKL();
    cout << " " << data << " ";
    if (lt) lt->printRKL();
}
};

```

```

if (rt) rt->printRKL();
ЭКВИВАЛЕНТНО
if (this->rt) this->rt->printRKL();

```

Особенности первого подхода

TreeR *t= new TreeR(); // это непустое дерево

```
t->add(5);  
t->add(-7);  
t->add(3);  
t->add(-4);  
cout <<"T " << t << endl;  
t->printRKL();  
TreeR *t1 = new TreeR(t);  
cout << "T1 " <<t1 << endl;
```

Для класса TreeR пустота дерева определяется не на уровне класса, а на уровне указателя на объект класса.

TreeR* t0= nullptr; // это пустое дерево

Особенности первого подхода: проверка на пустоту

```
TreeR* t0= nullptr; // это пустое дерево
```

Поэтому нельзя реализовать функцию-член класса для проверки на пустоту.

Для безопасной работы перед вызовом функций-членов класса следует выполнять проверку на пустоту следующим образом:

```
if (t0!=nullptr) t0->add(3);  
if (t0!=nullptr) t0->RKL();  
if (t0!=nullptr) cout <<"T " << t0 << endl;
```

Особенности первого подхода: удаление

```
~TreeR(){  
    if (lt != nullptr)  
        delete lt;  
    if (rt != nullptr)  
        delete rt;  
}
```

При использовании операции удаления отдельных узлов из дерева возникает проблема при удалении последней вершины, которая не может быть решена в самой операции.

Пример. Реализовать класс бинарного дерева поиска, содержащий внутри себя указатель на корень дерева и рекурсивное описание узла дерева.

Определить **открытые функции-члены класса:**

- конструктор,
- конструктор копии,
- деструктор,
- добавление элемента в дерево и
- вывод всех элементов в поток.

При их реализации **использовать рекурсивные функции в закрытой** части описания класса.

```
class Tree{
private:
    struct TNode;
    typedef TNode* node_ptr;

    struct TNode{
        int data;
        node_ptr lt, rt;
        TNode (int val, node_ptr l=nullptr, node_ptr r=nullptr):
            data(val), lt(l), rt(r){}
    };

    node_ptr root;

    void delTree(node_ptr t){
        if (t!=nullptr){
            delTree(t->lt);
            delTree(t->rt);
            delete t;
        }
    }
}
```

Продолжение: закрытые функции

```
void add(node_ptr& t, int a){
    if (t == nullptr)
        t = new TNode(a);
    else if (t->data > a)
        add(t->lt, a);
    else
        add(t->rt, a);
}
```

```
void printLKR(node_ptr t, ostream& os) const{
    if (t){
        printLKR(t->lt, os);
        os << t->data << " ";
        printLKR(t->rt, os);
    }
}
```

```
void copy(node_ptr t, node_ptr &newT) const {
    if (t != nullptr){
        newT = new TNode(t->data, 0, 0);
        copy(t->lt, newT->lt);
        copy(t->rt, newT->rt);
    }
    else newT = nullptr;
}
```

Продолжение: открытые функции

public:

```
Tree(): root(nullptr) {}  
Tree(const Tree& t){  
    copy( t.root, root);  
}
```

```
~Tree(){  
    delTree (root);  
}
```

```
void addNode(int a){  
    add(root, a);  
}
```

```
friend ostream& operator<<(ostream& os, const Tree &t){  
    t.printLKR(t.root,os);  
    return os;  
}  
};
```

Особенности второго подхода

В примере типы `TNode` (узел) и `node_ptr` (указатель на узел) определены внутри класса, что подчёркивает **инкапсуляцию внутренней** организации класса `Tree`.

Именно в определении типа **TNode** присутствует **рекурсия**.

Поэтому **рекурсивные функции** оперируют с указателями на узел дерева и должны быть объявлены как **private**.

Для доступа к ним используются **открытые функции** класса, которые **вызывают рекурсивные функции**, передавая в качестве параметра указатель на корень дерева.

Особенности второго подхода: пустота и деструктор

Такое описание класса допускает существование пустого дерева.

Конструктор без параметров создаёт пустое дерево.

Если предусмотреть операцию удаления узла дерева, то можно получить в процессе её выполнения пустое дерево.

Для безопасной работы с таким деревом рекомендуется иметь операцию проверки дерева на пустоту.

```
Tree t;  
t.addNode(5);  
t.addNode(7);  
t.addNode(3);  
t.addNode(4);  
cout <<"T " << t << endl;  
Tree t1(t);  
cout << "T1 " <<t1 << endl;
```

Рекомендации

Реализация класса бинарного дерева вторым способом лишена недостатков, перечисленных для первого способа.

Именно

второй способ реализации рекомендуется использовать при создании классов для рекурсивных структур.

Действие передаваемое параметром (callback)

```
// Указатель на функцию с таким  
прототипом  
typedef double (*BinOp) (double, double);
```

```
BinOp bop = &add;  
(*bop)(3, 5);
```

```
// Или как и описание переменной  
double (*op)(double, double)
```

```
op = mult;  
op(3, 5);
```

```
template <typename T>  
// action — переменная типа "указатель на функцию"  
void for_each(node<T>* p, void (*action)(T&)) {  
    while(p) {  
        action(p -> data);  
        p = p -> next;  
    }  
}
```

```
void print(int &x) { cout << x << ' '; }
```

```
void inc(int &x) { x++; }
```

```
for_each(pn, print);  
for_each(pn, inc);  
for_each(pn, print);
```