# 10. Динамические структуры данных

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

* линейные списки:
  + - однонаправленные;
    - двунаправленные;
* бинарные деревья;

Они отличаются способом связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки динамической памяти.

Наиболее простой динамической структурой является линейный однонаправленный список, элементами которого служат объекты структурного типа (рис.7).

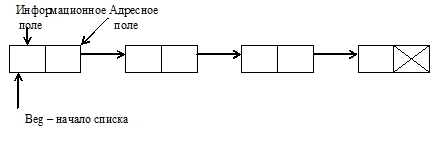


Рис. 1. Линейный однонаправленный список

Описание простейшего элемента такого списка выглядит следующим образом:

class имя\_типа

{

информационное поле;

адресное поле;

}

* информационное поле – это поле любого, ранее объявленного или стандартного, типа;
* адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.

Информационных полей может быть несколько.

//пример 1

class Point

{

int data; //информационное поле

Point next; //адресное поле

};

//пример 2

class Person

{

string name; //информационное поле

int age; //информационное поле

Person next; //адресное поле

};

Каждый элемент списка содержит ключ, который идентифицирует этот элемент. Ключ обычно бывает либо целым числом (пример 1), либо строкой (пример 2).

Над списками можно выполнять следующие операции:

* начальное формирование списка (создание первого элемента);
* добавление элемента в конец списка;
* добавление элемента в начало списка;
* поиск элемента с заданным ключом;
* удаление элемента с заданным ключом;
* удаление элемента с заданным номером;
* добавление элемента с заданным номером;
* и т. д.

Рассмотрим основные операции над списком из примера 1.

## 10.1. Создание элемента списка

Каждый элемент списка содержит как минимум два поля: информационное и адресное. Для создания одного элемента необходимо выделить под него память и заполнить поля элемента. Значение адресного поля можно вести с клавиатуры, в адресное поле записать нулевое значение.

Для удобства работы с элементом списка добавим в класс Point конструктор без параметров, конструктор с параметрами и перегрузим метод ToString() для вывода информационного поля.

class Point

{

public int data;//информационное поле

public Point next;//адресное поле

public Point()//конструктор без параметров

{

data=0;

next=null;

}

public Point(int d)//конструктор с параметрами

{

data=d;

next=null;

}

public override string ToString()

{

return data+" ";

}

}

## 10.2. Создание списка из n элементов

Для того чтобы создать список из n элементов нужно создать первый элемент списка, а затем добавить в него оставшиеся (n-1) элементы. Добавление может осуществляться либо в начало списка, либо в конец списка. При добавлении элемента в начало списка элементы списка будут располагаться в порядке обратном тому, в котором элементы вводились, т. к. тот элемент, который был введен первым, в списке окажется последним.

/\*формирование списка из n элементов путем добавления элементов в начало списка\*/

//метод для формирования одного элемента списка

static Point MakePoint(int d)

{

Point p = new Point(d);

return p;

}

static Point MakeList(int size) //добавление в начало

{

if (size == 0) return null;

Random rnd = new Random();

int info = rnd.Next(0, 11);//формируем инф. поле с помощью ДСЧ

//формируем первый элемент

Console.WriteLine("The element {0} is adding...", info);

Point beg = MakePoint(info); //можно просто вызвать конструктор

//добавляем остальные элементы

for (int i = 1; i < size; i++)

{

info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

Point p = MakePoint(info);

p.next = beg;

beg = p;

}

return beg;

}

Можно добавлять элементы не в начало, а в конец списка, тогда порядок элементов не изменится.

/\*формирование списка из n элементов путем добавления элементов в конец списка\*/

static Point MakeListToEnd(int size) //добавление в конец

{

if (size == 0) return null;

Random rnd = new Random();

int info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

Point beg = MakePoint(info);

Point r = beg;

for (int i = 1; i < size; i++)

{

info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...",info);

Point p = MakePoint(info);

r.next = p;

r = p;

}

return beg;

}

## 10. 3. Перебор элементов списка

Чтобы перебрать элементы списка нужно встать на первый элемент p=beg и переходить от одного элемента к следующему, используя адресное поле next: p=p.next, до тех пор, пока список не закончится, т. е. p не станет равно нулевому значению. При обходе выполняется обработка ключевого поля списка. Рассмотрим перебор элементов списка на примере функции печати.

static void ShowList(Point beg)

{

//проверка наличия элементов в списке

if (beg == null)

{

Console.WriteLine("The List is empty");

return;

}

Point p = beg;

while (p!=null)

{

Console.Write(p);

p = p.next;//переход к следующему элементу

}

Console.WriteLine();

}

## 10. 4. Удаление элемента с заданным номером

Чтобы удалить элемент с заданным номером (например, с номером k) нужно поставить указатель на элемент с номером k-1 и изменить его адресное поле, присвоив ему значение адреса элемента с номером k+1. Затем элемент с номером k удаляется с помощью функции delete (рис.).

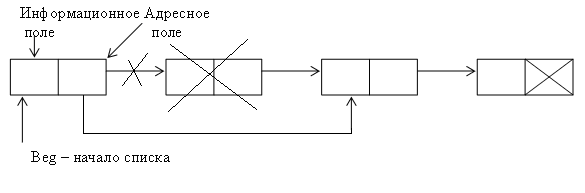


Рис. 2. Удаление элемента из списка

//Удаление из однонаправленного списка элемента с номером k

static Point DelElement(Point beg, int number)

{

if (beg == null)//пустой список

{

Console.WriteLine("Error! The List is empty");

return null;

}

if (number == 1)//удаляем первый элемент

{

beg = beg.next;

return beg;

}

Point p = beg;

//ищем элемент для удаления и встаем на предыдущий

for (int i = 1; i < number - 1 && p != null; i++)

p = p.next;

if (p == null)//если элемент не найден

{

Console.WriteLine("Error! The size of List less than Number");

return beg;

}

//исключаем элемент из списка

p.next = p.next.next;

return beg;

}

Удаление элемента с заданным ключом осуществляется аналогично, но вместо условия (i < number – 1) //проверка номера

нужно использовать условие:

(p.next.data!=KEY)//проверка ключа,

где KEY заданный ключ. Ключ KEY может передаваться как параметр функции удаления.

## 10. 5. Добавление элемента с заданным номером

Для добавления в список элемента с номером k нужно поставить указатель на элемент с номером k-1. Затем нужно создать новый элемент и поменять значения адресных полей таким образом, чтобы адресное поле нового элемента содержало адрес элемента с номером k, а адресное поле k-1 элемента – адрес нового элемента (см. рис.).

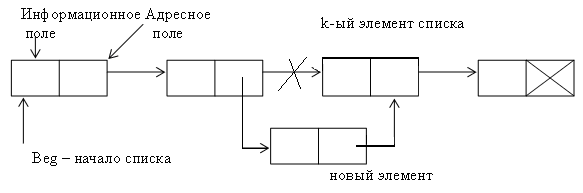


Рис.3. Добавление элемента с номером k

static Point AddPoint(Point beg, int number)

{

Random rnd = new Random();

int info = rnd.Next(10, 100);

Console.WriteLine("The element {0} is adding...", info);

//создаем новый элемент

Point NewPoint = MakePoint(info);

if (beg == null)//список пустой

{

beg = MakePoint(rnd.Next(10, 100));

return beg;

}

if (number == 1) //добавление в начало списка

{

NewPoint.next = beg;

beg = NewPoint;

return beg;

}

//вспом. переменная для прохода по списку

Point p = beg;

//идем по списку до нужного элемента

for (int i = 1; i < number-1 && p != null; i++)

p = p.next;

if (p == null)//элемент не найден

{

Console.WriteLine("Error! The size of List less than Number");

return beg;

}

//добавляем новый элемент

NewPoint.next = p.next;

p.next = NewPoint;

return beg;

}

## 10.6. Двунаправленные списки

Двунаправленный список имеет два адресных поля, которые указывают на следующий элемент списка и на предыдущий. Поэтому двигаться по такому списку можно как слева направо, так и справа налево.

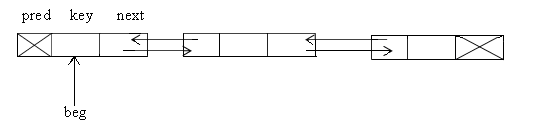


Рис. 4. Двунаправленный список

//пример описания двунаправленного списка

class Point

{

public int data;

public Point next, pred;

public Point()

{

data = 0;

next = null;

pred = null;

}

public Point(int d)

{

data = d;

next = null;

pred = null;

}

public override string ToString()

{

return data + " ";

}

}

Ниже приводится программа, которая создает двунаправленный список, выполняет удаление элемента с заданным номером, добавление элемента с заданным номером и печать полученных списков

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace DoubleList

{

class Point

{

public int data;

public Point next, pred;

public Point()

{

data = 0;

next = null;

pred = null;

}

public Point(int d)

{

data = d;

next = null;

pred = null;

}

public override string ToString()

{

return data + " ";

}

}

class Program

{

static Point MakePoint(int d)

{

Point p = new Point(d);

return p;

}

static Point MakeList(int size) //добавление в начало

{

Random rnd = new Random();

int info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

Point beg = MakePoint(info);

for (int i = 1; i < size; i++)

{

info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

Point p = MakePoint(info);

p.next = beg;

beg.pred = p;

beg = p;

}

return beg;

}

static void ShowList(Point beg)

{

if (beg == null)

{

Console.WriteLine("The List is empty");

return;

}

Point p = beg;

while (p != null)

{

Console.Write(p);

p = p.next;

}

Console.WriteLine();

}

static Point AddPoint(Point beg, int number)

{

Random rnd = new Random();

int info = rnd.Next(10, 100);

Console.WriteLine("The element {0} is adding...", info);

Point NewPoint = MakePoint(info);

if (beg == null)//список пустой

{

beg = MakePoint(rnd.Next(10, 100));

return beg;

}

if (number == 1) //добавление в начало списка

{

NewPoint.next = beg;

beg.pred = NewPoint;

beg = NewPoint;

return beg;

}

Point p = beg;

for (int i = 1; i < number-1 && p != null; i++) p = p.next;

if (p == null)

{

Console.WriteLine("Error! The size of List less than Number");

return beg;

}

NewPoint.next = p.next;

NewPoint.pred = p;

p.next = NewPoint;

if(NewPoint.next!=null)//не последний

NewPoint.next.pred = NewPoint;

return beg;

}

static Point DelElementByNumber(Point beg, int number)

{

if (beg == null)//пустой список

{

Console.WriteLine("Error! The List is empty");

return null;

}

if (number == 1)//удаляем первый элемент

{

beg = beg.next;

if (beg != null)

beg.pred = null;

return beg;

}

Point p = beg;

//ищем элемент для удаления

for (int i = 1; i < number && p != null; i++) p = p.next;

if (p == null)//если элемент не найден

{

Console.WriteLine("Error! The size of List less than Number");

return beg;

}

//исключаем элемент из списка

p.pred.next = p.next;

if (p.next != null)

p.next.pred = p.pred;

return beg;

}

static void PrintMenu()

{

Console.WriteLine("1. Создать список.");

Console.WriteLine("2. Напечатать список.");

Console.WriteLine("3. Удалить элемент по номеру.");

Console.WriteLine("4. Добавить элемент");

Console.WriteLine("5. Выход");

}

static void Main(string[] args)

{

bool ok = true;

int answer = 0, number = 0;

int size = 0;

Point beg = null;

do

{

PrintMenu();

do

{

ok = Int32.TryParse(Console.ReadLine(), out answer);

} while (!ok);

switch (answer)

{

case 1: //создать список

{

Console.WriteLine("Введите количество элемнтов списка");

do

{

ok = Int32.TryParse(Console.ReadLine(), out size);

} while (!ok);

beg = MakeList(size);

ShowList(beg);

break;

}

case 2:

{

ShowList(beg);

break;

}

case 3:

{

Console.WriteLine("Введите номер удаляемого элемента");

do

{

ok = Int32.TryParse(Console.ReadLine(), out number);

} while (!ok);

beg = DelElementByNumber(beg, number);

ShowList(beg);

break;

}

case 4:

{

Console.WriteLine("Введите номер добавляемого элемента");

do

{

ok = Int32.TryParse(Console.ReadLine(), out number);

} while (!ok);

beg = AddPoint(beg, number);

ShowList(beg);

break;

}

}

} while (answer != 5);

}

}

}

## 10. 7. Бинарные деревья

Бинарное дерево - это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных не более двух ссылок на другие бинарные деревья (поддеревья). На каждое поддерево имеется ровно одна ссылка (рис. )

Описать такую структуру можно следующим образом:

class Point

{

public int data;

public Point left, right;

public Point()

{

data = 0;

left = null;

right = null;

}

public Point(int d)

{

data = d;

left = null;

right = null;

}

public override string ToString()

{

return data + " ";

}

}

Начальный узел называется конем дерева. Узел, не имеющий поддеревьев, называется листом дерева. Исходящие листы называются предками, входящие – потомками. Высота дерева определяется количеством уровней, на которых располагаются узлы.

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева - больше называется ***деревом поиска***. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения в каждом узле. Такой поиск является более эффективным, чем поиск по линейному списку, т. к. время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму количества узлов (аналогично бинарному поиску).

В ***идеально сбалансированном дереве*** количество узлов в справа и слева отличается не более, чем на единицу.

Линейный список можно представить как вырожденное бинарное дерево, в котором каждый узел имеет не более одной ссылки. Для списка среднее время поиска равно половине длины списка.

Деревья и списки являются рекурсивными структурами данных, т. к. каждое поддерево также является деревом. Дерево можно определить как рекурсивную структуру, в которой

каждый элемент является:

* либо пустой структурой;
* либо элементом, с которым связано конечное число поддеревьев.

Действия с рекурсивными структурами удобнее всего описываются с помощью рекурсивных алгоритмов.

## 10.8. Обход дерева

Для того, чтобы выполнить определенную операцию над всем узлами дерева, надо обойти все узлы. Такая задача называется обходом дерева. При обходе узлы должны посещаться в определенном порядке. Существуют три принципа упорядочивания, которые естественно вытекают из структуры дерева.

Рассмотрим следующее дерево.

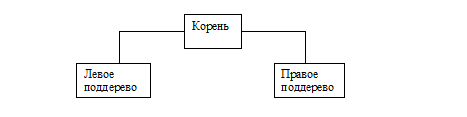


Рис.. Структура бинарного дерева

На этом дереве можно определить три метода упорядочения:

* Слева направо: Левое поддерево – Корень – Правое поддерево;
* Сверху вниз: Корень – Левое поддерево – Правое поддерево;
* Снизу вверх: Левое поддерево – Правое поддерево – Корень.

Эти три метода можно сформулировать в виде рекурсивных алгоритмов. Эти алгоритмы служат примером того, что действия с рекурсивными структурами лучше всего описываются рекурсивными алгоритмами.

//обход слева направо

void Run(Point p)

{

if(p!=null)

{

Run(p.left);//переход к левому поддереву

<обработка p.data>

Run(p.right);//переход к правому поддереву

}

}

Если в качестве операции обработки узла поставить операцию вывода информационного поля узла, то мы получим функцию для печати дерева.

Для дерева поиска на рис. обход слева направо даст следующие результаты:

1 3 5 7 8 9 12

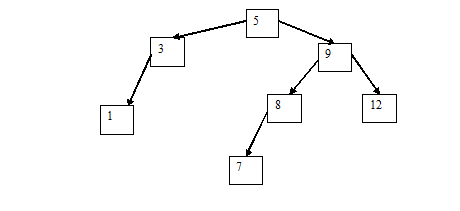


Рис. Пример дерева поиска

//Обход сверху вниз

void Run(Point p)

{

if(p!=null)

{

<обработка p.data>

Run(p.left);//переход к левому поддереву

Run(p.right);//переход к правому поддереву

}

}

Результаты обхода сверху вниз для дерева поиска на рис. :

5 3 1 9 8 7 12

//Обход снизу вверх

void Run(Point p)

{

if(p!=null)

{

Run(p.left);//переход к левому поддереву

Run(p.right);//переход к правому поддереву

<обработка p.data>

}

}

Результаты обхода снизу вверх для дерева поиска на рис. :

1 3 7 8 12 9 5

## 10.9. Формирование дерева

Рассмотрим построение идеально сбалансированного дерева. При построении такого дерева надо распределять узлы таким образом, чтобы количество узлов в левом и правом поддеревьях отличалось не более чем на единицу.

//вспомогательная функция для ввода информации

static int GetInfo()

{

int info = rnd.Next(0, 100);

Console.WriteLine("The element {0} is adding...", info);

return info;

}

//формирование идеально-сбалансированного дерева

static Point IdealTree(int size, Point p)

{

Point r;

int nl, nr;

if(size==0){p=null;return p;}

nl=size/2;

nr=size-nl-1;

int d = GetInfo();

r = new Point(d);

r.left=IdealTree(nl,r.left);

r.right=IdealTree(nr,r.right);

p=r;

return p;

}

//печать дерева по уровням

static void ShowTree(Point p, int l)

{

if(p!=null)

{

ShowTree(p.left,l+3);//переход к левому поддереву

for (int i = 0; i < l; i++) Console.Write(" ");

Console.WriteLine(p.data);

ShowTree(p.right,l+3);//переход к правому поддереву

}

}

static void Main(string[] args)

{

int size = 5;

Point idTree=null;

idTree= IdealTree(size, idTree);

ShowTree(idTree, 5);

}

Функция печати, которая используется в данной программе, печатает дерево по уровням слева направо. Переменная l считает количество пробелов, которые надо отступить для печати следующего уровня. Результат работы программы представлен на рис.17.

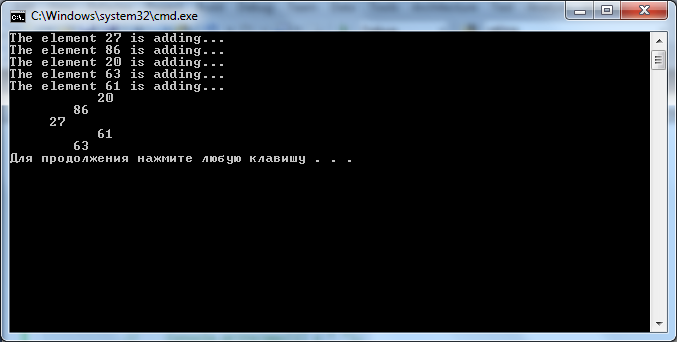


Рис. Результат работы программы формирования и печати идеально сбалансированного дерева

При формировании дерева поиска нужно учитывать упорядоченность элементов в таком дереве, т. е. добавление элемента с ключом больше текущего осуществляется в правое поддерево, а меньше текущего – в левое. Ключи не дублируется, поэтому необходимо проверить существует ли элемент с заданным ключом в дереве и если существует, то завершить функцию добавления элемента.

static Point MakePoint(int d)//формирование элемента дерева

{

Point p=new Point(d);

return p;

}

//добавление элемента d в дерево поиска

static Point Add(Point root, int d)

{

Point p = root;//корень дерева

Point r = null;

//флаг для проверки существования элемента d в дереве

bool ok = false;

while (p!=null && !ok)

{

r = p;

if (d == p.data) ok = true;//элемент уже существует

else

if (d < p.data) p = p.left;//пойти в левое поддерево

else p = p.right; //пойти в правое поддерево

}

if (ok) return p;//найдено, не добавляем

//создаем узел

Point NewPoint = MakePoint(d);//выделили память

// если d<r->key, то добавляем его в левое поддерево

if (d < r.data) r.left = NewPoint;

// если d>r->key, то добавляем его в правое поддерево

else r.right = NewPoint;

return NewPoint;

}

## 10.10. Обработка деревьев с помощью рекурсивного обхода

**Задача 1**. Найти количество четных элементов в дереве.

Для решения этой задачи необходимо перебрать все элементы дерева и проверить информационные поля на четность. Для перебора будем использовать обход дерева слева направо. Результат запоминаем в специальной переменной, которую передаем по ссылке.

//количество четных элементов в дереве

public static void EvenCount(Point p, ref int rez)

{

if(p!=null)

{

EvenCount(p.left, ref rez);

if(p.data%2==0 &&p.data!=0)rez++;

EvenCount(p.right, ref rez);

}

}