***Реализация списка:***

**С++:**

struct list \* init(int a) // а- значение первого узла

{

  struct list \*lst;

  // выделение памяти под корень списка

  lst = (struct list\*)malloc(sizeof(struct list));

  lst->field = a;

  lst->ptr = NULL; // это последний узел списка

  return(lst);

}

Функция добавления узла в ОЗС:

struct list \* addelem(list \*lst, int number)

{

  struct list \*temp, \*p;

  temp = (struct list\*)malloc(sizeof(list));

  p = lst->ptr; // сохранение указателя на следующий узел

  lst->ptr = temp; // предыдущий узел указывает на создаваемый

  temp->field = number; // сохранение поля данных добавляемого узла

  temp->ptr = p; // созданный узел указывает на следующий элемент

  return(temp);

}

Удаление узла в ОЗС:

struct list \* deletelem(list \*lst, list \*root)

{

  struct list \*temp;

  temp = root;

  while (temp->ptr != lst) // просматриваем список начиная с корня

  { // пока не найдем узел, предшествующий lst

    temp = temp->ptr;

  }

  temp->ptr = lst->ptr; // переставляем указатель

  free(lst); // освобождаем память удаляемого узла

  return(temp);

}

Удаление корня:

struct list \* deletehead(list \*root)

{

  struct list \*temp;

  temp = root->ptr;

  free(root); // освобождение памяти текущего корня

  return(temp); // новый корень списка

}

Вывод элементов:

void listprint(list \*lst)

{

  struct list \*p;

  p = lst;

  do {

    printf("%d ", p->field); // вывод значения элемента p

    p = p->ptr; // переход к следующему узлу

  } while (p != NULL);

}

Взаимообмен узлов ОЗС:

struct list \* swap(struct list \*lst1, struct list \*lst2, struct list \*head)

{

  // Возвращает новый корень списка

  struct list \*prev1, \*prev2, \*next1, \*next2;

  prev1 = head;

  prev2 = head;

  if (prev1 == lst1)

    prev1 = NULL;

  else

    while (prev1->ptr != lst1) // поиск узла предшествующего lst1

      prev1 = prev1->ptr;

  if (prev2 == lst2)

    prev2 = NULL;

  else

    while (prev2->ptr != lst2) // поиск узла предшествующего lst2

      prev2 = prev2->ptr;

  next1 = lst1->ptr;  // узел следующий за lst1

  next2 = lst2->ptr;  // узел следующий за lst2

  if (lst2 == next1)

  {                       // обмениваются соседние узлы

    lst2->ptr = lst1;

    lst1->ptr = next2;

    if (lst1 != head)

      prev1->ptr = lst2;

  }

  else

    if (lst1 == next2)

    {

      // обмениваются соседние узлы

      lst1->ptr = lst2;

      lst2->ptr = next1;

      if (lst2 != head)

        prev2->ptr = lst2;

    }

    else

    {

      // обмениваются отстоящие узлы

      if (lst1 != head)

        prev1->ptr = lst2;

      lst2->ptr = next1;

      if (lst2 != head)

        prev2->ptr = lst1;

      lst1->ptr = next2;

    }

    if (lst1 == head)

      return(lst2);

    if (lst2 == head)

      return(lst1);

    return(head);

}

**Java:**

class ListElement {

    ListElement next;    // указатель на следующий элемент

    int data;            // данные

}

class List {

    private ListElement head;       // указатель на первый элемент

    private ListElement tail;       // указатель последний элемент

    void addFront(int data)           //добавить спереди

    {

        ListElement a = new ListElement();  //создаём новый элемент

        a.data = data;              //инициализируем данные.

                                    // указатель на следующий элемент автоматически инициализируется как null

        if(head == null)            //если список пуст

        {                           //то указываем ссылки начала и конца на новый элемент

            head = a;               //т.е. список теперь состоит из одного элемента

            tail = a;

        }

        else {

            a.next = head;          //иначе новый элемент теперь ссылается на "бывший" первый

            head = a;               //а указатель на первый элемент теперь ссылается на новый элемент

        }

    }

    void addBack(int data) {          //добавление в конец списка

        ListElement a = new ListElement();  //создаём новый элемент

        a.data = data;

        if (tail == null)           //если список пуст

        {                           //то указываем ссылки начала и конца на новый элемент

            head = a;               //т.е. список теперь состоит из одного элемента

            tail = a;

        } else {

            tail.next = a;          //иначе "старый" последний элемент теперь ссылается на новый

            tail = a;               //а в указатель на последний элемент записываем адрес нового элемента

        }

    }

    void printList()                //печать списка

    {

        ListElement t = head;       //получаем ссылку на первый элемент

        while (t != null)           //пока элемент существуе

        {

            System.out.print(t.data + " "); //печатаем его данные

            t = t.next;                     //и переключаемся на следующий

        }

    }

    void delEl(int data)          //удаление элемента

    {

        if(head == null)        //если список пуст -

            return;             //ничего не делаем

        if (head == tail) {     //если список состоит из одного элемента

            head = null;        //очищаем указатели начала и конца

            tail = null;

            return;             //и выходим

        }

        if (head.data == data) {    //если первый элемент - тот, что нам нужен

            head = head.next;       //переключаем указатель начала на второй элемент

            return;                 //и выходим

        }

        ListElement t = head;       //иначе начинаем искать

        while (t.next != null) {    //пока следующий элемент существует

            if (t.next.data == data) {  //проверяем следующий элемент

                if(tail == t.next)      //если он последний

                {

                    tail = t;           //то переключаем указатель на последний элемент на текущий

                }

                t.next = t.next.next;   //найденный элемент выкидываем

                return;                 //и выходим

            }

            t = t.next;                //иначе ищем дальше

        }

    }

}

**Python:**

class Node:

    def \_\_init\_\_(self, value = None, next = None):

        self.value = value

        self.next = next

Для определения связного списка потребуется еще один класс – **LinkedList**, в конструкторе которого будут определяться первый и последний элементы списка и его длина. Также в классе будут использоваться встроенный метод **str** для распечатки содержимого списка и метод **clear** для очистки списка.

import copy

import random

class LinkedList:

    def \_\_init\_\_(self):

        self.first = None

        self.last = None

        self.length = 0

    def \_\_str\_\_(self):

        if self.first != None:

            current = self.first

            out = 'LinkedList [\n' +str(current.value) +'\n'

            while current.next != None:

                current = current.next

                out += str(current.value) + '\n'

            return out + ']'

        return 'LinkedList []'

    def clear(self):

        self.\_\_init\_\_()

Функция для определения длины списка:

def Len(self):

    self.length =0

    if self.first != None:

        self.length +=1

        current = self.first

        while current.next != None:

            current = current.next

            self.length +=1

    return self.length

Добавление элементов в начало списка:

def Push(self, x):

    if self.first == None:

        self.first = Node(x,None)

        self.last = self.first

    else:

        self.first = Node(x,self.first)

Добавление элементов в конец списка:

def add(self, x):

    if self.first == None:

        self.first = Node(x, None)

        self.last = self.first

    elif self.last == self.first:

        self.last = Node(x, None)

        self.first.next = self.last

    else:

        current = Node(x, None)

        self.last.next = current

        self.last = current

Вставка элемента в список:

def InsertNth(self,i,x):

    if (self.first == None):

        self.first = Node(x,self.first)

        self.last = self.first.next

        return

    if i == 0:

      self.first = Node(x,self.first)

      return

    curr=self.first

    count = 0

    while curr != None:

        if count == i-1:

          curr.next = Node(x,curr.next)

          if curr.next.next == None:

            self.last = curr.next

          break

        curr = curr.next

Удаление головы:

def Pop(self):

    oldhead=self.first

    if oldhead==None:

        return None

    self.first=oldhead.next

    if self.first==None:

        self.last=None

    return oldhead.value

Удаление элемента из списка:

def Del(self,i):

    if (self.first == None):

      return

    old = curr = self.first

    count = 0

    if i == 0:

      self.first = self.first.next

      return

    while curr != None:

        if count == i:

          if curr.next == self.last:

            self.last = curr

            break

          else:

            old.next = curr.next

          break

        old = curr

        curr = curr.next

        count += 1

Вставка элемента в отсортированный список:

def SortedInsert(self,x):

    if (self.first == None):

      self.first = Node(x,self.last)

      return

    if self.first.value > x:

      self.first = Node(x,self.first)

      return

    old = curr = self.first

    while curr != None:

        if curr.value > x:

          curr = Node(x,curr)

          old.next = curr

          return

        old = curr

        curr = curr.next

    curr = Node(x,None)

    old.next = curr

Удаление повторяющихся значений:

def RemoveDuplicates(self):

    if (self.first == None):

        return

    old = curr = self.first

    while curr != None:

        \_del = 0

        if curr.next != None:

            if curr.value == curr.next.value:

              curr.next = curr.next.next

              \_del = 1

        if \_del == 0:

          curr = curr.next

В Python для копирования списков можно использовать стандартный модуль **copy**, как показано ниже.

import copy

L2 = copy.deepcopy(L) // создание копии списка L

***Реализация дерева:***

**С++:**

Узел дерева можно описать как структуру:

struct tnode {

  int field;           // поле данных

  struct tnode \*left;  // левый потомок

  struct tnode \*right; // правый потомок

};

При этом обход дерева в префиксной форме будет иметь вид

void treeprint(tnode \*tree) {

  if (tree!=NULL) { //Пока не встретится пустой узел

    cout << tree->field; //Отображаем корень дерева

    treeprint(tree->left); //Рекурсивная функция для левого поддерева

    treeprint(tree->right); //Рекурсивная функция для правого поддерева

  }

}

Обход дерева в инфиксной форме будет иметь вид

void treeprint(tnode \*tree) {

  if (tree!=NULL) { //Пока не встретится пустой узел

    treeprint(tree->left); //Рекурсивная функция для левого поддерева

    cout << tree->field; //Отображаем корень дерева

    treeprint(tree->right); //Рекурсивная функция для правого поддерева

  }

}

Обход дерева в постфиксной форме будет иметь вид

void treeprint(tnode \*tree) {

  if (tree!=NULL) { //Пока не встретится пустой узел

    treeprint(tree->left); //Рекурсивная функция для левого поддерева

    treeprint(tree->right); //Рекурсивная функция для правого поддерева

    cout << tree->field; //Отображаем корень дерева

  }

}

Узел дерева можно описать как структуру:

struct tnode {

  int field;           // поле данных

  struct tnode \*left;  // левый потомок

  struct tnode \*right; // правый потомок

};

При этом обход дерева в префиксной форме будет иметь вид

void treeprint(tnode \*tree) {

  if (tree!=NULL) { //Пока не встретится пустой узел

    cout << tree->field; //Отображаем корень дерева

    treeprint(tree->left); //Рекурсивная функция для левого поддерева

    treeprint(tree->right); //Рекурсивная функция для правого поддерева

  }

}

Обход дерева в инфиксной форме будет иметь вид

void treeprint(tnode \*tree) {

  if (tree!=NULL) { //Пока не встретится пустой узел

    treeprint(tree->left); //Рекурсивная функция для левого поддерева

    cout << tree->field; //Отображаем корень дерева

    treeprint(tree->right); //Рекурсивная функция для правого поддерева

  }

}

Обход дерева в постфиксной форме будет иметь вид

void treeprint(tnode \*tree) {

  if (tree!=NULL) { //Пока не встретится пустой узел

    treeprint(tree->left); //Рекурсивная функция для левого поддерева

    treeprint(tree->right); //Рекурсивная функция для правого поддерева

    cout << tree->field; //Отображаем корень дерева

  }

}

Добавление узлов в дерево

struct tnode \* addnode(int x, tnode \*tree) {

  if (tree == NULL) { // Если дерева нет, то формируем корень

    tree =new tnode; // память под узел

    tree->field = x;   // поле данных

    tree->left =  NULL;

    tree->right = NULL; // ветви инициализируем пустотой

  }else  if (x < tree->field)   // условие добавление левого потомка

    tree->left = addnode(x,tree->left);

  else    // условие добавление правого потомка

    tree->right = addnode(x,tree->right);

  return(tree);

}

Удаление поддерева

void freemem(tnode \*tree) {

  if(tree!=NULL) {

    freemem(tree->left);

    freemem(tree->right);

    delete tree;

  }

}

**Java:**

private static class Node<V extends Comparable<V>> {

private Node parent;

private Node left;

private Node right;

private int k = 0;

private final V data;

public Node(V data) {

this.data = data;

this.parent = null;

this.left = null;

this.right = null;

}

}

public abstract class HashTree<E extends Comparable<E>> {

private Node root = null;

private Node[] nodes;

public HashTree(int capacity) {

this.nodes = new Node[capacity];

}

public abstract int getElementHash(E element);

…

}

Добавление узла:

public void addElement(E element) {

int index = getElementHash(element);

if (nodes[index] != null) {

return;

}

Node<E> node = new Node<>(element);

nodes[index] = node;

this.root = connectNodes(this.root, node);

}

Удаление узла:

public E removeElement(E element) {

int index = getElementHash(element);

Node node = nodes[index];

if (node == null) {

return null;

}

nodes[index] = null;

E data = (E) node.data;

Node l = getElemInArray(node.left);

Node r = getElemInArray(node.right);

if (l != null) {

l.parent = null;

}

if (r != null) {

r.parent = null;

}

l = connectNodes(l, r);

if (node.parent == null) {

this.root = l;

if (this.root != null) {

this.root.parent = null;

}

return data;

}

int p = getElementHash((E) node.parent.data);

if (nodes[p] != null) {*//здесь сравниваются ИМЕННО значение указателей,*

*//интересует равенство адресов памяти, а не значений*

if (nodes[p].left == node) {

nodes[p].left = null;

}

if (nodes[p].right == node) {

nodes[p].right = null;

}

}

connectNodes(nodes[p], l);

return data;

}

Присоединение узла или поддерева:

private Node connectNodes(Node parent, Node node) {

if (node == null) {

return parent;

}

if (parent == null) {

return node;

} else {

if (compare(node, parent) < 0) {

return connectNodes(node, parent);

}

Node cur = parent;

Node n = node;

while (cur != null) {

if (cur.left == null) {

cur.left = n;

n.parent = cur;

cur.k++;

break;

}

if (cur.right == null) {

if (compare(n, cur.left) <= 0) {

cur.right = cur.left;

cur.left = n;

n.parent = cur;

cur.k++;

break;

} else {

cur.right = n;

n.parent = cur;

cur.k++;

break;

}

}

if (compare(n, cur.left) <= 0) {

Node tmp = cur.left;

cur.left = n;

n.parent = cur;

cur.k++;

cur = n;

n = tmp;

continue;

}

if (compare(n, cur.right) < 0

&& compare(n, cur.left) > 0) {

cur.k++;

if (cur.right.k < cur.left.k) {

Node tmp = cur.right;

cur.right = n;

n.parent = cur;

cur = n;

n = tmp;

} else {

cur = cur.left;

}

continue;

}

if (compare(n, cur.left) > 0) {

cur.k++;

cur = cur.left.k < cur.right.k ? cur.left : cur.right;

}

}

return parent;

}

**Python:**

class TreeNode:

def \_\_init\_\_(self,key,val,left=None,right=None,parent=None):

self.key = key

self.payload = val

self.leftChild = left

self.rightChild = right

self.parent = parent

def hasLeftChild(self):

return self.leftChild

def hasRightChild(self):

return self.rightChild

def isLeftChild(self):

return self.parent and self.parent.leftChild == self

def isRightChild(self):

return self.parent and self.parent.rightChild == self

def isRoot(self):

return not self.parent

def isLeaf(self):

return not (self.rightChild or self.leftChild)

def hasAnyChildren(self):

return self.rightChild or self.leftChild

def hasBothChildren(self):

return self.rightChild and self.leftChild

def replaceNodeData(self,key,value,lc,rc):

self.key = key

self.payload = value

self.leftChild = lc

self.rightChild = rc

if self.hasLeftChild():

self.leftChild.parent = self

if self.hasRightChild():

self.rightChild.parent = self

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

self.size = 0

def length(self):

return self.size

def \_\_len\_\_(self):

return self.size

def put(self,key,val):

if self.root:

self.\_put(key,val,self.root)

else:

self.root = TreeNode(key,val)

self.size = self.size + 1

def \_put(self,key,val,currentNode):

if key < currentNode.key:

if currentNode.hasLeftChild():

self.\_put(key,val,currentNode.leftChild)

else:

currentNode.leftChild = TreeNode(key,val,parent=currentNode)

else:

if currentNode.hasRightChild():

self.\_put(key,val,currentNode.rightChild)

else:

currentNode.rightChild = TreeNode(key,val,parent=currentNode)

def \_\_setitem\_\_(self,k,v):

self.put(k,v)

def get(self,key):

if self.root:

res = self.\_get(key,self.root)

if res:

return res.payload

else:

return None

else:

return None

def \_get(self,key,currentNode):

if not currentNode:

return None

elif currentNode.key == key:

return currentNode

elif key < currentNode.key:

return self.\_get(key,currentNode.leftChild)

else:

return self.\_get(key,currentNode.rightChild)

def \_\_getitem\_\_(self,key):

return self.get(key)

def \_\_contains\_\_(self,key):

if self.\_get(key,self.root):

return True

else:

return False

def delete(self,key):

if self.size > 1:

nodeToRemove = self.\_get(key,self.root)

if nodeToRemove:

self.remove(nodeToRemove)

self.size = self.size-1

else:

raise KeyError('Error, key not in tree')

elif self.size == 1 and self.root.key == key:

self.root = None

self.size = self.size - 1

else:

raise KeyError('Error, key not in tree')

def \_\_delitem\_\_(self,key):

self.delete(key)

def spliceOut(self):

if self.isLeaf():

if self.isLeftChild():

self.parent.leftChild = None

else:

self.parent.rightChild = None

elif self.hasAnyChildren():

if self.hasLeftChild():

if self.isLeftChild():

self.parent.leftChild = self.leftChild

else:

self.parent.rightChild = self.leftChild

self.leftChild.parent = self.parent

else:

if self.isLeftChild():

self.parent.leftChild = self.rightChild

else:

self.parent.rightChild = self.rightChild

self.rightChild.parent = self.parent

def findSuccessor(self):

succ = None

if self.hasRightChild():

succ = self.rightChild.findMin()

else:

if self.parent:

if self.isLeftChild():

succ = self.parent

else:

self.parent.rightChild = None

succ = self.parent.findSuccessor()

self.parent.rightChild = self

return succ

def findMin(self):

current = self

while current.hasLeftChild():

current = current.leftChild

return current

def remove(self,currentNode):

if currentNode.isLeaf(): #leaf

if currentNode == currentNode.parent.leftChild:

currentNode.parent.leftChild = None

else:

currentNode.parent.rightChild = None

elif currentNode.hasBothChildren(): #interior

succ = currentNode.findSuccessor()

succ.spliceOut()

currentNode.key = succ.key

currentNode.payload = succ.payload

else: # this node has one child

if currentNode.hasLeftChild():

if currentNode.isLeftChild():

currentNode.leftChild.parent = currentNode.parent

currentNode.parent.leftChild = currentNode.leftChild

elif currentNode.isRightChild():

currentNode.leftChild.parent = currentNode.parent

currentNode.parent.rightChild = currentNode.leftChild

else:

currentNode.replaceNodeData(currentNode.leftChild.key,

currentNode.leftChild.payload,

currentNode.leftChild.leftChild,

currentNode.leftChild.rightChild)

else:

if currentNode.isLeftChild():

currentNode.rightChild.parent = currentNode.parent

currentNode.parent.leftChild = currentNode.rightChild

elif currentNode.isRightChild():

currentNode.rightChild.parent = currentNode.parent

currentNode.parent.rightChild = currentNode.rightChild

else:

currentNode.replaceNodeData(currentNode.rightChild.key,

currentNode.rightChild.payload,

currentNode.rightChild.leftChild,

currentNode.rightChild.rightChild)

**Реализация отображения:**

**С++:**

Пример реализации класса multimap из STL

template <typename \_Key, typename \_Tp,

typename \_Compare = std::less<\_Key>,

typename \_Alloc = std::allocator<std::pair<const \_Key, \_Tp> > >

class multimap

{

public:

typedef \_Key key\_type;

typedef \_Tp mapped\_type;

typedef std::pair<const \_Key, \_Tp> value\_type;

typedef \_Compare key\_compare;

typedef \_Alloc allocator\_type;

private:

// concept requirements

typedef typename \_Alloc::value\_type \_Alloc\_value\_type;

\_\_glibcxx\_class\_requires(\_Tp, \_SGIAssignableConcept)

\_\_glibcxx\_class\_requires4(\_Compare, bool, \_Key, \_Key,

\_BinaryFunctionConcept)

\_\_glibcxx\_class\_requires2(value\_type, \_Alloc\_value\_type, \_SameTypeConcept)

public:

class value\_compare

: public std::binary\_function<value\_type, value\_type, bool>

{

friend class multimap<\_Key, \_Tp, \_Compare, \_Alloc>;

protected:

\_Compare comp;

value\_compare(\_Compare \_\_c)

: comp(\_\_c) { }

public:

bool operator()(const value\_type& \_\_x, const value\_type& \_\_y) const

{ return comp(\_\_x.first, \_\_y.first); }

};

private:

typedef typename \_Alloc::template rebind<value\_type>::other

\_Pair\_alloc\_type;

typedef \_Rb\_tree<key\_type, value\_type, \_Select1st<value\_type>,

key\_compare, \_Pair\_alloc\_type> \_Rep\_type;

\_Rep\_type \_M\_t;

public:

typedef typename \_Pair\_alloc\_type::pointer pointer;

typedef typename \_Pair\_alloc\_type::const\_pointer const\_pointer;

typedef typename \_Pair\_alloc\_type::reference reference;

typedef typename \_Pair\_alloc\_type::const\_reference const\_reference;

typedef typename \_Rep\_type::iterator iterator;

typedef typename \_Rep\_type::const\_iterator const\_iterator;

typedef typename \_Rep\_type::size\_type size\_type;

typedef typename \_Rep\_type::difference\_type difference\_type;

typedef typename \_Rep\_type::reverse\_iterator reverse\_iterator;

typedef typename \_Rep\_type::const\_reverse\_iterator const\_reverse\_iterator;

multimap()

: \_M\_t(\_Compare(), allocator\_type()) { }

explicit

multimap(const \_Compare& \_\_comp,

const allocator\_type& \_\_a = allocator\_type())

: \_M\_t(\_\_comp, \_\_a) { }

multimap(const multimap& \_\_x)

: \_M\_t(\_\_x.\_M\_t) { }

template <typename \_InputIterator>

multimap(\_InputIterator \_\_first, \_InputIterator \_\_last)

: \_M\_t(\_Compare(), allocator\_type())

{ \_M\_t.\_M\_insert\_equal(\_\_first, \_\_last); }

template <typename \_InputIterator>

multimap(\_InputIterator \_\_first, \_InputIterator \_\_last,

const \_Compare& \_\_comp,

const allocator\_type& \_\_a = allocator\_type())

: \_M\_t(\_\_comp, \_\_a)

{ \_M\_t.\_M\_insert\_equal(\_\_first, \_\_last); }

multimap&

operator=(const multimap& \_\_x)

{

\_M\_t = \_\_x.\_M\_t;

return \*this;

}

allocator\_type

get\_allocator() const

{ return \_M\_t.get\_allocator(); }

iterator

begin()

{ return \_M\_t.begin(); }

const\_iterator

begin() const

{ return \_M\_t.begin(); }

iterator

end()

{ return \_M\_t.end(); }

const\_iterator

end() const

{ return \_M\_t.end(); }

reverse\_iterator

rbegin()

{ return \_M\_t.rbegin(); }

const\_reverse\_iterator

rbegin() const

{ return \_M\_t.rbegin(); }

reverse\_iterator

rend()

{ return \_M\_t.rend(); }

const\_reverse\_iterator

rend() const

{ return \_M\_t.rend(); }

bool

empty() const

{ return \_M\_t.empty(); }

size\_type

size() const

{ return \_M\_t.size(); }

size\_type

max\_size() const

{ return \_M\_t.max\_size(); }

iterator

insert(const value\_type& \_\_x)

{ return \_M\_t.\_M\_insert\_equal(\_\_x); }

iterator

insert(iterator \_\_position, const value\_type& \_\_x)

{ return \_M\_t.\_M\_insert\_equal(\_\_position, \_\_x); }

template <typename \_InputIterator>

void

insert(\_InputIterator \_\_first, \_InputIterator \_\_last)

{ \_M\_t.\_M\_insert\_equal(\_\_first, \_\_last); }

void

erase(iterator \_\_position)

{ \_M\_t.erase(\_\_position); }

size\_type

erase(const key\_type& \_\_x)

{ return \_M\_t.erase(\_\_x); }

void

erase(iterator \_\_first, iterator \_\_last)

{ \_M\_t.erase(\_\_first, \_\_last); }

void

swap(multimap& \_\_x)

{ \_M\_t.swap(\_\_x.\_M\_t); }

void

clear()

{ \_M\_t.clear(); }

key\_compare

key\_comp() const

{ return \_M\_t.key\_comp(); }

value\_compare

value\_comp() const

{ return value\_compare(\_M\_t.key\_comp()); }

iterator

find(const key\_type& \_\_x)

{ return \_M\_t.find(\_\_x); }

const\_iterator

find(const key\_type& \_\_x) const

{ return \_M\_t.find(\_\_x); }

size\_type

count(const key\_type& \_\_x) const

{ return \_M\_t.count(\_\_x); }

iterator

lower\_bound(const key\_type& \_\_x)

{ return \_M\_t.lower\_bound(\_\_x); }

const\_iterator

lower\_bound(const key\_type& \_\_x) const

{ return \_M\_t.lower\_bound(\_\_x); }

iterator

upper\_bound(const key\_type& \_\_x)

{ return \_M\_t.upper\_bound(\_\_x); }

const\_iterator

upper\_bound(const key\_type& \_\_x) const

{ return \_M\_t.upper\_bound(\_\_x); }

std::pair<iterator, iterator>

equal\_range(const key\_type& \_\_x)

{ return \_M\_t.equal\_range(\_\_x); }

std::pair<const\_iterator, const\_iterator>

equal\_range(const key\_type& \_\_x) const

{ return \_M\_t.equal\_range(\_\_x); }

template <typename \_K1, typename \_T1, typename \_C1, typename \_A1>

friend bool

operator== (const multimap<\_K1, \_T1, \_C1, \_A1>&,

const multimap<\_K1, \_T1, \_C1, \_A1>&);

template <typename \_K1, typename \_T1, typename \_C1, typename \_A1>

friend bool

operator< (const multimap<\_K1, \_T1, \_C1, \_A1>&,

const multimap<\_K1, \_T1, \_C1, \_A1>&);

};

template <typename \_Key, typename \_Tp, typename \_Compare, typename \_Alloc>

inline bool

operator==(const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_x,

const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_y)

{ return \_\_x.\_M\_t == \_\_y.\_M\_t; }

template <typename \_Key, typename \_Tp, typename \_Compare, typename \_Alloc>

inline bool

operator<(const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_x,

const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_y)

{ return \_\_x.\_M\_t < \_\_y.\_M\_t; }

template <typename \_Key, typename \_Tp, typename \_Compare, typename \_Alloc>

inline bool

operator!=(const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_x,

const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_y)

{ return !(\_\_x == \_\_y); }

template <typename \_Key, typename \_Tp, typename \_Compare, typename \_Alloc>

inline bool

operator>(const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_x,

const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_y)

{ return \_\_y < \_\_x; }

template <typename \_Key, typename \_Tp, typename \_Compare, typename \_Alloc>

inline bool

operator<=(const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_x,

const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_y)

{ return !(\_\_y < \_\_x); }

template <typename \_Key, typename \_Tp, typename \_Compare, typename \_Alloc>

inline bool

operator>=(const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_x,

const multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_y)

{ return !(\_\_x < \_\_y); }

template <typename \_Key, typename \_Tp, typename \_Compare, typename \_Alloc>

inline void

swap(multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_x,

multimap<\_Key, \_Tp, \_Compare, \_Alloc>& \_\_y)

{ \_\_x.swap(\_\_y); }

**Java:**

**Класс SAX-анализатор:**

/\*\*

\* Метод обработки начала элемента

\*/

public void startElement(String uri, String Sname,

String qname, Attributes attr)throws SAXException

{

// Обработка события на начало тэга

if(qname.equals("groupnode"))

{

try {

baseObject = Class.forName(attr.getValue("baseobject"));

parNameNode = attr.getValue("parnamenode");

parent = Class.forName(attr.getValue("parent"));

visible = new Boolean(attr.getValue("visible")).booleanValue();

enabled = new Boolean(attr.getValue("enabled")).booleanValue();

parameter = new Boolean(attr.getValue("parameter")).booleanValue();

nameNode = attr.getValue("namenode") ;

// Создаем экземпляр классаGroupNode...

//проверяем есть ли запрос с параметром

if(parameter)

{

nameField = attr.getValue("nameField");

typeField = attr.getValue("typeField");

valueField = attr.getValue("valueField");

groupNode = new GroupNode(baseObject,nameNode,parNameNode, parent,

visible,enabled,parameter,nameField,typeField,valueField);

}

else

groupNode = new GroupNode(baseObject,nameNode, parNameNode,

parent,visible,enabled,parameter);

listGroupNode.add(groupNode);

} catch (ClassNotFoundException ex)

{

ex.printStackTrace();

}

Создание корневых элементов:

listGroupNode = treeHandler.getListGroupNode();

for (Object elem : listGroupNode)

{

// проверяем загруженный список на наличие корневых нод

if(javax.swing.tree.TreeNode.class.equals(((GroupNode) elem).getParentObj()))

{

root.add((GroupNode) elem);

// проверяем есть ли запросы с параметром

if(((GroupNode) elem).isParameter()){

// если есть проверяем какого типа параметр

if(((GroupNode) elem).getTypeField().equals("String"))

this.createNode((GroupNode) elem,queryDB.getParamColNto(

((GroupNode) elem).getBaseObject(),

((GroupNode) elem).getNameField(),

((GroupNode) elem).getValueField(),

((GroupNode) elem).getValueField()

));

if(((GroupNode) elem).getTypeField().equals("long"))

this.createNode((GroupNode) elem,queryDB.getParamColNto(

((GroupNode) elem).getBaseObject(),

((GroupNode) elem).getNameField(),

new Integer(((GroupNode) elem).getValueField()).intValue()

));

}

else

{

this.createNode((GroupNode) elem,queryDB.getColNto(

((GroupNode) elem).getBaseObject() ));

}

}

Создание узлов:

public void createNode(DefaultMutableTreeNode groupNode,List listNode){

for (NodeTreeObj nto :listNode )

{

DefaultMutableTreeNode dtmT = new DefaultMutableTreeNode(nto);

groupNode.add(dtmT);

fillGroupNode((GroupNode) groupNode,dtmT);

}

}

Создание узла с объектом группировки:

void fillGroupNode(GroupNode groupNode,DefaultMutableTreeNode dmtn)

{

NodeTreeObj ntoL = (NodeTreeObj) dmtn.getUserObject();

for (GroupNode tmpGroupNode : this.getListGroupNode()) {

if(tmpGroupNode.getParNameNode().equals(groupNode.toString()))

{

try{

// Определяем базовую бизнес сущность для узла группировки

if( tmpGroupNode.getParentObj() = = ntoL.getNameCls())

{

// Создание класса по шаблону

GroupNode realGroupNode =(GroupNode)tmpGroupNode.clone();

dmtn.add(realGroupNode);

fillNodeTree(realGroupNode, ntoL);

}

}catch(Exception ex){ex.printStackTrace();}

}

}

}

Метод заполнения узла:

public void fillNodeTree(GroupNode fGroupNode, NodeTreeObj nto){

try{

Method meId = fGroupNode.getBaseObject().getMethod("getId");

Method meName = fGroupNode.getBaseObject().getMethod("getName");

// Находим метод вызова дочерних объектов определенного класса

Method nesMethod = this.getNesMethod(nto.getNameCls(), fGroupNode.getBaseObject());

Object obj = queryDB.getObjTreeDB(nto);

Set list = (Set) nesMethod.invoke(obj);

if (list!=null){

List listNto = new ArrayList();

for ( Object elem : list)

{

// вызываем эти методы и заносим полученные данные в коллекцию

NodeTreeObj nto1 = new NodeTreeObj(

new Long(meId.invoke(elem).toString()).longValue(),

fGroupNode.getBaseObject(),

meName.invoke(elem).toString()

);

listNto.add(nto1);

}

this.createNode(fGroupNode,listNto);

}

}catch(Exception exp){ exp.printStackTrace();}

}

**Python:**

**Пример реализации:** [**https://github.com/python/cpython/blob/2.7/Objects/dictobject.c**](https://github.com/python/cpython/blob/2.7/Objects/dictobject.c)