Отчет №10. Итераторы деревьев

Сканирование узлов дерева более сложно, чем сканирование массивов и последовательных списков, так как дерево является нелинейной структурой и существует несколько способов прохождения дерева. Мы уже разбирали эти способы ранее. Проблема каждого из них состоит в том, что до завершения рекурсивного процесса из него невозможно выйти. Нельзя остановить сканирование, проверить содержимое узла, выполнить какие-нибудь операции с данными, а затем вновь продолжить сканирование со следующего узла дерева. Используя же итератор, клиент получает средство сканирования узлов дерева, как если бы они представляли собой линейный список, без обременительных деталей алгоритмов прохождения, лежащих в основе процесса.

Наш класс использует класс Stack и наследуется от базового класса итератора. Поэтому сначала опишем класс Stack и базовый класс итератора.

```
#include <iostreani.h>
#include <stdlib.h>

const int MaxStackSize = 50;

class Stack
{
    private:
        DataType stacklist[MaxStackSize];
        int top;
    public:
        // конструктор; инициализирует вершину
        Stack(void);

        // операции модификации стека
        void Push(const DataType& item);
        DataType Pop(void);
        void ClearStack(void);

        // доступ к стеку
        DataType Peek(void) const;

       // методы проверки состояния стека
       int StackEmpty(void) const;
       int StackFull(void) const; // для реализации, основанной на массиве
};
```

Описание

Данные в стеке имеют тип DataType, который должен определяться с использованием оператора typedef. Пользователь должен проверять, полный ли стек, перед попыткой поместить в него элемент и проверять, не пустой ли стек, перед извлечением данных из него. Если предусловия для операции push или pop не удовлетворяются, печатается сообщение об ошибке и программа завершается.

StackEmpty возвращает TRUE, если стек пустой, и FALSE — в противном случае. Используйте StackEmpty, чтобы определить, может ли выполняться операция Pop.

StackFull возвращает TRUE, если стек полный, и FALSE — в противном случае. Используйте StackFull, чтобы определить, может ли выполняться операция Push.

ClearStack делает стек пустым, устанавливая top = -1. Этот метод позволяет использовать стек для других целей.

Реализация класса Stack

Конструктор Stack присваивает индексу top значение -1, что эквивалентно условию пустого стека.

```
//инициализация вершины стека
Stack::Stack (void) : top(-1)
{ }
```

Операции стека.

Две основные операции стека вставляют (Push) и удаляют (Pop) элемент из стека. Класс содержит функцию Peek, позволяющую получать данные элемента, находящегося в вершине стека, не удаляя в действительности этот элемент.

При помещении элемента в стек, top увеличивается на 1, и новый элемент вставляется в конец массива stacklist. Попытка добавить элемент в полный стек приведет к сообщению об ошибке и завершению программы.

```
// поместить элемент в стек
void Stack::Push (const DataTypes item)
{
   // если стек полный, завершить выполнение программы
   if (top ==  MaxStackSize-1)
  {
      cerr << "Переполнение стека!" << endl;
      exit(1);
   }
   // увеличить индекс top и копировать item в массив stacklist
   top++;
   stacklist[top] = item;
}
```

Операция Pop извлекает элемент из стека, копируя сначала значение из вершины стека в локальную переменную temp и затем увеличивая top на 1. Переменная temp становится возвращаемым значением. Попытка извлечь элемент из пустого стека приводит к сообщению об ошибке и завершению программы.

```
// взять элемент из стека
DataType Stack::Pop (void)

   DataType temp;

  // стек пуст, завершить программу
  if (top == -1)
  {
    cerr <<  "Попытка обращения к пустому стеку! " << end.1;
    exit(1);
  }

  // считать элемент в вершине стека
  temp = stacklist[top];

  // уменьшить top и возвратить значение из вершины стека
  top--;
  return temp;
}
```

Операция Peek в основном дублирует определение Pop с единственным важным исключением. Индекс top не уменьшается, оставляя стек нетронутым.

```
// возвратить данные в вершине стека
DataType Stack::Peek (void) const
{
    // если стек пуст, завершить программу
    if (top == -1)
   {
       cerr <<  "Попытка считать данные из пустого стека!" <<  end.1;
       exit(1);
    }
    return stacklist[top];
}
```

Условия тестирования стека

Во время своего выполнения операции стека завершают программу при попытках клиента обращаться к стеку неправильно; например, когда мы пытаемся выполнить операцию Peek над пустым стеком. Для защиты целостности стека класс предусматривает операции тестирования состояния стека.

Функция StackEmpty проверяет, является ли top равным -1. Если — да, стек пуст и возвращаемое значение — TRUE; иначе возвращаемое значение — FALSE.

```
// тестирование стека на наличие в нем данных
int Stack::StackEmpty(void) const
{
   return top == -1;
}
```

Функция StackFull проверяет, равен ли top значению MaxStackSize - 1. Если так, то стек заполнен и возвращаемым значением будет TRUE; иначе, возвращаемое значение — FALSE.

```
// проверка, не переполнен ли стек
int Stack::StackFuli(void) const
{
   return top == MaxStackSize-1;
}
```

Метод ClearStack переустанавливает вершину стека на -1. Это восстанавливает начальное условие, определенное конструктором.

```
// удалить все элементы из стека
void Stack::ClearStack(void)
{
    top = -1;
}
```

Стековые операции Push и Pop используют прямой доступ к вершине стека и не зависят от количества элементов в списке.

Абстрактный базовый класс Iterator

Класс Iterator позволяет абстрагироваться от тонкостей реализации алгоритма перебора, что дает независимость от деталей реализации базового класса. Мы определяем абстрактный класс Iterator как шаблон для итераторов списков общего вида.

Спецификация класса Iterator

Объявление

```
template <class T>
class Iterator
{
protected:
// Флаг, показывающий, достиг ли итератор конца списка.
// Должен поддерживаться производными классами.
int iterationComplete;

public:

// конструктор
Iterator(void);

// обязательные методы итератора
virtual void Next(void) = 0;
virtual void Reset(void) = 0;

// методы для выборки/модификации данных
virtual T& Data(void) = 0;

// проверка конца списка
virtual int EndOfList(void) const;

};
```

Обсуждение

Итератор является средством прохождения списка. Его основные методы: Reset (установка на первый элемент списка), Next (переход на следующий элемент), EndOfList (определение конца списка). Функция Data осуществляет доступ к данным текущего элемента списка.

Итератор симметричного метода прохождения

Симметричное прохождение бинарного дерева поиска, в процессе которого узлы посещаются в порядке возрастания их значений, является полезным инструментом.

Объявление

```
// итератор симметричного прохождения бинарного дерева.
// использует базовый класс Iterator
template <class T>
class InorderIterator: public Iterator<T>
{
  private:
    // поддерживать стек адресов узлов
    Stack< TreeNode <T> * > S;
    // корень дерева и текущий узел
    TreeNode<T> *root, *current;

    // сканирование левого поддерева используется функцией Next
    TreeNode<T> *GoFarLeft(TreeNode<T> *t);

  public:
    // конструктор
    InorderIterator(TreeNode<T> *tree);

    // реализации базовых операций прохождения
    virtual void Next(void);
    virtual void Reset(void);
    virtual T& Data(void);

    // назначение итератору нового дерева
    void SetTree(TreeNode<T> *tree);
};
```

Описание

Класс InorderIterator построен по общему для всех итераторов образцу. Метод EndOfList определен в базовом классе Iterator. Конструктор инициализирует базовый класс и с помощью GoFarLeft находит начальный узел сканирования.

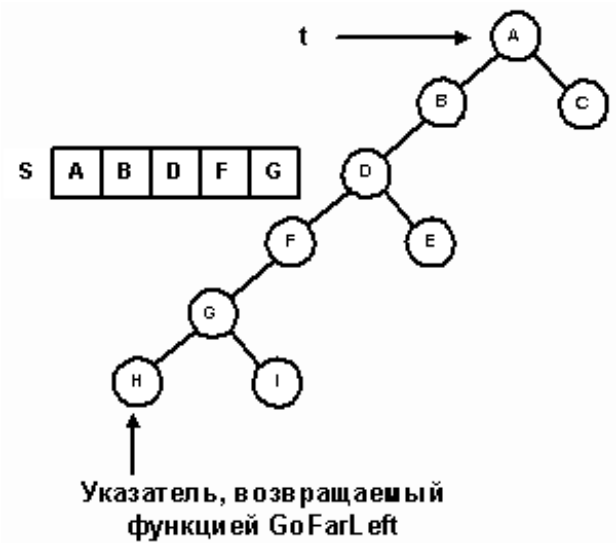
Пример

```
TreeNode<int> *root;           // бинарное дерево
InorderIterator treeiter(root);    // присоединить итератор

// распечатать начальный узел сканирования.
// для смешанного прохождения это самый левый узел дерева
cout << treeiter.Data();

// сканирование узлов и печать их значений
for (treeiter.Reset(); !treeiter.EndOfList(); treeiter.Next())
  cout << treeiter.Data() << " ";
```

Реализация класса InorderIterator

Итерационный симметричный метод прохождения эмулирует рекурсивное сканирование с помощью стека адресов узлов. Начиная с корня, осуществляется спуск вдоль левых поддеревьев. По пути указатель каждого пройденного узла запоминается в стеке. Процесс останавливается на узле с нулевым левым указателем, который становится первым посещаемым узлом в симметричном сканировании. Спуск от узла t и запоминание адресов узлов в стеке выполняет метод GoFarLeft. Вызовом этого метода с t=root ищется первый посещаемый узел.

```
// вернуть адрес крайнего узла на левой ветви узла t.
// запомнить в стеке адреса всех пройденных узлов
template <class T>
TreeNode<T> *InorderIterator<T>::GoFArLeft(TreeNode<T> *t)
{
  // если t=NULL, вернуть NULL
  if (t == NULL)
    return NULL;
  // пока не встретится узел с нулевым левым указателем,
  // спускаться по левым ветвям, запоминая в стеке S
  // адреса пройденных узлов. Возвратить указатель на этот узел
  while (t->Left() != NULL)
  {
    S.Push(t);
    t = t->Left();
  }
  return t;
}
```



После инициализации базового класса конструктор присваивает элементу данных root адрес корня бинарного дерева поиска. Узел для начала симметричного сканирования получается в результате вызова функции GoFarLeft с root в качестве параметра. Это значение запоминается в переменной current.

```
// инициализировать флаг iterationComplete. Базовый класс сбрасывает его, но
// дерево может быть пустым. начальный узлел сканирования - крайний слева узел.
template <class T>
InorderIterator<T>::InorderIterator(TreeNode<T> *tree):
                      Iterator<T>(), root(tree)
{
  iterationComplete = (root == NULL);
  current = GoFarLeft(root);
}
```

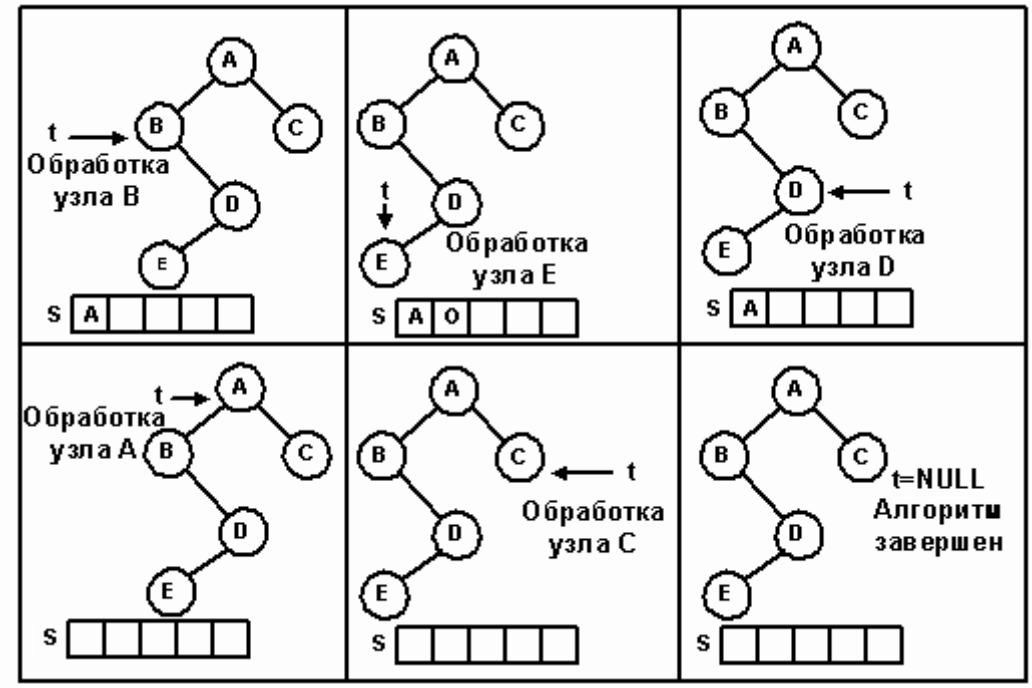
Метод Reset по существу является таким же, как и конструктор, за исключением того, что он очищает стек.

Перед первым обращением к Next указатель current уже указывает на первый узел симметричного сканирования. Метод Next работает по следующему алгоритму.

Если правая ветвь узла не пуста, перейти к его правому сыну и осуществить спуск по левым ветвям до узла с нулевым левым указателем, попутно запоминая в стеке адреса пройденных узлов.

Если правая ветвь узла пуста, то сканирование его левой ветви, самого узла и его правой ветви завершено. Адрес следующего узла, подлежащего обработке, находится в стеке. Если стек не пуст, удалить следующий узел. Если же стек пуст, то все узлы обработаны и сканирование завершено.

Итерационное прохождение дерева, состоящего из пяти узлов, изображено на рисунке.



```
template <class T>
void InorderIterator<T>::Next(void)
{
  // ошибка, если все узлы уже посещались
  if (iterationComplete)
  {
    cerr << "Next: итератор прошел конец списка!" << endl;
    exit(1);
  }

  // current - текущий обрабатываемый узел.
  // если есть правое поддерево, спуститься до конца по его левой ветви,
  // попутно запоминая в стеке адреса пройденных узлов
  if (current->Right() != NULL)
    current = GoFarLeft(current->Right());

  // правого поддерева нет, но в стеке есть другие узлы,
  // подлежащие обработке. вытолкнуть из стека новый текущий адрес,
  // продвинуться вверх по дереву
  else if (!S.StackEmpty())
    current = S.Pop();
  // нет ни правого поддерева, ни узлов в стеке. сканирование завершено
  else
    iterationComplete = 1;
}
```

Источники:

1. У. Топп, У. Форд – Структуры данных в C++
2. https://habr.com/ru/post/144850/ - полезное про обходы деревьев
3. https://www.rsdn.org/article/alg/bintree/avl.xml полезный код к АВЛ деревьям, также про итераторы