Тема 4. Деревья. Обходы. Итераторы по дереву

Деревья.

Следующая структура, естественно возникающая в рамках ссылочной реализации, — это дерево.

Определение дерева всем известно* (по крайней мере, на интуитивном уровне). Поэтому условимся о следующих терминах и обозначениях

* Ориентированный связный граф, причем только одна вершина не имеет входящих ребер (корень), а остальные вершины имеют ровно одно входящее ребро.

** Неориентированный связный граф без циклов с одной выделенной вершиной, называемой корнем.

*** Рекурсивное определение. Пустое дерево. Дерево из одной вершины. Новый корень с потомками на корнях существующих деревьев.

— корень

— потомок и родитель

— концевая вершина (лист) — узел, не имеющий потомков

— ветвь — цепочка “родителей – потомков” от корня до листа

можно также рассматривать восходящую и нисходящую ветви (корень вверху, лист внизу).

— растояние от корня до верщины — количество вершин в ветви от корня до данного узла

— длина ветви — количество вершин от корня до листа

— уровень — множество вершин, лежащих на одном расстоянии от корня

Произвольное (сильно ветвящееся дерево) — количество потомков конкретного узла может быть любым.

Бинарное дерево — каждая вершина имеет не более двух потомков.

Иллюстрации: деревья, термины, ссылки.

Схема ссылочной реализации бинарного дерева. Структура узла

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

Схема ссылочной реализации произвольного дерева. Структура узла

```
template <class T>
struct TreeNode {
    T value;
    TreeNode *child, *brother, *parent;
};
```

Лучше их реализовывать внутренним образом так как много разных типов деревьев, и пусть специфические узлы определяются внутри с одинаковыми именами, чтобы не плодить разные имена типов узлов под каждое специфическое дерево.

Обходы.

Иллюстрация обхода бинарного и произвольного дерева
Рекурсивная процедура обхода.
Бинарное дерево

```
template <class T>
void Walk ( typename BinTree<T>::TreeNode *root, void (*Process)(T &x) )
{  // top-bottom, left-right
    if (root == 0) return;
    Process(root->value);
    Walk(root->left, Process);
    Walk(root->right, Process);
}
```

Например, поиск узла полным проходом по дереву

```
template <class T>
typename BinTree<T>::TreeNode * TotalSearch
( typename BinTree<T>::TreeNode *root, const T &x )
{
    typename BinTree<T>::TreeNode * p;
    if (root == 0) return 0;
    if (root->value == x) return root;
    p = TotalSearch (root->left, x);
    if (p) return p;
    p = TotalSearch (root->left, x);
    if (p) return p;
    return 0;
}
```

Произвольное дерево

```
template <class T>
void Walk ( typename GenTree<T>::TreeNode *root, void (*Process)(T &x) )
{
    typename GenTree<T>::TreeNode * p;
    if (root == 0) return;
    Process(root->value);
    for (p = root->child; p; p = p->next) {
        Walk(p, Process);
    }
}
```

но если иерархия не важна, то можно по аналогии с бинарным деревом

```
template <class T>
void Walk ( typename Gen<T>::TreeNode *root, void (*Process)(T &x) )
{
    if (root == 0) return;
```

```
    Process(root->value);
    Walk(root->child, Process);
    Walk(root->next, Process);
}
```

Обратите внимание, ссылка parent не используется !

Процедуры обходов не являются итератором, поскольку основаны на стеке вложенных вызов и не могут прервать обход, а потом продолжить с того же места.

Итератор по дереву

Рассмотрим бинарное дерево. Пусть находимся в некоторой вершине. Какая вершина должна стать следующей (++ для итератора) ?

```
p - указатель на текущую вершину
следующая:
   p->left
   если p->left == 0, то
      следующая p->right
          если p->right == 0, то
          надо подняться вверх по ветви,
          пока не обнаружим указатель направо
          и идти на первый же узел справа.
          (различаем подъем слева и справа)
            если и этого нет, то конец
```

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

```
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;
}
```

Можно ли построить итератор без указателя parent ? Можно, но придется запоминать пройденный путь (узлы) в стеке. Текущая позиция 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 () : root(0) {}
    void CreateSampleTree ();

    class P_Iterator // итератор с использованием parent
    {
    };

    class S_Iterator // итератор с использованием стека и без parent
    {
    };

    P_Iterator begin();
    P_Iterator end();
    S_Iterator begin();
    S_Iterator end();

};
```