Тема 4. Деревья

Деревья. Обходы. Итераторы по дереву

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

Определение дерева всем известно (по крайней мере, на интуитивном уровне). Оно часто определяется как:

* 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.

Условимся об использовании следующих понятий:

— 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 BinTree

```
template <class T>
void Walk ( typename BinTree<T>::TreeNode *root, void (*Process)(T &x) )
```

```
{  // top-bottom, left-right
    if (!root) 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) 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) 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
```

Другие обходы получаются перестановкой порядка обработки текущего элемента (текущего корня) и его поддеревьев. Таким образом мы можем построить 6 различных вариантов обхода бинарного дерева.

Схемы перехода к следующей вершине выписываются достаточно очевидным образом, но надо соблюдать аккуратность. Например, обход снизу-вверх, слева-направо, т.е. сначала обрабатываются поддеревья, а потом текущий корень.

```
p - указатель на текущую вершину
    // значит поддеревья этого элемента уже пройдены
    // и надо подниматься наверх
q = p->parent
если q = 0, то конец
иначе
   если p лежит слева от q, то
      следующая позиция есть q, выход
   иначе // p справа от q
      r = q->left
      если r = 0, то следующая есть q
      иначе
         спускаемся до последнего элемента
         самой правой ветви от r, это
         и есть очередной начальный элемент
         в поддереве от r
```

Этот алгоритм можно реализовать чуть более компактно с сохранением той же идеи.

Примерно по таким же схемам строятся проходы в остальных случаях. Примеры итераторов, реализующих подобные проходы, представлены в примерах Tree1.zip и Tree2.zip. В первом случае итераторы оформлены как внешние классы, во втором как внутренние. Для случая внутренних классов выбор соответствующего итератора выполняется при выборе начальной позиции с помощью функции типа begin(), т.е. вводятся несколько функций для инициализации каждого конкртеного итераторы.

Все итераторы используют указатель parent. Но как мы видели выше операции с деревом могут и не требовать этого указателя. Можно ли построить итератор без указателя parent ? Да, можно, но тогда придется запоминать пройденный путь (узлы) в стеке для того, чтобы в любой момент можно было бы подняться к родительской вершине, поскольку именно она находится на вершине стека. Например, выбор следующей позиции при проходе сверху-вниз, слева направо как в первом примере может выглядеть примерно так

```
TreeNode * NextPosition (Stack<TreeNode*> st)
{
    TreeNode *p;
    if (st.Size() == 0) return nullptr;
    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 nullptr;
}
```

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.

В примерах Tree1.zip и Tree2.zip дерево создается с помощью простейшей процедуры добавления элемента у порядоченное дерево поиска, которое мы булем обсуждать в следующий раз. А пока после заполнения дерева проверяется работа различных итераторов с распечаткой значений в узлах этого дерева.