Тема 3. Ссылочные схемы хранения данных

Лекция 9. Итераторы по дереву, продолжение

Итератор по дереву

На прошлой лекции обсудили итератор по дереву, точнее логику выбора очередной вершины при смещении итератора.

```
TreeNode * NextPosition (TreeNode *p)
{
    TreeNode *q;
    if (p == 0) return 0;
    if (p->left) return p->left;
    if (p->right) return p->right;
    for (q = p->parent; q; p = q, q = q->parent) {
        if (q->right && q->right != p) return q->right;
    }
    return 0;
}
```

Но в процедурах обхода указатель parent не используется. Можно ли построить итератор без указателя parent ? В процедурах обхода обратный путь запоминался естествунным образом в стеке вызовов при рекурсивной работе функции. Поэтому и мы можем запоминать пройденный путь (узлы) в стеке. Для реализации итератора будем считать, что текущая позиция лежит на вершине стека.

```
TreeNode * NextPosition (Stack<TreeNode*> st)
{
    TreeNode *p;
    if (st.Size() == 0) return 0;
    if (st.Top()->left) { st.Push(st.Top()->left); return st.Top(); }
    if (st.Top()->right) { st.Push(st.Top()->right); return st.Top(); }
    while (st.Size() > 1) {
       st.Pop(p);
       if (st.Top()->right && st.Top()->right != p) {
           st.Push(st.Top()->right);
           return st.Top();
       }
    }
    st.Del();
    return 0;
}
```

Теперь осталось проверить этот код и исправить все ошибки и опечатки, которые в нем могли возникнуть. Но для этого надо создать какое-нибудь нетривиальное дерево. Т.е. надо определить класс Tree, к которому можно будет применять итератор.

Для начала сделаем совсем примитивный класс бинарного дерева, и заполним его “вручную”, устанавливая ссылки между некоторым количеством элементов. Для этого

временно реализуем в классе отдельный метод, который создает некоторе фикированное дерево по заданной последовательности команд. В дальнейшем, когда мы введем разные лисциплины формирования дерева, мы избавимся от этого метода и будем уже заполнять дерево “по правилам”.

```
template <class T>
class Tree
{
 private:
   struct TreeNode
   {
      T value;
      TreeNode *left, *right, *parent;
   };

   TreeNode *root;
 public:
   Tree ();
   void CreateSampleTree ();

   class P_Iterator // итератор с использованием parent
   {
   };
   P_Iterator begin();
   P_Iterator end();

   class S_Iterator // итератор с использованием стека и без parent
   {
   };
   S_Iterator begin();
   S_Iterator end();
};
```

Пример: Tree-test.cpp

Можно аналогично придумать логику выбора узлов дерева, соответствующую другим способам прохода — снизу-вверх, справа-налево и т.д. Можно добавить константные итераторы и пр.

Деревья поиска

Определение. Бинарное дерево называется деревом поиска, если для любого узла A выполнено соотношение

```
x < A.value < y
```

где x - произвольное значение из левого поддерева, y - произвольное значение из правого поддерева.

Можно дать определение с нестрогим неравенством `x <= A.value < y` , но на практике все интересные случаи используют строгие неравенства, поэтому их и имеем в виду.

Такое определение сразу задает правила работы с деревом поиска, а именно, сам поиск значения, и операции добавления и удаления элементов так, чтобы поддерживалось определение.

Далее в этом тексте будем писать не совсем строгий код, чтобы не загромождать и проиллюстрировать именно логику работы (т.е. опустим template, строгое описание типов и т.п.). А строгий код будет уже в рабочих примерах.

Возможны рекурсивные и нерекурсивные реализации.

Поиск. Хотим узнать, есть ли в дереве конкретное значение X и в каком узле оно находится

```
TreeNode * Search(TreeNode * root, const T & x)
{
    if (root == nullptr) return nullptr;
    if (x == root->value) return root;
    if (x > root->value) return Search(root->right, x);
    else  return Search(root->left, x);
}

TreeNode * Search(TreeNode * root, T & x)
{
    if (root == nullptr) return nullptr;
    TreeNode *p = root;
    while (p) {
       if (x == p->value) return p;
       p = (x > p->value) ? p->right : p->left;
    }
    return nullptr;
}
```

Добавление. Процедура возвращает указатель на вершину всего дерева после добавления

```
TreeNode * Add(TreeNode * root, const T & x)
{
    if (root == nullptr) {
        root = new TreeNode;
        root->value = x;
    } else {                    // нет проверки на наличие x в дереве !!!
        if (x > root->value) {
            root->right = Add(root->right, x);
            root->right->parent = root;
        } else {
            root->left = Add(root->left, x);
            root->left->parent = root;
        }
```

```
    }
    return root;
 }

TreeNode * Add(TreeNode * root, const T & x)
{
    TreeNode *px = new TreeNode;
    px->value = x;
    if (root == nullptr) {
        root = px;
    } else {          // нет проверки на наличие x в дереве !!!
        for (TreeNode *p = root; p; ) {
            if (x > p->value) {
                if (p->right) { p = p->right; }
                else { p->right = px;
                       px->parent = p;
                       break;
                }
            } else {
                if (p->left) { p = p->left; }
                else { p->left = px; break; }
            }
        }
    }
    return root;
 }
```

Удаление. Иллюстрация на картинках.

```
TreeNode * Remove(TreeNode * root, const T & x)
{
    TreeNode *p;
    if (root == nullptr) return nullptr;
    if (x > root->value) {
        root->right = Remove(root->right, x);
        if (root->right) root->right->parent = root;
    } else if (x < root->value) {
        root->left = Remove(root->left, x);
        if (root->left) root->left->parent = root;
    } else {
        if (root->right == nullptr) {
            p = root->left;
            delete root;
            return p;
        }
        if (root->left == nullptr) {
            p = root->right;
            delete root;
```

```
            return p;
        }
        p = SearchRightmost (root->left);
        root->value = p->value;
        root->left = Remove(root->left, root->value);
        if (root->left) root->left->parent = root;
    }
    return root;
}

TreeNode * p = SearchRightmost (TreeNode *root)
{
    for ( ; root->right; root = root->right);
    return root;
}
```

Нерекурсивный вариант — в качестве самостоятельного упражнения.
Нигде не использовался указатель parent.
1 — можно его учитывать и соответственно устанавливать
2 — можно не учитывать и вообще убрать из реализации
Недостатки — дерево может выродиться в список, трудоемкость $O(N)$.