Лабораторная работа № 5

Бинарные деревья поиска

Цель работы: изучить базовые алгоритмы работы с деревьями: построение, обход, поиска элемента, удаление элемента, подсчет количества узлов, нахождение высоты дерева, исследовать свойства деревьев, закрепить навыки структурного программирования.

Общие сведения

Листинг базовых функций приведен в приложении А.

Задание

1. Реализовать функции вставки, поиска и удаления узла, обхода дерева, вставки в корень, вывода дерева на экран, нахождения высоты дерева и количества узлов, а также функцию вставки, строящую рандомизированное дерево.
2. Построить зависимости высоты деревьев (обычного бинарного поиска и рандомизированного) от количества ключей (ключи – случайные и упорядоченные величины), полагая, что ключи целые числа.
3. Реализовать заданную функцию (в соответствии с вариантом: *T* – тип ключей, *D* – диапазон изменения значений ключей).
4. Составить отчет, в котором привести графики полученных зависимостей, анализ свойств алгоритмов, листинги функций вставки, поиска, удаления узла с комментариями и выводы по работе.

№	*T*	*D*	Функция
1	`int`	[1000; 2000]	Подсчет суммы длин путей от корня до каждого из узлов, содержащих четные числа
2	`char`	[a..я, A..Я]	Подсчет количества гласных в листьях
3	`int`	[0; 1000]	Подсчет количества нечетных чисел в узлах, имеющих ровно два поддерева
4	`char`	[a..я, A..Я]	Подсчет количества согласных на 2 и 3 уровнях
5	`int`	[-500; 500]	Подсчет сумму четных отрицательных чисел в узлах, поддеревья которых содержат не более 4 узлов
6	`char`	[a..я, A..Я]	Подсчет количества гласных на четных уровнях
7	`int`	[1000; 2000]	Подсчет суммы четных чисел во внутренних узлах
8	`char`	[a..z, A..Z]	Подсчет количества согласных в узлах, высота поддеревьев которых одинакова
9	`int`	[2, 4, 6, … 2000]	Подсчет суммы длин путей от корня до каждого из внутренних узлов
10	`char`	[a..z, A..Z]	Подсчет количества гласных во внутренних узлах

Приложение А

Пример реализации базовых функций работы с деревьями

```
typedef struct Node * pNode;
typedef void(*pFunction)(pNode);
typedef int Item;
struct Node  {pNode Left; pNode Right; Item  Data;};
//---------------------Создание узла-----------------------------------------
pNode NewNode(Item D)
{
pNode  pN = malloc(sizeof(struct Node));
pN->Left  = 0;
pN->Right = 0;
pN->Data  = D;
return pN;
}
//---------------------Прямой обход------------------------------------------
void VisitPre(pNode root, pFunction Function)
{
if (root)
  {
  Function(root);
  VisitPre(root->Left,Function);
  VisitPre(root->Right,Function);
  }
}
//---------------------Обратный обход----------------------------------------
void VisitPost(pNode root, pFunction Function)
{
if (root)
  {
  VisitPost(root->Left,Function);
  VisitPost(root->Right,Function);
  Function(root);
  }
}
//---------------------Вставка узла------------------------------------------
void Insert(pNode * proot, Item D)
{
#define root (*proot)
if(!root)
  root = NewNode(D);
else
  if(D < root->Data)
    Insert(&(root->Left),D);
  else
    Insert(&(root->Right),D);
#undef root
}
//---------------------Поиск узла--------------------------------------------
pNode  Find(pNode root, pNode * parent, int * LR, Item D)
{
*parent = 0;
*LR     = 0;
while (root)
  if(D < root->Data)
    {
    *parent = root;
    *LR     = 0;
    root    = root->Left;
```

```
    }
  else
    if(D > root->Data)
       {
       *parent = root;
       *LR     = 1;
       root    = root->Right;
       }
    else
       return root;
return 0;
}
//---------------------Поиск правого минимального---------------------------
pNode  FindMinRight(pNode root, pNode * parent)
{
while (root->Left)
  {
  *parent = root;
  root   =  root->Left;
  }
return root;
}
//---------------------Удаление одного узла---------------------------------
void Delete(pNode root)
{
if (root) free(root);
}
//---------------------Удаление узла с заданным ключом----------------------
void DeleteNode(pNode * ROOT, Item D)
{
int LR = 0;
pNode parent = 0, root = Find(*ROOT, &parent, &LR, D);
if (root)
  {
  if(root->Left && root->Right)
    {
    pNode MinRightParent = root;
    pNode MinRight       = FindMinRight(root->Right, &MinRightParent);
    MinRight->Left       = root->Left;
    MinRightParent->Left = MinRight->Right;
    MinRight->Right      = root->Right;
    if (parent)
      if (LR) parent->Right = MinRight;
      else    parent->Left  = MinRight;
    else
      *ROOT = MinRight;
    }
  else
    if (parent)
      if (LR) parent->Right = root->Right ? root->Right : root->Left;
      else    parent->Left  = root->Right ? root->Right : root->Left;
    else
      *ROOT = root->Right ? root->Right : root->Left;
  Delete(root);
  }
}
//--------------------------Поворот вправо---------------------------------
void RotateRight(pNode * pLev0)
{
```

```
#define Lev0 (*pLev0)
pNode Lev1  = Lev0->Left;
Lev0->Left  = Lev1->Right;
Lev1->Right = Lev0;
Lev0        = Lev1;
#undef Lev0
}
//---------------------------Поворот влево-----------------------------------
void RotateLeft(pNode * pLev0)
{
#define Lev0 (*pLev0)
pNode Lev1  = Lev0->Right;
Lev0->Right = Lev1->Left;
Lev1->Left  = Lev0;
Lev0        = Lev1;
#undef Lev0
}
//-------------------------Вставка в корень----------------------------------
void InsertRoot(pNode * proot, Item D)
{
#define root (*proot)
if(!root)
  root = NewNode(D);
else
  if(D < root->Data)
    {
    InsertRoot(&(root->Left),D);
    RotateRight(&(root));
    }
  else
    {
    InsertRoot(&(root->Right),D);
    RotateLeft(&(root));
    }
#undef root
}
```