Отчёт №2. Деревья. Алгоритм вставки узла в бинарное дерево. Использование алгоритмов прохождения деревьев

Рекурсивные методы прохождения деревьев.

Рекурсивное определение бинарного дерева определяет эту структуру как корень с двумя поддеревьями, которые идентифицируются полями левого и правого указателей в корневом узле. Сила рекурсии проявляется вместе с методами прохождения.

Каждый алгоритм прохождения дерева выполняет в узле три действия: заходит в узел, рекурсивно спускается по левому поддереву и по правому поддереву. Спустившись к поддереву, алгоритм определяет, что он находится в узле, и может выполнить те же три действия. Спуск прекращается по достижении пустого дерева (указатель == NULL). Различные алгоритмы рекурсивного прохождения отличаются порядком, в котором они выполняют свои действия в узле.

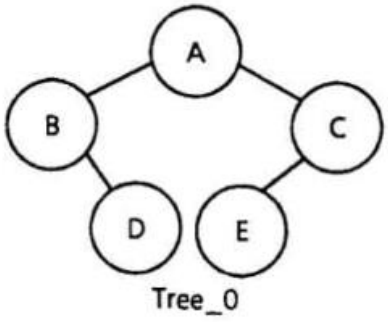
Симметричный метод прохождения дерева

Симметричный методом прохождения начинает свои действия в узле спуском по его левому поддереву. Затем выполняется второе действие – обработка данных в узле. Третье действие – рекурсивное прохождение правого поддерева. В процессе рекурсивного спуска действия алгоритма повторяются в каждом новом узле.

Итак, порядок операций при симметричном методе следующий:

1. Прохождение левого поддерева
2. Посещение узла
3. Прохождение правого поддерева

Мы называем такое прохождение LNR(left, node, right). Для дерева Tree_0 в функции MakeCharTree «посещение» означает печать значения из поля данных узла.



Tree_0

При симметричном методе прохождения дерева Tree_0 выполняются следующие операции:

Действие	Печать	Замечания
Спуститься от A к B:		Левый сын узла B равен NULL
Посетить B;	B	
Спуститься от B к D:		D — листовой узел
Посетить D;	D	Конец левого поддерева узла A
Посетить корень A:	A	
Спуститься от A к C:		
Спуститься от C к E:		E — листовой узел
Посетить E;	E	
Посетить C;	C	Готово!

Узлы посещаются в порядке B D A E C. Рекурсивная функция сначала спускается по левому дереву [t -> Left()], а затем посещает узел. Второй шаг рекурсии спускается по правому дереву [t -> Right()].

```
//симметричное рекурсивное прохождение узлов дерева
template <class T>
void Inorder (TreeNode<T>*t, void visit(T& item))
{
    //рекурсивное прохождение завершается на пустом поддереве
    if (t != NULL)
    {
        //спуститься по левому поддереву
        Inorder(t->Left(), visit);
        //посетить узел
        visit(t->data);
        //спуститься по правому поддереву
        Inorder(t->Right(), visit);
    }
}
```

Обратный метод прохождения дерева

При обратном прохождении посещение узла откладывается до тех пор, пока не будут рекурсивно пройдены оба его поддерева. Порядок операций дает так называемое LRN (left, right, node) сканирование

1. Прохождение левого поддерева
2. Прохождение правого поддерева
3. Посещение узла

При обратном прохождении дерева Tree_0 узлы посещаются в порядке D B E C A.

Действие	Печать	Замечания
Спуститься от A к B:		Левый сын узла B равен NULL
Спуститься от B к D:		D — листовой узел
Посетить D;	D	Все сыновья узла B пройдены
Посетить B;	B	Левое поддерево узла A пройдено
Спуститься от A к C:		
Спуститься от C к E:		E — листовой узел
Посетить E;	E	Левый сын узла C
Посетить C;	C	Правый сын узла A
Посетить корень A:	A	Готово!

Функция сканирует дерево снизу вверх. Мы спускаемся вниз по левому дереву [t -> Left()], а затем вниз по правому [t -> Right()]. Последней операцией является посещение узла.

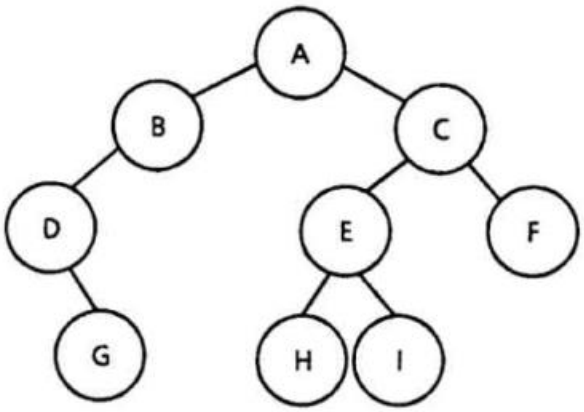
Прямой метод прохождения определяется посещением узла в первую очередь и последующим прохождением сначала левого, а потом правого его поддеревьев (NLR).

Ясно, что префиксы pre, in и post в названиях функций показывают, 0когда происходит посещение узла. В каждом случае сначала осуществлялось прохождение по левому поддереву, а уже потом по правому. Фактически существуют еще три алгоритма, которые выбирают сначала правое поддерево и потом левое. Для печати дерева будем использовать RNL-прохождение.

Алгоритмы прохождения посещают каждый узел дерева. Они дают эквивалент последовательного сканирования массива или связанного списка. Функции прямого, симметричного и обратного методов прохождения содержатся в файле treescan.h.

Пример из учебника

1. Для символьного дерева Tree_2 имеет место следующий порядок



Tree_2

посещения узлов.

Прямой:	**A B D G C E H I F**
Симметричный:	**D G B A H E I C F**
Обратный:	**G D B H I E F C A**

2. Результат симметричного прохождения дерева Tree_2 производится следующими операторами:

```
// функция visit распечатывает поле данных
void PrintChar(char& elem)
{
  cout <<elem << "  ";
}

TreeNode<char> *root;

MakeCharTree(root, 2); // сформировать дерево Tree_2 с корнем root

// распечатать заголовок и осуществить прохождение, используя
// функцию PrintChar для обработки узла
cout < "Симметричное прохождение: ";
Inorder (root, PrintChar);
```

Использование алгоритмов прохождения деревьев

На рекурсивных алгоритмах прохождения основаны многие приложения деревьев. Эти алгоритмы обеспечивают упорядоченный доступ к узлам. Далее будет рассматриваться использование алгоритмов прохождения для подсчета кол-ва листьев на дереве, глубины дерева и для печати дерева.

Приложение: посещение узлов дерева

Для многих приложений требуется просто обойти узлы дерева, неважно в каком порядке. В этих случаях клиент волен выбирать любой алгоритм прохождения

В данном приложении функция проходит дерево с целью подсчета его листьев. При распознавании очередного листа происходит приращение параметра count.

```
// эта функция использует обратный метод прохождения.
// во время посещения узла проверяется, является ли он листовым
template CountLeaf (<TreeNode<T> *t, int& count)
{
  // Использовать обратный метод прохождения
  if (t != NULL)
  {
    CountLeaf(t->Left(), count);  // пройти левое поддерево
    CountLeaf(t->Right(), count); // пройти правое поддерево

    // Проверить, является ли данный узел листом.
    // Если да, то произвести приращение переменной count
    if (t->Left() == NULL && t->Right() == NULL)
      count++;
  }
}
```

Следующая функция использует обратный метод прохождения для вычисления глубины бинарного дерева. В каждом узле вычисляется глубина его левого и правого поддерева, а итоговая глубина на единицу больше максимальной глубины поддеревьев

```
// эта функция использует обратный метод прохождения для вычисления глубины
// левого и правого поддеревьев узла и возвращает результирующее
// значение глубины, равное 1 + max(depthLeft, depthRight).
// глубина пустого дерева равна -1
template <class T>
void Depth (TreeNode<T> *t)
{
  int depthLeft, depthRight, depthval;

  if (t == NULL)
    depthval = -1;
  else
  {
    depthLeft = Depth(t->Left());
    depthRight = Depth(t->Right());
    depthval = 1 + (depthLeft > depthRight?depthLeft:depthRight);
  }
  return depthval;
}
```

Следующая программа иллюстрирует использование выше описанных функций для прохождения символьного дерева, итоговые значения распечатываются

```
#include <iostream.h>

// включить класс TreeNode и библиотеку функций
#include "treenode.h"
#include "treelib.h"

void main(void)
{
  TreeNode<char> *root;

  // использовать дерево Tree_2
  MakeCharTree(root, 2);

  // переменная, которая обновляется функцией CountLeaf
  int leafCount = 0;

  // вызвать функцию CountLeaf для подсчета числа листьев
  CountLeaf(root, leafCount);
  cout << "Число листьев равно " << leafCount << endl;

  // вызвать функцию Depth для вычисления глубины дерева
  cout << "Глубина дерева равна "
       << Depth(root) << endl;
}
```

```
/*
<Выполнение программы 11.1>

Число листьев равно 4
Глубина дерева равна 3

*/
```

Приложение: печать дерева

Функция печати дерева создает изображение дерева. Поскольку принтер выводит информацию построчно, алгоритм использует RNL-прохождение и распечатывает узлы правого поддерева раньше узлов левого поддерева. Узлы дерева печатаются в порядке F C I E H A B G D

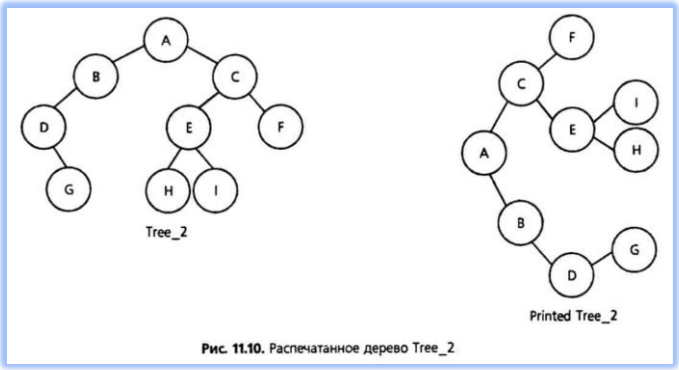


Рис. 11.10. Распечатанное дерево Tree_2

```
                Уровни
          0     1     2     3
      12                F
      6         C
      18                      I
Отступ 12               E
      18                      H
      0  A
      6         B
      18                      G
      12                D
```

Рис. 11.11. Печать дерева Tree_2

Код функции PrintTree находится в файле treeprint.h.

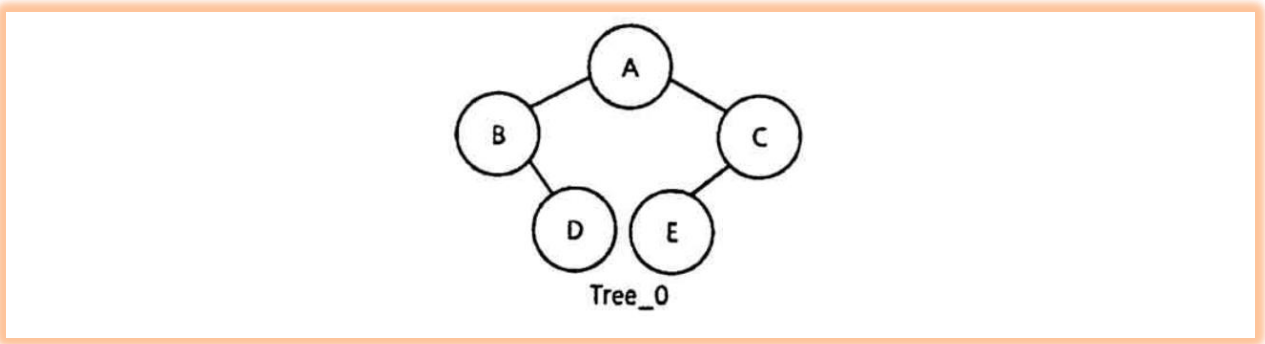
```
// промежуток между уровнями
const int indentBlock = 6;

// вставить num пробелов на текущей строке
void IndentBlanks(int num)
{
  for (int i = 0; i < num; i++)
    cout << " ";
}
// распечатать дерево боком, используя RNL-прохождение
template <class T>

void PrintTree (Treenode<T> *t, int level)
{
  // печатать дерево с корнем t, пока t != NULL
  if (t != NULL)
  {
    // печатать правое поддерево узла t
    PrintTree(t->Right(), level+1);

    // выровнять текущий уровень и вывести поле данных
    IndentBlanks(indentBlock * level);
    cout << t->data << endl;
    // печатать левое поддерево
    PrintTree(t->Left(), level+1);
  }
}
```

Приложение: копирование и удаление деревьев

Функция CopyTree принимает исходное дерево и создает его дубликат. Процедура DeleteTree удаляет узел каждый узел дерева, включая корень, и высвобождает занимаемую узлами память. Функции, разработанные для бинарных деревьев общего вида, находятся в файле treelib.h.

Копирование дерева. Функция CopyTree использует для посещения узла обратный метод прохождения. Этот метод гарантирует, что мы спустимся по

дереву на максимальную глубину, прежде чем начнем операцию посещения, которая создает узел для нового дерева. Функция CopyTree строит новое дерево снизу вверх. Сначала создаются сыновья, а затем они присоединяются к своим родителям, как только те будут созданы.

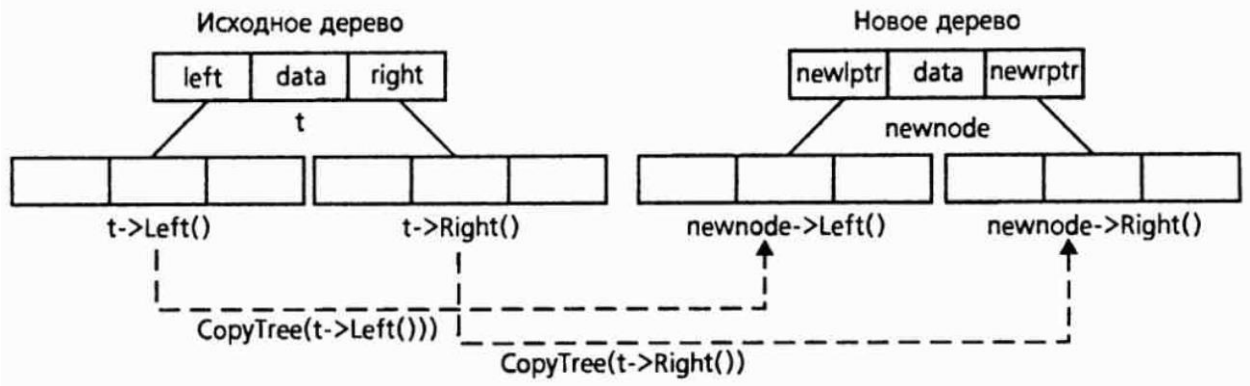



```
d = GetTreeNode('D');
e = GetTreeNode('E');
b = GetTreeNode('B', NULL, d);
c = GetTreeNode('C', e, NULL);
a = GetTreeNode('A', b, c);
root = a;
```

Сначала мы создаем сына D, который затем присоединяется к своему родителю C во время рождения или создания последнего. Наконец, создается корень и присоединяется к своим сыновьям B и C.

Алгоритм копирования дерева начинает с корня и в первую очередь строит левое поддерево узла, а затем – правое его поддерево. Только после этого создается новый узел. Тот же рекурсивный процесс повторяется для каждого узла. Соответственно узлу t исходного дерева создается новый узел с указателями newlptr и newrptr.

При обратном прохождении сыновья посещаются перед их родителями. В результате в новом дереве создаются поддеревья, соответствующие t ->Left() и t -> Rilght(). Сыновья присоединяются к своим родителям в момент создания последних.

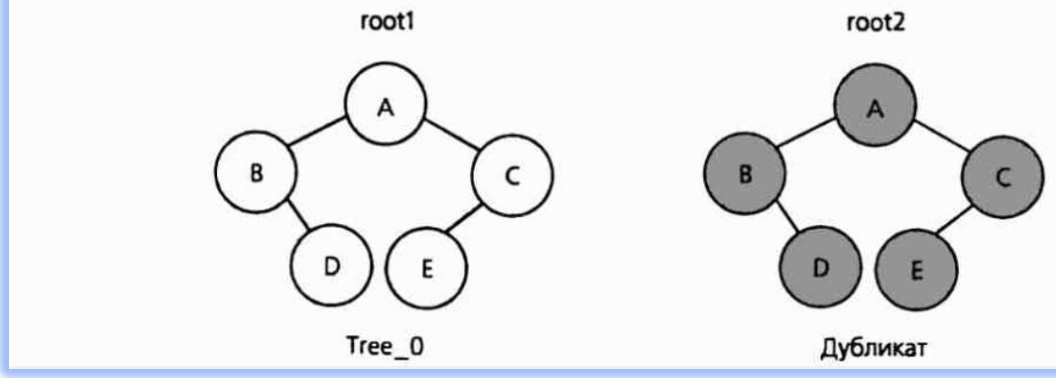


```
newlptr = CopyTree(t->Left());
newrptr = CopyTree(t->Right());

// создать родителя и присоединить к нему его сыновей
newnode = GetTreeNode(t->data, newlptr, newrptr);
```

Суть посещения узла t в исходном дереве заключается в создании нового уза на дереве-дубликате.

Символьное дерево Tree_0 является примером, иллюстрирующим рекурсивную функцию CopyTree. Предположим, что главная процедура определяет корни root1, root2 и создает дерево Tree_0. Функция CopyTree создает новое дерево с корнем root2.

```
TreeNode<char> *root1, *root2;   // объявить два дерева
MakeCharTree(root1, 0);          // root1 указывает на Tree_0

root2 = CopyTree(root1);         // создать копию дерева Tree_0
```



1. Пройти потомков узла A, начиная с левого поддерева в узле B и далее к узлу D, который является правым поддеревом узла B. Создать новый узел с данными, равными D, и левым и правым указателями, равными NULL [рис. 11.12 (A)].
2. Сыновья узла B пройдены. Создать новый узел с данными, равными B, левым указателем, равным NULL, и правым указателем, указывающим на узел D [рис. 11.12 (B)].
3. Поскольку левое поддерево узла A пройдено, начать прохождение его правого поддерева и дойти до узла E. Создать новый узел с данными из узла E и указательными полями, равными NULL.
4. После обработки E перейти к его родителю и создать новый узел с данными из C. В поле правого указателя поместить NULL, а левому указателю присвоить ссылку на дочерний узел E [рис. 11.13 (A)].
5. Последний шаг выполняется в узле A. Создать новый узел с данными из A и присоединить к нему сына B слева и сына C справа [рис. 11.13 (B)]. Копирование дерева завершено.

Функция CopyTree возвращает указатель на вновь созданный узел. Это возвращаемое значение используется родителем, когда тот создает свой собственный узел и присоединяет к нему своих сыновей. Функция возвращает корень вызывающей программе.

```
// создать дубликат дерева t и возвратить корень нового дерева
template <class T>
TreeNode<T> *CopyTree(TreeNode<T> *t)
{
  // переменная newnode указывает на новый узел, создаваемый
  // посредством вызова GetTreeNode и присоединяемый в дальнейшем
  // к новому дереву. указатели newlptr и newrptr адресуют сыновей
  // нового узла и передаются в качестве параметров в GetTreeNode
  TreeNode<T> *newlptr, *newrptr, *newnode;

  // остановить рекурсивное прохождение при достижении пустого дерева
  if (t == NULL)
    return NULL;

  // CopyTree строит новое дерево в процессе прохождения узлов дерева t. в каждом
  // узле этого дерева функция CopyTree проверяет наличие левого сына. если он
  // есть, создается его копия. в противном случае возвращается NULL. CopyTree
  // создает копию узла с помощью GetTreeNode и подвешивает к нему копии сыновей.

  if (t->Left() != NULL)
    newlptr = CopyTree(t->Left());
  else
    newlptr = NULL;

  if (t->Right() != NULL)
    newrptr = CopyTree(t->Right());
  else
    newrptr = NULL;

  // построить новое дерево снизу вверх, сначала создавая
  // двух сыновей, а затем их родителя
  newnode = GetTreeNode(t->data, newlptr, newrptr);

  // вернуть указатель на вновь созданное дерево
  return newnode;
}
```

Удаление дерева. Когда в приложении используется такая динамическая структура, как дерево, ответственность за освобождение занимаемой им памяти ложится на программиста. Для бинарного дерева общего вида разработаем функцию, в которой применяется обратный метод прохождения. Это гарантирует, что мы посетим всех сыновей родительского узла, прежде чем удалим его.

```
// использовать обратный алгоритм для прохождения узлов дерева
// и удалить каждый узел при его посещении
 template <class T>
 void DeleteTree(TreeNode<T> *t)
 {
 if (t != NULL)
 {
     DeleteTree(t->Left());
    DeleteTree(t->Right());
    FreeTreeNode(t);
  }
}
```

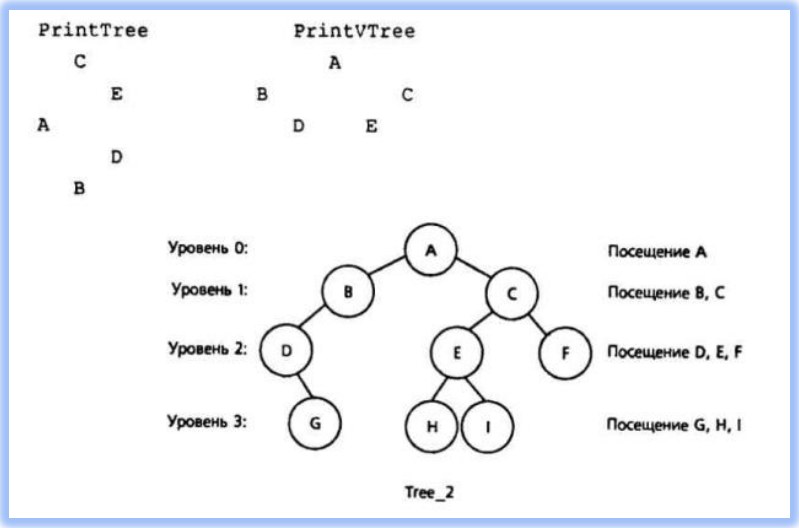
Более общая процедура удаления дерева удаляет узлы и сбрасывает корень. Функция ClearTree вызывает DeleteTree для удаления узлов дерева и присваивает указателю на корень значение NULL.

```
// вызвать функцию DeleteTree для удаления узлов дерева.
// затем сбросить указатель на его корень в NULL
template <class T>
void ClearTree(TreeNode<T> &t)
{
  DeleteTree(t);
  t = NULL;
  // теперь корень пуст
}
```

Приложение: вертикальная печать дерева

Функция PrintTree создает повернутое набок изображение дерева. На каждой строке узел распечатывается в позиции, определяемой его уровнем. Хотя такое дерево воспринимать трудно, этот прием позволяет распечатывать большие деревья.

Функция PrintVTree требует нового алгоритма прохождения, который сканирует дерево уровень за уровнем, начиная с корня на ур. 0. Этот метод, называемый поперечным прохождением или прохождением уровней, не спускается рекурсивно вдоль поддеревьев, а просматривает дерево поперек, посещая все узлы на одном уровне, и затем переходит на уровень ниже. В отличие от рекурсивного спуска здесь более предпочтителен итерационный алгоритм, использующий очередь элементов. Для каждого узла в очередь помещается всякий непустой левый или правый указатель на сына этого узла. Это гарантия того, что одноуровневые узлы следующего уровня будут посещаться в нужном порядке. Символьное дерево Tree_2 иллюстрирует этот алгоритм.



Алгоритм поперечного прохождения

Шаг инициализации:

Поместить в очередь корневой узел.

Шаги итерации:

Прекратить процесс, если очередь пуста.

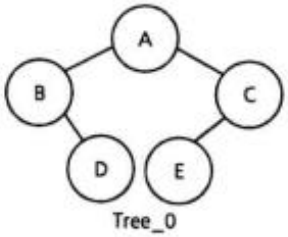
Удалить из очереди передний узел p и распечатать его значение.

Использовать этот узел для идентификации его детей на следующем уровне дерева.

```
if (p->Left() != NULL)     // проверить наличие левого сына
  Q.QInsert(p->Left());
if (p->Right() != NULL)    // проверить наличие правого сына
  Q.QInsert(p->Right());
```

Пример 11.3

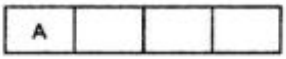
Алгоритм поперечного прохождения иллюстрируется на дереве Tree_0.



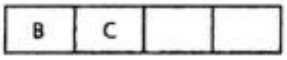
Tree_0

Инициализация: Вставить узел A в очередь.

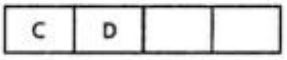
1: Удалить узел A из очереди.
Печатать A.
Вставить сыновей узла A в очередь.
Левый сын = B
Правый сын = C



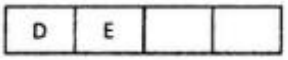
2: Удалить узел B из очереди.
Распечатать B.



3: Удалить узел C из очереди.
Левый сын = E

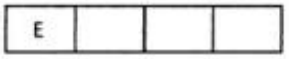


4: Удалить узел D из очереди.
Распечатать D.



Узел D не имеет сыновей.

5: Удалить узел E из очереди.
Алгоритм завершается. Очередь пуста.




```
// Прохождение дерева уровень за уровнем с посещением каждого узла
template <class T>
void LevelScan(TreeNode<T> *t, void visit(T& item))
{
  // запомнить сыновей каждого узла в очереди, чтобы их
  // можно было посетить в этом порядке на следующем уровне
  Queue<TreeNode<T> *> Q;
  TreeNode<T> *p;
  // инициализировать очередь, вставив туда корень
  Q.Qinsert(t);

  // продолжать итерационный процесс, пока очередь не опустеет
  while(!Q.QEmpty())
  {
    // удалить первый в очереди узел и выполнить функцию visit
    p = Q.QDelete();
    visit(p->data);

    // если есть левый сын, вставить его в очередь
    if (p->Left() != NULL)
      Q.QInsert(p->Left());
    // если есть правый сын, вставить его в очередь
    if (p->Right() != NULL)
      Q.QInsert(p->Right());
  }
}
```

Алгоритм PrintVTree. В функцию вертикальной печати дерева передается корень дерева, максимальная ширина данных и ширина экрана:

```
void PrintVTree(TreeNode<T> *t, int dataWidth, int screenWidth)
```

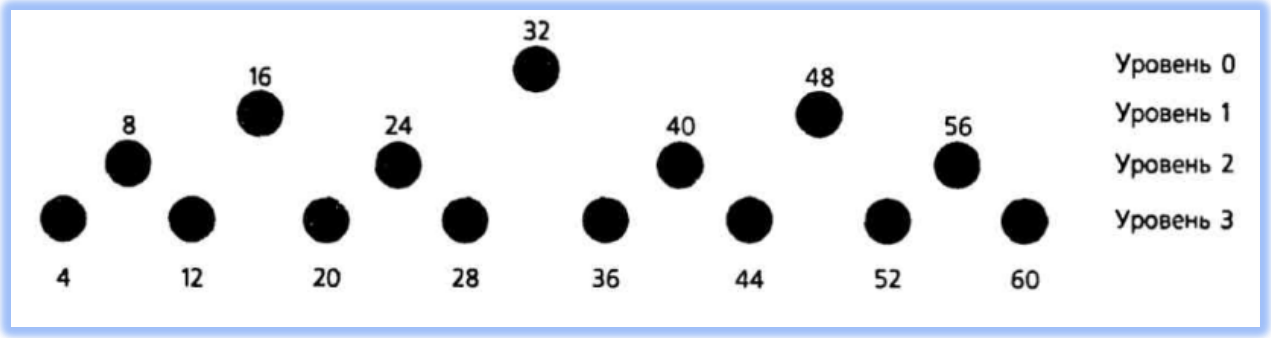
Параметры ширины позволяют организовать экран. Пусть dataWidth = 2 и screenWidth = 64 = 2^6. Тот факт, что значение ширины равно степени двойки, позволяет описать поуровневую организацию данных. Поскольку мы не знаем структуру дерева, то полагаем, что места должно хватать для полного бинарного дерева. Узлы строятся в координатах (уровень, смещение).

Уровень 0: Корень рисуется в точке (0, 32).

Уровень 1: Поскольку корень смещен на 32 позиции, следующий уровень имеет смещение 32/2 = 16 = screenWidth/2^2. Два узла первого уровня располагаются в точках (1, 32-смещение) и (1, 32+смещение) т.е. в точках (1, 16) и (1, 48).

Уровень 2: На втором уровне смещение равно screenWidth/2^3 = 8. Четыре узла второго уровня располагаются в точках (2, 16-смещение), (2, 16+смещение), (2, 48-смещение), (2, 48+смещение), т.е. в точках (2,8), (2, 24), (2, 40) и (2, 56).

Уровень i: Смещение равно screenWidth/2^(i+1). Позиция каждого узла данного уровня определяется во время посещения его родителя на уровне i-1. Пусть позиция родителя равна (i-1, parentPos). Если узел i-го уровня является левым сыном, то его позиция равна (i, parentPos-смещение), а если правым – (i, parentPos+смещение).



PrintVTree использует две очереди и поперечный метод прохождения узлов дерева. В очереди Q находятся узлы, а очередь QI содержит уровни и позиции печати в форме записей типа Info. Когда узел добавляет в очередь

Q, соответствующая ему информация о печати запоминается в QI. Элементы удаляются в тандеме во время посещения узла.

```
// запись для хранения координат (x,y) узла
struct Info
{
 int xIndent, yLevel;
};

// Очереди для хранения узлов и информации о печати
Queue<TreeNode<T> * Q;
Queue<Info> QI;
```

Эта программа распечатывает символьное дерево Tree_2 на 30 или на 60-символьном листе. Ширина данных для вывода dataWidth = 1.

```
#include <iostream.h>

// включить функцию PrintVTree из библиотеки
#include "treelib.h"
#include "treeprnt.h"

void main (void)
{
  // объявить символьное дерево
  TreeNode<char> *root;

  // назначить дереву Tree_2 корень root
  MakeCharTree(root, 2);
  cout << "Печать дерева на 30-символьном экране" << endl;
  PrintVTree(root, 1, 30);
  cout << endl << endl;

  cout << "Печать дерева на 60-символьном экране" << endl;
  PrintVTree(root, 1, 60);
}
/*
```

```
<Выполнение программы 11.3>

Печать дерева на 30-символьном экране
             A

     B              C

  D              E     F

    G           H   I

Печать дерева на 60-символьном экране
                           A


            B                              C


     D                             E           F

        G                       H     I

*/
```

Вторая лабораторная работа:

- Построить бинарное дерево из 15 целочисленных узлов
- Создать функцию печати дерева горизонтально и вертикально
- Распечатать дерево на экране и в текстовом файле.
- Созданные функции находятся в модуле

Файл main.cpp

```
#include <iostream>
#include<fstream>
#include<time.h>
#include <TreeNode.h>
using namespace std;

int main()
{
    //подключаем файл для вывода
    string s = "rezult.txt";
    //открываем файл для записи
    ofstream file(s, ios_base::app);
    //для рандомных чисел
    srand(time(0));
    //количество элементов очереди или узлов дерева
    queue<int> D1;

    D1.push(50);
    D1.push(60);
    D1.push(40);
    D1.push(45);
    D1.push(65);

    D1.push(55);
    D1.push(35);
    D1.push(62);
    D1.push(58);
    D1.push(68);

    D1.push(20);
    D1.push(39);
    D1.push(36);
    D1.push(34);
    D1.push(67);

    //создаем узел дерева - корень
    TreeNode<int> *B1, *B2;
    //заполняем дерево очередью
    B1 = CreateBinTree(D1);

    //выводим глубину дерева
    cout <<"Depth = "<< Depth(B1)<< endl;
    file <<"Depth = "<< Depth(B1)<< endl;
    //создаем счетчик листьев дерева
    int leafCount = 0;
    CountLeaf(B1,leafCount);
    cout <<"Count Leaf = "<<leafCount<<endl;
    file <<"Count Leaf = "<<leafCount<<endl;
    //количество узлов дерева
```

```
    int size = 0;
    Size(B1,size);
    cout<<"Size = "<<size<<endl;
    file<<"Size = "<<size<<endl;

    //Горизонтальная печать дерева
    cout<<"Горизонтальная печать"<<endl;
    file<<"Горизонтальная печать"<<endl;
    PrintTree(B1,0);
    PrintTreeFile(B1,0,s);
    //проверим функцию копирования
    B2 = CopyTree(B1);
    //вертикальная печать
    cout<<"Вертикальная печать"<<endl;
    file<<"Вертикальная печать"<<endl;
    PrintVTreeFile(B2,0,64,PrintFile,s);
    PrintVTree(B1,0,64,Print);
    DeleteTree(B1);
    DeleteTree(B2);
    file<<endl;
    return 0;
}
```

Файл TreeNode.H

```
#ifndef TREENODE_H
#define TREENODE_H
#include <cstdlib>
#include <iostream>
#include<fstream>
#include <math.h>
#include<queue>
using namespace std;

//BinSTree зависит от TreeNode
template<class T>
class BinSTree;

//класс узла бинарного дерева
template <class T>

class TreeNode
{
  private:
    //указатели левого и правого дочерних узлов
    TreeNode<T> *left;
    TreeNode<T> *right;

  public:
    //открытый элемент, допускающий обновление
    T data;
    //конструктор
    TreeNode (const T& item, TreeNode<T> *lptr = NULL,
              TreeNode<T> *rptr = NULL){
        data = item;
        left=lptr;
        right=rptr;
    }

    //методы доступа к полям указателей
    TreeNode<T>* Left() const{
```

```
        return left;
    }
    TreeNode<T>* Right() const{
        return right;
    }

    //сеттеры
    void set_Left(TreeNode<T>* newNode){
        left = newNode;
    }

    void set_Right(TreeNode<T>* newNode){
        right = newNode;
    }

    //для доступа к полям
    friend class BinSTree<T>;
};

//освободить память, занимаемую узлом
template<class T>
void FreeTreeNode(TreeNode<T> *p){
    delete p;
}

//создание и инициализация узла бинарного дерева
template<class T>
TreeNode<T>* GetTreeNode(const T& item, TreeNode<T> *lptr = NULL,
                                        TreeNode<T> *rptr = NULL){
    TreeNode<T> *p;
    p = new TreeNode<T>(item,lptr,rptr);
    //проверка памяти
    if (p == NULL)
    {
        cout<<"Error!"<<endl;
        exit(1);
    }
    return p;
 }

//симметричное рекурсивное прохождение узлов дерева
template<class T>
void Inorder(TreeNode<T> *t,void visit(T& item)){
    if (t!=NULL){
        Inorder(t->Left(), visit);
        visit(t->data);
        Inorder(t->Right(),visit);
    }
}

//обратное рекурсивное прохождение узлов дерева
template<class T>
void Postorder(TreeNode<T> *t,void visit(T& item)){
    if (t!=NULL){
        Postorder(t->Left(), visit);
        Postorder(t->Right(), visit);
        visit(t->data);
    }
}

//прямой метод прохождения узлов дерева
```

```
template<class T>
void Preorder(TreeNode<T> *t,void visit(T& item)){
    if (t!=NULL){
          visit(t->data);
        Postorder(t->Left(), visit);
        Postorder(t->Right(), visit);
    }
}

//функция подсчета листьев дерева
//count предусматривает себя как глобальный параметр и передается по
ссылке
template<class T>
void CountLeaf(TreeNode<T> *t, int& count){
 if (t!=NULL)
 {
     //используем обратный метод прохождения
     CountLeaf(t->Left(),count);
     CountLeaf(t->Right(),count);
     //является ли узел листовым?
     if(t->Left()==NULL && t->Right()==NULL)
         count++;
 }
}

//количество элементов дерева
template <class T>
void Size(TreeNode<T>* t, int& count) {
    if (t != NULL) {
        count++;
        Size(t->Left(),count);
        Size(t->Right(),count);
    }
}

//функция подсчета глубины дерева
template <class T>
int Depth(TreeNode<T> *t){
    int depthLeft, depthRight, depthval;
    if (t == NULL)
        //если корневой узел пустой, то глубина дерева -1
        depthval = -1;
    else
    {
        //используем обратный метод прохождения
        depthLeft = Depth(t->Left());
        depthRight = Depth(t->Right());
        //прибавляем наибольшего сына
        depthval = 1+(depthLeft>depthRight?depthLeft:depthRight);
    }
   return depthval;
}

//функция вставки для построения дерева
template <class T>
void Insert(TreeNode<T>* root, const T item){
  //t - текущий узел, parent - предыдущий узел
  TreeNode<T>* t = root, * parent = NULL;
  //создание узла из переданного значения
  TreeNode<T>* newNode = GetTreeNode(item);
  //закончить на пустом дереве
```

```
  while (t != NULL)
  {
     parent = t; //обновить указатель parent
     //выбор пути
     if (newNode->data < t->data) //если новые данные меньше текущего узла
         t = t->Left(); //идём влево
     else
         t = t->Right(); //а иначе - вправо
   }
   if (parent == NULL) //если родителя нет
       root = newNode; //вставить в качестве корневого узла
   else if (newNode->data < parent->data) //если новые данные меньше
родительского узла
           parent->set_Left(newNode); //то вставить в качестве левого
потомка
        else
           parent->set_Right(newNode); // иначе вставить в качестве
правого потомка
}

//создание бинарного дерева из очереди
template <class T>
TreeNode<T>* CreateBinTree(queue<T> Q_data) {
    TreeNode<T>* root = GetTreeNode(Q_data.front());
    Q_data.pop();
    while (!Q_data.empty()) {
        Insert(root, Q_data.front());
        Q_data.pop();
    }
    return root;
}

//вставка пробелов
void IndentBlanks(int num) {
    for (int i = 0; i < num; i++){
        cout << " ";
    }
}

//горизонтальная печать дерева на экран
template <class T>
void PrintTree(TreeNode<T>* t, int level) {
    if (t != NULL) { //пока указатель не пуст
        PrintTree(t->Right(), level+1); //печатаем правое поддерево
        IndentBlanks(6 * level); //вставка пробелов и выравнивание
        cout << t->data << endl;
        PrintTree(t->Left(), level+1); //печатаем левое поддерево
    }
}

//горизонтальная печать дерева в файл
template <class T>
void PrintTreeFile(TreeNode<T>* t, int level,string s1) {
    if (t != NULL) { //пока указатель не пуст
        ofstream f(s1, ios_base::app);
        PrintTreeFile(t->Right(), level+1,s1); //печатаем правое поддерево
        for (int i = 0; i < 6*level; i++){
            f << " ";
        }
        f<<t->data<<endl;
        PrintTreeFile(t->Left(), level+1,s1); //печатаем левое поддерево
```

```
    }
}

//функция удаления дерева
template <class T>
void DeleteTree(TreeNode<T>*t){
    if (t!=NULL){
        DeleteTree(t->Left());
        DeleteTree(t->Right());
        FreeTreeNode(t);
    }
}

//функция удаления дерева под корень
template <class T>
void ClearTree(TreeNode<T> &t){
    DeleteTree(t);
    //очищаем корень
    t=NULL;
}

//функция создания дубликата дерева с возвращением указателя корня на
новое дерво
template<class T>
TreeNode<T>* CopyTree(TreeNode<T> *t){
    TreeNode<T> *newlptr, *newrptr, *newnode;
    if(t==NULL)
        return NULL;
    if(t->Left()!=NULL)
        newlptr=CopyTree(t->Left());
    else
        newlptr =NULL;
    if(t->Right()!=NULL)
        newrptr=CopyTree(t->Right());
    else
        newrptr =NULL;
    //новое дерво строится снизу вверх
    newnode=GetTreeNode(t->data,newlptr,newrptr);
    return newnode;
}

//тип координат узла
struct Info {
    int xIndent, yLevel;
};

template <class T>
void Print(queue<T>& Q, queue<Info>& QI){
    int lastDepth = 0; int lastIndent = 0;
        while (!Q.empty()) {
            Info Position = QI.front();
            QI.pop();
            if (Position.yLevel > lastDepth) {
                cout << endl;
                lastDepth = Position.yLevel;
                lastIndent = 0;
            }
            IndentBlanks((Position.xIndent - lastIndent));
            lastIndent = Position.xIndent;
            cout << Q.front();
            Q.pop();
```

```
        }
}

template <class T>
void PrintFile(queue<T>& Q, queue<Info>& QI, string s1){
    ofstream f(s1, ios_base::app);
    int lastDepth = 0; int lastIndent = 0;
        while (!Q.empty()) {
            Info Position = QI.front();
            QI.pop();
            if (Position.yLevel > lastDepth) {
                f << endl;
                lastDepth = Position.yLevel;
                lastIndent = 0;
            }
            //IndentBlanks((Position.xIndent - lastIndent));
            for (int i = 0; i < (Position.xIndent - lastIndent); i++){
                f << " ";
            }
            lastIndent = Position.xIndent;
            f << Q.front();
            Q.pop();
        }
}

//прохождение дерева по уровням
template <class T>
void LevelScan(TreeNode<T> *t,queue<T> &out, queue<Info> &inf, int
dataWidth, int screenWidth){
    //запомнить сыновей каждого узла в очереди чтобы их
    //можно было посетить в этом порядке
    queue<TreeNode<T>*> Q;//очередь из узлов
    queue<Info> QI; //очередь из координат узлов
    Info Position;//текущие координаты
    TreeNode<T> *p;//указатель на обрабатываемый узел
    //инициализируем очередь, вставив туда корень
    Q.push(t);
    //инициализируем очередь , вставляя координаты корня
    QI.push({screenWidth/2-(dataWidth/2),0});
    //пока очередь не опустеет
    while (!Q.empty()){
        //удалить первый узел и посетить
        p=Q.front();
        Q.pop();
        Position = QI.front();
        QI.pop();
        out.push(p->data);//вставляем данные из узла в очередь
        inf.push(Position);
        if (p->Left()!=NULL){
            Q.push(p->Left());
            int div = pow(2,(Position.yLevel+2));
            QI.push({Position.xIndent - (screenWidth/div) -
(dataWidth/2),Position.yLevel + 1});
        }
        if (p->Right()!=NULL){
            Q.push(p->Right());
            int div = pow(2,(Position.yLevel+2));
            QI.push({Position.xIndent + (screenWidth/div) -
(dataWidth/2),Position.yLevel + 1});
        }
    }
```

```
}

//вертикальная печать дерева на экран
template <class T>
void PrintVTree(TreeNode<T>* t, int dataWidth, int screenWidth, void
f(queue<T>& Q_data, queue<Info>& Q_info)){
    queue<T> Q;
    queue<Info> QI;
    LevelScan(t,Q,QI,dataWidth,screenWidth);
    f(Q,QI);
}

//вертикальная печать дерева в файл
template <class T>
void PrintVTreeFile(TreeNode<T>* t, int dataWidth, int screenWidth, void
f(queue<T>& Q_data, queue<Info>& Q_info,string s1),string s){
    queue<T> Q;
    queue<Info> QI;
    LevelScan(t,Q,QI,dataWidth,screenWidth);
    f(Q,QI,s);
}

#endif // TREENODE_H
```

Результат работы программы:

```
04:41:55: Запускается D:\homework\5_semester\saod\build-lab2-Desktop_Qt_6_1_3_MinGW_64_bit-Debug\debug\lab2.exe ...
Depth = 4
Count Leaf = 6
Size = 15
Gorizontal
                68
                     67
           65
                62
      60
                58
           55
50
           45
      40
                39
                     36
           35
                34
                20
Vertical
                          50
               40                                  60
       35               45               55               65
    20       39                                  58       62       68
      34    36                                                  6704:41:55: D:\homework\5_semester\saod\build-lab2-
Desktop_Qt_6_1_3_MinGW_64_bit-Debug\debug\lab2.exe завершился с кодом 0
```