Коллекции и рекурсия
Бинарные деревья

Описание (из прошлой лекции)

```
struct el{
  int item;
  el* next;
};
```

```
class List{
  public:
   . . .
  private:
      el* head;
      el* tail;
     . .
  };
```

Метод печати в обратном порядке

- Нельзя сделать метод рекурсивным, т.к. List
  не является рекурсивной структурой
  данных
- Рекурсия заложена в структуре el
- Поэтому организовать рекурсию можно
  относительно поля head, но это - приватные
  данные

- Создадим два метода: открытый нерекурсивный, который будет вызывать закрытый рекурсивный

```
public:
void revPrint();

private:
void revP(el * f);
```

```
void List::revPrint()
{
        revP(head);
}
void List::revP(el * f)
{
   if (f)  {
         revP(f->next);
         cout<<f->item<<"  ";
     }
}
```

- Как обойтись без методов-оберток?
- Нужно, чтобы класс представлял рекурсивную структуру

```
class ListNew{
 public:
   . . .
 private:
  int   item;
  ListNew * next;
   . . .
};
```

```
void ListNew :: revPrintNew( )
{
  if (next)
        next->revPrintNew();
  cout<< "  " <<item<<"  " ;
}
```

Проблема

- Что является пустым списком?

- Одно из решений – заглавный элемент

Деревья

```
class  TreeR {
 private:
    int data;
    TreeR *lt,  *rt;
 public :
     TreeR (int a=0) : data(a), lt(0), rt(0) {}
    ~TreeR();
     void add (int a);
     void printLKR();
};
```

Реализация

```
void TreeR:: add(int a)  // на примере дерева поиска
{
 if (data > a) {
   if (lt) lt->add(a);
   else lt = new TreeR(a);
 }
 else {
   if (rt) rt->add(a);
   else rt = new TreeR(a);
 }
}
```

Реализация

```
void TreeR:: printLKR()
 {
    if (lt) lt->LKR();
   cout<<"  "<<data<<"                    ";
    if (rt) rt->LKR();
}
```

Реализация

```
TreeR:: ~TreeR()
{
   if (lt) delete lt;
   if (rt) delete rt;
}
```

Самостоятельно

- Конструктор копирования
- Сравнение на равенство двух деревьев
- Операцию присваивания
- Операцию сложения двух деревьев

Работа с деревом

```
TreeR* t1 ; // это пустое дерево
  t1= new TreeR(5);
```

- если дерево не пусто, в нем есть корневой узел
- нельзя реализовать метод проверки на пустоту

Работа с деревом

- поэтому вызовы методов нужно предварять проверкой

```
if (t1) t1->add(3);
if (t1)  t1->LKR();
if (t1)  delete t1 ;
```

- или всегда создавать дерево при объявлении указателя

TreeR* t1= new TreeR(5);

- это может создать алгоритмические проблемы
- кроме того может возникнуть трудность при удалении из дерева последней вершины

Двоичное дерево поиска

Реализация класса

```
class Node;
typedef Node* pnode;

class  Node {
    int data;
    pnode  lt, rt;
    Node () {}
    Node (int a, pnode t1=0, pnode t2 =0):
                    data(a),lt(t1),rt(t2) {}

friend class BSTree;
};
```

- Все рекурсивные методы делаем закрытыми
- Для их вызова простые открытые методы

```
class BSTree {

private:
    pnode root;
    void printPref (pnode &t);
    void copy(pnode t, pnode &newT) const;
    void destroy (pnode & t);
    void addEl (pnode & t, int a);
    void deleteEl (pnode & t, int a);
    void deleteNode (pnode &t);
    void delLeftLeaf(pnode &t,int &repl);
```

```
public:
    BSTree() {root=0;}
    BSTree(const BSTree& tree);
    ~BSTree();
    void printPref();
    bool isEmpty() const;
    void addEl(int val);
    void deleteEl (int val);
};
```

Открытые методы

```
BSTree :: BSTree(const BSTree& tree)
{
    copy (tree.root, root);
}
BSTree :: ~BSTree()
{
    destroy(root);
}
void BSTree ::printPref()
{
    printPref(root);
}
```

Открытые методы

```
bool BSTree ::isEmpty() const
{
    return root==0;
}
void BSTree ::addEl(int val)
{
    addEl (root,val);
}
void BSTree ::deleteEl(int val)
{
    deleteEl(root,val);
}
```

Рекурсивные методы

```
void BSTree ::addEl(pnode &t, int a)
{
   if (t==0)
        t = new Node (a, 0, 0);
   else
        if (a < t->data)
                        addEl(t->lt,a);
        else addEl(t->rt,a);
}
```

```
void BSTree :: printPref (pnode &t)            //LKR
{
    if (t!=0)
    {
        printPref(t->lt);
        cout<< t->data<<"   ";
        printPref(t->rt);
    }
}
```

```
void BSTree :: copy(pnode t, pnode &newT) const  //KLR
{
    if ( t!=0)
    {
        newT = new Node(t->data,0,0);
        copy(t->lt,newT->lt);
        copy(t->rt,newT->rt);
    }
    else newT=0;
}
```

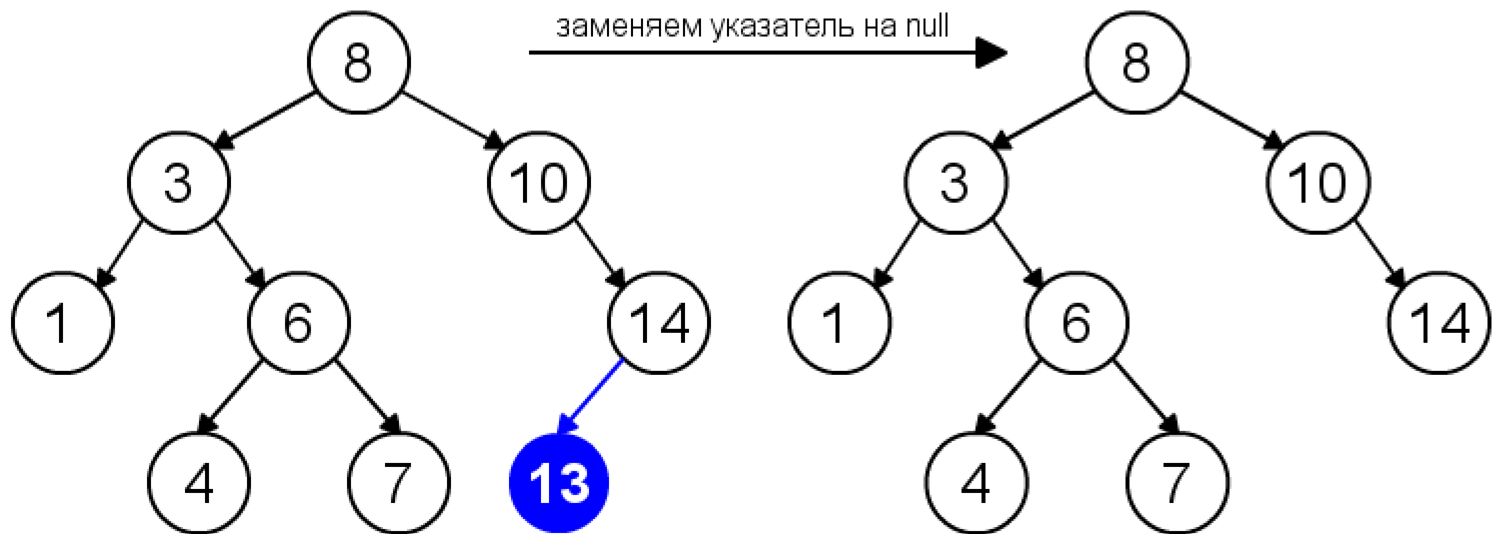
```
void BSTree :: destroy(pnode &t)              //LRK
{
    if (t!=0)
    {
        destroy(t->lt);
        destroy(t->rt);
        delete t;
        t=0;
    }
}
```

Операция удаления

- Рассмотрим три случая

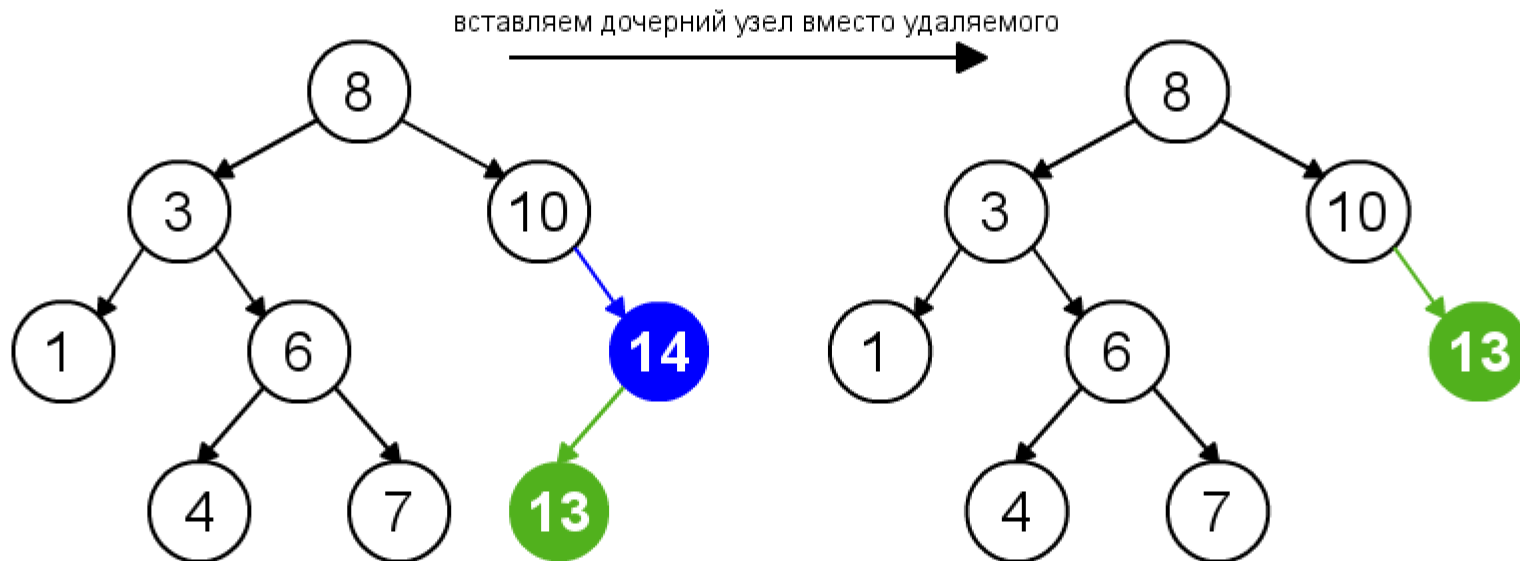
1. узел является листом
2. узел имеет только одно поддерево
3. у узла два поддерева

1 случай



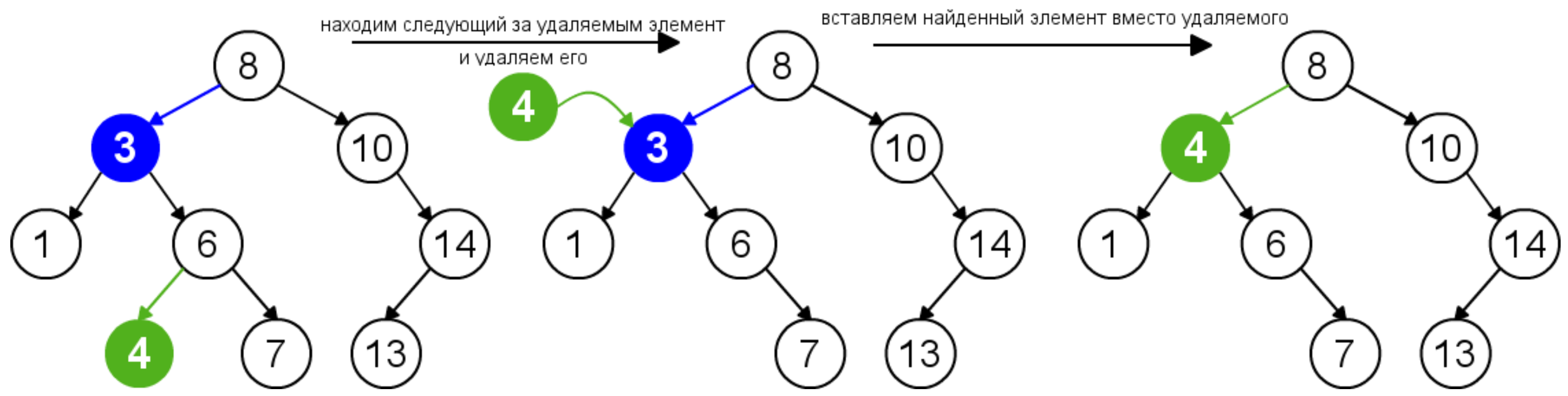
2 случай

- одинаково для левого и правого поддерева



3 случай

- Находим самый левый элемент в правом поддереве
- Его значение вставляем вместо удаляемого узла
- А сам этот узел удаляем



находим следующий за удаляемым элемент
и удаляем его
вставляем найденный элемент вместо удаляемого
8
3
10
1
6
14
4
7
13

Три метода, реализующие удаление

- Найти узел по значению, которое нужно удалить и вызвать для этого узла метод исключающий его из дерева

```
void BSTree :: deleteEl(pnode &t, int a)
{
   if (t!=0)
   if (a == t->data)
         deleteNode(t); // узел найден - удаляем
   else
         if (a<t->data)
                  deleteEl(t->lt,a); // рекурсия
         else     deleteEl(t->rt,a); // рекурсия
}
```

- Удаление найденного узла (работает в корректных условиях, т.к. вызывается, когда узел найден)

```
void BSTree :: deleteNode (pnode &t)
{       // 1. корень является листом
        // 2. у корня нет левого поддерева
        // 3. у корня нет правого поддерева
        // 4. корень имеет два поддерева

        pnode delNode;
        int repl;
        if ((t->lt ==0) && (t->rt==0)) // 1
        {
                delete t;
                t=0;
        }
```

```
else
     if (t->lt ==0)                    // 2
     {
            delNode = t;
            t = t->rt;
            delNode->rt = 0;
            delete delNode;
     }
```

```
else if (t->rt ==0)                    // 3
{
        delNode = t;
        t = t->lt;
        delNode->lt = 0;
        delete delNode;
}
```

В 4 случае вызываем вспомогательный метод, который ищет самый левый узел в правом поддереве. запоминаем его значение, а сам узел - удаляем

```
    else                    // 4
    {
        delLeftLeaf (t->rt, repl);
        t->data = repl;
    }
}
```

```
void BSTree::delLeftLeaf(pnode &t,int &repl)
{
	if (t->lt == 0) // у самого левого нет левых
				//потомков
	{
		repl = t->data;
		pnode delNode = t;
		t = t->rt;
		delNode->rt=0;
		delete delNode;
	}
	else	delLeftLeaf (t->lt,repl);
}
```

Эффективность операций

- Максимальное количество сравнений, необходимых для выполнения операций поиска, вставки и удаления, равно высоте бинарного дерева поиска
- Для дерева из N узлов возможные значения высоты h

$$[\ \log_2(N+1)\ ] <= h <= N$$

- Бинарное дерево наименьшей высоты называется совершенным или сбалансированным
- Чтобы обеспечить эффективность операций нужно стремиться поддерживать структуру близкую к сбалансированности
- Существуют специальные алгоритмы балансировки