МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИНСТИТУТ МАТЕМАТИКИ И МЕХАНИКИ ИМ. Н.И. ЛОБАЧЕВСКОГО

Кафедра теории функций и приближений

С. В. МАКЛЕЦОВ

ПРОГРАММНЫЕ МОДЕЛИ НА ОСНОВЕ ДИНАМИЧЕСКИХ СТРУКТУР ДАННЫХ

Учебное пособие

Казань — 2021

Печатается по решению методической комиссии
Института математики и механики им. Н.И. Лобачевского

Рецензент:

кандидат физико-математических наук,
доцент кафедры теории функций и приближений КФУ **Р. Г. Насибуллин**.

Маклецов С. В.
Программные модели на основе динамических структур данных: учебное пособие / С. В. Маклецов. — Казань: Казанский (Приволжский) федеральный университет, 2021. — 108 с.

В настоящем пособии рассматриваются методы построения программных моделей на основе динамических структур данных, таких как стеки, очереди, одно- и двунаправленные списки, деревья. Издание в первую очередь предназначено для бакалавров 1 курса Института математики и механики им. Н. И. Лобачевского Казанского (Приволжского) федерального университета, обучающихся по направлениям 02.03.01 — «Математика и компьютерные науки» и 01.03.01 — «Математика», но также может быть полезно и для других студентов, изучающих дисциплины компьютерного цикла. Пособие включает теоретические сведения по изучаемым темам, примеры разобранных практических заданий, реализованные на языке программирования C++, контрольные вопросы и задания для самостоятельного выполнения, а также список рекомендуемой литературы.

© Казанский федеральный университет, 2021

© Маклецов С.В., 2021

ОГЛАВЛЕНИЕ

Введение 4
Общие сведения о динамических структурах данных 5
Базовые программные конструкции 7
Стек 9
Разбор архитектуры стека и пример реализации 9
Контрольные вопросы и задания для самостоятельного выполнения 19
Очередь 21
Разбор архитектуры очереди и пример реализации 21
Контрольные вопросы и задания для самостоятельного выполнения 28
Список 30
Базовая архитектура списков 30
Работа с односвязным списком 30
Работа с двусвязным списком 38
Контрольные вопросы и задания для самостоятельного выполнения 46
Дерево 47
Базовая архитектура 47
Бинарные деревья поиска 48
Обход деревьев 59
Обход дерева вглубь 60
Обход дерева вширь 69
Балансировка бинарного дерева (АВЛ-деревья) 78
Бинарное дерево выражений 91
Дерево общего вида 93
Контрольные вопросы и задания для самостоятельного выполнения 104
Рекомендуемая литература 107

ВВЕДЕНИЕ

Целью настоящего пособия является получение как теоретических сведений о понятии, классификации, формировании и особенностях оперирования данными, находящихся в динамических структурах, так и практических навыков решения задач с использованием динамических структур в языке C/C++.

Пособие содержит детальное описание устройства динамических структур данных (структур, память для которых выделяется и освобождается по мере необходимости) и построенных на их базе программных моделей. При этом рассматривается именно технология их формирования, а не применение готовых конструкций, присутствующих в стандартной библиотеке языка C++.

Для успешного освоения материала и приобретения практических навыков программирования необходимо воспользоваться любым компилятором языка C/C++ и средой разработки программ. В качестве таковых рекомендуются (альтернативно):

- MS Visual Studio 2019 или выше (версия «Community» является бесплатной для некоммерческого использования);
- компилятор языка C++ MinGW (распространяется свободно) в совокупности со средой разработки CLion компании JetBrains (бесплатно для студентов Университета при условии некоммерческого использования).

ОБЩИЕ СВЕДЕНИЯ О ДИНАМИЧЕСКИХ СТРУКТУРАХ ДАННЫХ

Динамические структуры данных *(ДСД) – это данные, внутреннее строение которых подчиняется определенному закону, но количество элементов, их расположение и взаимосвязи могут динамически измениться во время выполнения программы*. Задачи, для решения которых объем требуемой памяти заранее неизвестен, являются весьма распространенными. В таком случае возникает необходимость в распределении памяти блоками в процессе выполнения программы.

Динамические структуры данных могут основываться на выделении фиксированных блоков памяти, как в массивах. Такой подход позволяет облегчить доступ к элементам такой структуры, однако в большинстве случаев приводит к нерациональному использованию памяти. В настоящем пособии будут рассмотрены динамические структуры, основанные на применении отдельных блоков, связанных между собой при помощи указателей.

Как правило для динамической структуры характерно отсутствие жестко закрепленного за ней имени, нефиксированное число элементов, переменные размерность и характер взаимодействия элементов.

Необходимость в динамических структурах данных обычно возникает в случаях, когда:

- в программе нужно сохранять большие объемы информации;
- конкретное количество обрабатываемых значений труднопредсказуемо;
- требуется особым образом построить работу с имеющимся набором данных, например, сохранять тот или иной порядок значений, оптимизировать поиск, хранить сложноорганизованные сведения.

Обычно для получения доступа к структуре используется один или пара указателей, содержащих адреса памяти, в которых расположены ключе-

вые ее элементы (первый, последний, корневой и т.п.). Для получения доступа ко всем остальным элементам, которые в отличие от случая с массивом, могут располагаться в произвольных участках памяти (не обязательно смежных), используются связи, также формируемые при помощи указателей, сопоставляемых с каждым элементом.

Таким образом, каждый элемент структуры должен содержать не только хранимое значение, но и адреса одного или большего числа смежных элементов. Это обеспечивается применением составных типов данных, таких, например, как структура или класс.

Плюсом такого подхода является возможность обеспечения значительных изменений в структуре и количестве элементов, максимально возможным использованием фрагментированной памяти. Тем не менее, оно не лишено и недостатков, среди которых – низкая скорость произвольного доступа к элементам в сравнении с массивами, а также использование дополнительной памяти для хранения связей.

Существует несколько различных видов динамических структур данных (ДСД). Чаще всего они подразделяются на линейные и нелинейные.

Линейные ДСД – это структуры, в которых связи между элементами не зависят от выполнения каких-либо условий.

Наиболее распространенными линейными ДСД являются:

- список (односвязный и двусвязный);
- стек;
- очередь (и ее разновидности, такие как, дек).

Нелинейные ДСД – это структуры, в которых расположение элементов зависит от определенных условий. К нелинейным динамическим структурам данных обычно относят:

- деревья;
- графы;
- многосвязные списки.

БАЗОВЫЕ ПРОГРАММНЫЕ КОНСТРУКЦИИ

В настоящем пособии будут рассматриваться конструкции, построенные на базе структур.

***Структура**, как элемент языка программирования, – это составной пользовательский тип данных*. Структура может содержать несколько переменных, тип которых может быть почти любым. То есть программист может сам определить состав структуры в программе. Существует лишь одно ограничение: в структуре не могут находиться переменные типа самой структуры (однако к указателям это исключение не относится).

Для объявления структуры в языках Си/Си++ используется ключевое слово **`struct`**, следом за котором указывается имя структуры и, далее, список переменных (полей структуры), в фигурных скобках. Заканчивается описание точкой с запятой.

ОТМЕТИМ, ЧТО В СОСТАВЕ СТРУКТУР МОГУ БЫТЬ РАСПОЛОЖЕНЫ НЕ ТОЛЬКО ПЕРЕМЕННЫЕ (ПОЛЯ), НО И ФУНКЦИИ (МЕТОДЫ). ОДНАКО В НАСТОЯЩЕМ ПОСОБИИ ТАКАЯ ВОЗМОЖНОСТЬ ИСПОЛЬЗОВАТЬСЯ ПРАКТИЧЕСКИ НЕ БУДЕТ, ПОСКОЛЬКУ ОНА БОЛЕЕ ХАРАКТЕРНА ДЛЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ, РАССМОТРЕНИЕ КОТОРОГО ВЫХОДИТ ЗА РАМКИ ДАННОГО ПОСОБИЯ.

Пример описания и использования переменных типа структуры приведен в следующей программе.

```
1   #include <iostream>
2   using namespace std;
3
4   const int maxLen = 20;
5
6   //Описание структуры Person
7   struct Person{
8         unsigned long id;
9         char fio[maxLen];
10        unsigned int age;
11  };
12
13  void main()
14  {
15        setlocale(LC_ALL, "Rus");
16        //Создание массива структур
17        Person p[3];
18        //Инициализация элемента типа структуры целиком
```

```
19          p[0] = {1, "Иванов И. И.", 35};
20
21          //Заполнение значений по отдельным полям.
22          //Для доступа к полям через переменную типа структуры
23          //используется символ "."
24          p[1].id = 2;
25          p[1].age = 22;
26          strcpy_s(p[1].fio, maxLen, "Петрова А. Г.");
27
28          //Работа со структурой через указатель
29          Person* p2 = &p[2];
30          //Для доступа к полям структуры при использовании указателя применяется стрелка "->"
31          p2->id = 3;
32          strcpy_s(p2->fio, maxLen, "Гусев К. Б.");
33          p2->age = 45;
34
35          //Форматированный вывод данных из массива:
36          for (int i = 0; i < sizeof(p)/sizeof(Person); i++){
37              printf("%d. %-20s (%d).\n",p[i].id,p[i].fio,p[i].age);
38          }
39
40          system("pause");
41      }
```

Результат работы программы:

```
1. Иванов И. И.          (35).
2. Петрова А. Г.         (22).
3. Гусев К. Б.           (45).
Для продолжения нажмите любую клавишу . . .
```

В общем случае для работы с динамической структурой данных необходимо:

1) сформировать структурированный тип для хранения данных (в информационных полях) и указателей на другие элементы списка (в адресных полях);
2) создать указатели на один или несколько особых (начальных, конечных, корневых и т.п.) ее элементов;
3) реализовать функции для работы с ДСД (а именно, для добавления, удаления, поиска элементов и т.д.).

Ниже будут рассмотрены некоторые, наиболее распространенные динамические структуры данных и методы организации работы с ними.

СТЕК

РАЗБОР АРХИТЕКТУРЫ СТЕКА И ПРИМЕР РЕАЛИЗАЦИИ

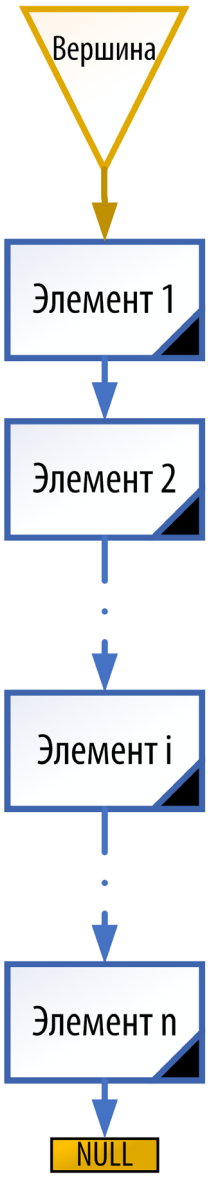


Рис. 1. Стек

Стек *– это линейный список (как правило, односвязный), в котором все элементы добавляются и удаляются только с одного конца, который называется вершиной.*

Стек реализует принцип LIFO (last-in-first-out), то есть элемент, размещаемый в списке в последнюю очередь, забирается оттуда первым. При этом технически устройство стека полностью соответствует конструкции списка, которая также будет рассматриваться ниже.

Для работы со стеком, как правило, используются следующие основные функции:

- **`push`** (помещение нового элемента в стек);
- **`pop`** (извлечение элемента из стека с возвратом значения);
- **`peek`** (получение значения элемента, расположенного на вершине стека, без удаления последнего).

Технически, стек представляет собой одну из самых простых динамических структур данных. Ниже представлен пример программы, содержащей базовые функции для работы со стеком и вариант их использования.

В программе объявлена структура **`elem`**, которая использована как основа для хранения данных элементов стека.

Файл «stack.h» содержит описание структуры элементов и функций для работы со стеком.

```
1   #pragma once
2
3   /// <summary>
4   /// Структура для хранения элементов стека
5   /// </summary>
6   struct elem
7   {
8           /// <summary>
9           /// информационное поле стека
10          /// </summary>
11          int value;
12
13          /// <summary>
14          /// адресное поле стека
15          /// </summary>
16          elem* next = nullptr;
17  };
18
19  /// <summary>
20  /// Добавление элемента в стек
21  /// </summary>
22  /// <param name="stack">
23  /// Стек, в который происходит добавление элемента.
24  /// Вершина стека изменяется после добавления элемента, поэтому параметр передается по ссылке.
25  /// </param>
26  /// <param name="value">Добавляемое в стек значение</param>
27  void push(elem*& stack, int value);
28
29  /// <summary>
30  /// Удаление элемента из стека с возвратом хранимого значения
31  /// </summary>
32  /// <param name="stack">
33  /// Стек, из которого нужно изъять значение
34  /// Вершина стека изменяется после изъятия элемента, поэтому параметр передается по ссылке.
35  /// </param>
36  /// <param name="value">Параметр заполняется значением, находившимся на вершине стека </param>
37  /// <returns>
38  /// true, в случае если стек не был пуст и значение успешно получено
39  /// false - в противном случае
40  /// </returns>
41  bool pop(elem*& stack, int& value);
42
43  /// <summary>
44  /// Получение значения с вершины стека, без удаления самого элемента
45  /// </summary>
46  /// <param name="stack">
47  /// Стек, с вершины которого будет получено значение.
48  /// При вызове метода стек не меняется, поэтому параметр объявлен константным.
49  /// </param>
50  /// <returns>
51  /// Возвращается константный указатель на информационное поле верхнего элемента стека
52  /// </returns>
53  const int* peek(const elem* stack);
54
```

```
55  /// <summary>
56  /// Получает следующий элемент стека
57  /// </summary>
58  /// <param name="elem">Элемент, относительно которого ищется следующий </param>
59  /// <returns>
60  /// Указатель на элемент стека либо nullptr, если в стеке больше нет элементов
61  /// </returns>
62  elem* next_elem(const elem* elem);
63
64  /// <summary>
65  /// Получает последний элемент стека
66  /// </summary>
67  /// <param name="elem">Элемент стека, в котором нужно найти последний элемент</param>
68  /// <returns>Последний элемент, расположенный в стеке</returns>
69  elem* last(const elem* elem);
70
71  /// <summary>
72  /// Удаление всего стека
73  /// </summary>
74  /// <param name="stack">
75  /// Удаляемый стек. Указатель на стек изменяется, поэтому параметр передается по ссылке.
76  /// </param>
77  void clear(elem*& stack);
```

Обратите внимание на стиль созданных комментариев. Такой особый синтаксис использован для создания документации к исходным кодам. Документация в Visual Studio генерируется автоматически, когда программист использует три слеша подряд (**///**) перед документируемой конструкцией.

Теги **<summary>**...**</summary>** внутри комментария-документации содержат общее описание объекта (функции, структуры или переменной).

Тег **<param>**...**</param>** с дополнительным атрибутом **name** используется для создания описания параметра функции с указанным именем.

Наконец, тег **<returns>**...**</returns>** используется для описания значения, возвращаемого функций.

Наличие таких комментариев позволяет среде программирования выводить подсказки при использовании описанных функций, структур и переменных в соответствии с имеющейся документацией.

Файл «stack.cpp» с реализацией функций для работы со стеком.

```
1   #include "stack.h"
2
3   void push(elem*& stack, int value)
4   {
5         //Создание нового элемента для размещения в стеке
6         elem* newel = new elem;
7         newel->value = value;
8         //Проверка пустоты стека
9         if (!stack)
10        {
```

```
11              //Новый элемент будет первым
12              stack = newel;
13      } else
14      {
15              //Стек уже существует. Новый элемент размещается на вершине
16              newel->next = stack;
17              stack = newel;
18      }
19  }
```

Обратите внимание, что добавление элемента выполняется по-разному, в зависимости от того, является ли он первым в стеке, или перед выполнением этого действия стек уже существовал. Подобные проверки будут применяться и при создании ДСД других видов.

```
20
21  bool pop(elem*& stack, int& value)
22  {
23      // Извлекаем элемент из стека, если он не пуст
24      if (!stack) return false; // Признак того, что значение не возвращено
25      elem* rem = stack;// Сохраняем ссылку на вершину
26      value = stack->value;// и значение, хранимое в верхнем элементе
27      stack = stack->next;// Смещаем указатель на новую вершину
28      delete rem;
29      return true;// Признак успешности получения значения с вершины стека
30  }
31
32  const int* peek(const elem* stack)
33  {
34      //Получаем значение с вершины стека, если он не пуст без удаления самого элемента
35      if (!stack) return nullptr;
36      return &stack->value;
37  }
38
39  elem* next_elem(const elem* elem)
40  {
41      if (elem) return elem->next;
42      return nullptr;
43  }
44
45  elem* last(const elem* el)
46  {
47      if (el) while (el->next) el = el->next;
48      return (elem*)el;
49  }
50
51  void clear(elem*& stack)
52  {
53      while (stack)// Удаляем все элементы стека
54      {
55          elem* rem = stack;
56          stack = stack->next;
57          delete rem;
58      }
59  }
```

Файл «main.cpp», содержит код, демонстрирующий работу функций, приведенных выше. В качестве примера стек заполняется десятью подряд идущими натуральными числами, затем его содержание выводится на экран. При этом последовательность «переворачивается».

```
#include <iostream>
using namespace std;

#include "stack.h"

/// <summary>
/// Вспомогательная функция заполнения стека подряд идущими натуральными числами
/// </summary>
/// <param name="stack">Указатель на вершину заполняемого стека</param>
void fill_stack_by_sequence(elem*& stack)
{
	for (int i = 1; i<=10; i++)
	{
		push(stack, i);
	}
}

/// <summary>
/// Вспомогательная функция вывода содержимого стека на экран
/// </summary>
/// <param name="stack">Указатель на вершину стека</param>
void show_stack(const elem* stack)
{
	// Получаем указатель на вершину стека (необходим для перемещения по элементам стека)
	auto curr = (elem*)stack;
	// Определяем последний элемент
	// (он потребуется для корректного вывода символа-разделителя)
	auto lst = last(stack);
	while (curr)
	{
		// Получение значения текущего элемента стека
		auto val = peek(curr);
		// Выводится элемент и символ-разделитель
		// (пробел, если в стеке еще есть элементы или переход
		// на новую строку, если элемент был последним).
		cout << *val << (curr == lst ? "\n" : " ");
		curr = next_elem(curr);
	}
}

void main()
{
	elem* stack = nullptr;
	// Для последовательности 1 2 3 4 5 6 7 8 9 10
	fill_stack_by_sequence(stack);
	show_stack(stack); //10 9 8 7 6 5 4 3 2 1
	clear(stack);
	system("pause");
}
```

Данное приложение демонстрирует работу стека, способного хранить исключительно целые числа. Однако существует возможность создать более универсальные динамические структуры данных, пригодные для хранения элементов практически любых типов.

С этой целью необходимо создать шаблонные функции, разместив их заголовки и реализацию в заголовочном файле «stack.h». Рассмотрим изменения, которые претерпит приведенная выше программа, при использовании шаблонов.

Напомним, что шаблонная функция (или структура) получается в результате размещения перед ней ключевого слова **`template`** с последующим указанием имени шаблонного типа в угловых скобках после слова **`typename`** (либо **`class`**). В этом случае формируется не функция, а инструкция для компилятора по созданию перегруженных функций для разных типов, подставляемых вместо шаблонного. При этом компилятор сам определяет по вариантам вызова функций какие именно варианты перегрузок будут сформированы. Следовательно, необходимо, чтобы вызов, объявление и реализация шаблонной функции находились внутри одной единицы компиляции. Именно поэтому весь код из файла «stack.cpp» переносится в «stack.h», который подключается инструкцией **`#include`** к файлу, содержащему вызовы функций.

Обратите внимание на особенности обозначения модифицируемых и повторяющихся по сравнению с предыдущим вариантом программы частей кода в последующих блоках.

Серым фоном отмечены строки, которые не претерпели смысловых изменений по сравнению с первой версией программы (не считая смены расположения и нумерации).

Строки, которые по сравнению с предыдущей версией изменяются или удаляются, в тексте программы перечеркнуты; также у них отсутствует нумерация.

Файл «stack.h» примет вид.

```
1   #pragma once
2
3   /// <summary>
4   /// Структура для хранения элементов стека
5   /// </summary>
6   template <typename T>
7   struct elem
8   {
9           /// <summary>
10          /// Информационное поле стека
11          /// </summary>
            int value;
12          T value;
13          /// <summary>
14          /// Адресное поле стека
15          /// </summary>
16          elem* next = nullptr;
17  };
18
19  /// <summary>
20  /// Добавление элемента в стек
21  /// </summary>
22  /// <param name="stack">
23  /// Стек, в который происходит добавление элемента.
24  /// Вершина стека изменяется после добавления элемента, поэтому параметр передается по ссылке.
25  /// </param>
26  /// <param name="value">Добавляемое в стек значение</param>
            void push(elem*& stack, int value);
27  template <typename T>
28  void push(elem<T>*& stack, T value);
29
30  /// <summary>
31  /// Удаление элемента из стека с возвратом хранимого значения
32  /// </summary>
33  /// <param name="stack">
34  /// Стек, из которого нужно изъять значение
35  /// Вершина стека изменяется после изъятия элемента, поэтому параметр передается по ссылке.
36  /// </param>
37  /// <param name="value">Параметр заполняется значением, находившимся на вершине стека</param>
38  /// <returns>
39  /// true, в случае если стек не был пуст и значение успешно получено
40  /// false - в противном случае
41  /// </returns>
            bool pop(elem*& stack, int& value);
42  template <typename T>
43  bool pop(elem<T>*& stack, T& value);
44
45  /// <summary>
46  /// Получение значения с вершины стека, без удаления самого элемента
47  /// </summary>
48  /// <param name="stack">
49  /// Стек, с вершины которого будет получено значение.
50  /// При вызове метода стек не меняется, поэтому параметр объявлен константным.
51  /// </param>
52  /// <returns>
53  /// Возвращается константный указатель на информационное поле верхнего элемента стека
```

```
54  /// </returns>
    const int* peek(const elem* stack);
55  template <typename T>
56  const T* peek(const elem<T>* stack);
57
58  /// <summary>
59  /// Получает следующий элемент стека
60  /// </summary>
61  /// <param name="elem">Элемент, относительно которого ищется следующий </param>
62  /// <returns>
63  /// Указатель на элемент стека либо nullptr, если в стеке больше нет элементов
64  /// </returns>
    elem* next_elem(const elem* elem);
65  template <typename T>
66  elem<T>* next_elem(const elem<T>* elem);
67
68  /// <summary>
69  /// Получает последний элемент стека
70  /// </summary>
71  /// <param name="elem">Элемент стека, в котором нужно найти последний элемент</param>
72  /// <returns>Последний элемент, расположенный в стеке</returns>
    elem* last(const elem* elem);
73  template <typename T>
74  elem<T>* last(const elem<T>* elem);
75
76  /// <summary>
77  /// Удаление всего стека
78  /// </summary>
79  /// <param name="stack">
80  /// Удаляемый стек. Указатель на стек изменяется, поэтому параметр передается по ссылке.
81  /// </param>
    void clear(elem*& stack);
82  template <typename T>
83  void clear(elem<T>*& stack);
84  /// ————————————————————————————————————————————————
85
    void push(elem*& stack, int value);
86  template <typename T>
87  void push(elem<T>*& stack, T value)
88  {
89      // Создание нового элемента для размещения в стеке
        auto newel = new elem;
90      auto newel = new elem<T>;
91      newel->value = value;
92      // Проверка пустоты стека
93      if (!stack)
94      {
95          // Новый элемент будет первым
96          stack = newel;
97      } else
98      {
99          // Стек уже существует. Новый элемент размещается на вершине
100         newel->next = stack;
101         stack = newel;
102     }
103 }
104
    bool pop(elem*& stack, int& value);
```

```
105 template <typename T>
106 bool pop(elem<T>*& stack, T& value)
107 {
108       // Извлекаем элемент из стека, если он не пуст
109       if (!stack) return false; // Признак того, что значение не возвращено
110       auto rem = stack; // Сохраняем ссылку на вершину
111       value = stack->value; // И значение, хранимое в верхнем элементе
112       stack = stack->next; // Смещаем указатель на новую вершину
113       delete rem;
114       return true; // Признак успешности получения значения с вершины стека
115 }
116
    const int* peek(const elem* stack);
117 template <typename T>
118 const T* peek(const elem<T>* stack)
119 {
120       // Получаем значение с вершины стека, если он не пуст без удаления самого элемента
121       if (!stack) return nullptr;
122       return &stack->value;
123 }
124
    elem* next_elem(const elem* elem)
125 template <typename T>
126 elem<T>* next_elem(const elem<T>* elem)
127 {
128       if (elem) return elem->next;
129       return nullptr;
130 }
131
    elem* last(const elem* el)
132 template <typename T>
133 elem<T>* last(const elem<T>* el)
134 {
135       while (el && el->next) el = el->next;
          return (elem*)el;
136       return (elem<T>*)el;
137 }
138
    void clear(elem*& stack)
139 template <typename T>
140 void clear(elem<T>*& stack)
141 {
142       while (stack) // Удаляем все элементы стека
143       {
144             auto rem = stack;
145             stack = stack->next;
146             delete rem;
147       }
148 }
```

Файл «main.cpp».

```
1 #include <iostream>
2 using namespace std;
3
4 #include "stack.h"
5
```

```
6   /// <summary>
7   /// Вспомогательная функция заполнения стека подряд идущими натуральными числами
8   /// </summary>
9   /// <param name="stack">Указатель на вершину заполняемого стека</param>
    void fill_stack_by_sequence(elem*& stack)
10  template <typename T>
11  void fill_stack_by_sequence(elem<T>*& stack)
12  {
13      for (int i = 1; i<=10; i++)
14      {
15          push(stack, i);
16      }
17  }
18
19  /// <summary>
20  /// Вспомогательная функция вывода содержимого стека на экран
21  /// </summary>
22  /// <param name="stack">Указатель на вершину стека</param>
    void show_stack(const elem* stack)
23  template <typename T>
24  void show_stack(const elem<T>* stack)
25  {
        auto curr = (elem*)stack;
26      auto curr = (elem<T>*)stack;
27      while (curr)
28      {
29          // Выводится элемент и символ-разделитель
30          // (пробел, если в стеке еще есть элементы или переход
31          // на новую строку, если элемент был последним).
32          cout << *peek(curr) << (curr->next ? " " : "\n");
33          curr = curr->next;
34      }
35  }
36
37  void main()
38  {
39      setlocale(LC_ALL, "Rus");
        elem* stack = nullptr;
40      elem<int>* stack = nullptr;
41      // Для последовательности 1 2 3 4 5 6 7 8 9 10
42      fill_stack_by_sequence(stack);
43      show_stack(stack); //10 9 8 7 6 5 4 3 2 1
44      clear(stack);
45
46      // Ту же структуру и разработанные функции можно успешно использовать
47      // для хранения данных совершенно иных типов
48      elem<const char*>* stack2 = nullptr;
49      // Для последовательности строк «ЭТОТ», «СТЕК», «ЗАПОЛНЕН», «СЛОВАМИ»...
50      push(stack2, "ЭТОТ");
51      push(stack2, "СТЕК");
52      push(stack2, "ЗАПОЛНЕН");
53      push(stack2, "СЛОВАМИ");
54      show_stack(stack2); // ...будет выведено «СЛОВАМИ ЗАПОЛНЕН СТЕК ЭТОТ»
55      clear(stack2);
56      system("pause");
57  }
```

Контрольные вопросы и задания для самостоятельного выполнения

1. В чем заключаются достоинства и недостатки последовательного (как в массивах) и связанного способов реализации динамических структур данных?
2. Каковы основные принципы функционирования стека?
3. Придумайте и приведите примеры реальных задач, для моделирования которых удобно использовать стек.

---

4. Создать стек, каждый элемент которого является парой двух целых чисел. Заменить все элементы, сумма значений которых равна определенному значению, на нулевые.
5. Дан стек, содержащий целые числа. Разделить стек на два, содержащих четные и нечетные числа, сохранив порядок элементов исходного стека.
6. Создать программу, отыскивающую проход по лабиринту. Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице. Программа находит проход через лабиринт, двигаясь от заданного входа. После отыскания прохода программа выводит найденный путь в виде координат квадратов.
7. Проверить правильность расстановки четырех видов скобок в математическом выражении. Выделить в выражении (указать порядковые номера символов в строке) одну (лишнюю) скобку, либо две скобки (если они не соответствуют друг другу).
8. Используя стек, написать программу для преобразования математического выражения, содержащего только 5 арифметических операций (сложение, вычитание, умножение, деление и возведение в

степень) и скобки, из стандартной (инфиксной) формы в постфиксную. (В постфиксной форме отсутствуют скобки, а знаки операторов следуют строго за операндами). Постфиксная запись выражения также иногда называется обратной польской нотацией. Например, для выражения

$$(2 + 2) * 2$$

обратная польская запись будет иметь вид:

$$2\ 2 + 2 *.$$

Другое выражение:

$$\frac{(a + b)^2}{(c - d)(e + f^{12})^3}$$

в постфиксной форме будет представлять собой:

$$a\ b + 2\hat{}c\ d - e\ f\ 12\hat{} + 3\hat{}\ */.$$

9. В условиях предыдущего задания вычислить значение выражения, предварительно приведя его к обратной польской форме записи:

$$\frac{13(48 + 11) + 56}{2}.$$

10. В условиях задания №4 вычислить значение выражения, предварительно переведя его в обратную польскую нотацию, для заданных пользователем значений переменных a, b, c:

$$\frac{3a(a + 2b)^c}{0.32(\boldsymbol{e}^{-a}c^{b-a})^{1.5}} - abc.$$

ОЧЕРЕДЬ

РАЗБОР АРХИТЕКТУРЫ ОЧЕРЕДИ И ПРИМЕР РЕАЛИЗАЦИИ

Очередь конструктивно, также как и стек, *представляет собой линейный список, работа с элементами которого осуществляется по принципу FIFO (first-in-first-out): элементы, заносимые в очередь первыми, первыми из нее и изымаются.* Обычно элемент добавляется в конец (хвост) очереди, а забираются с начала (головы) очереди.

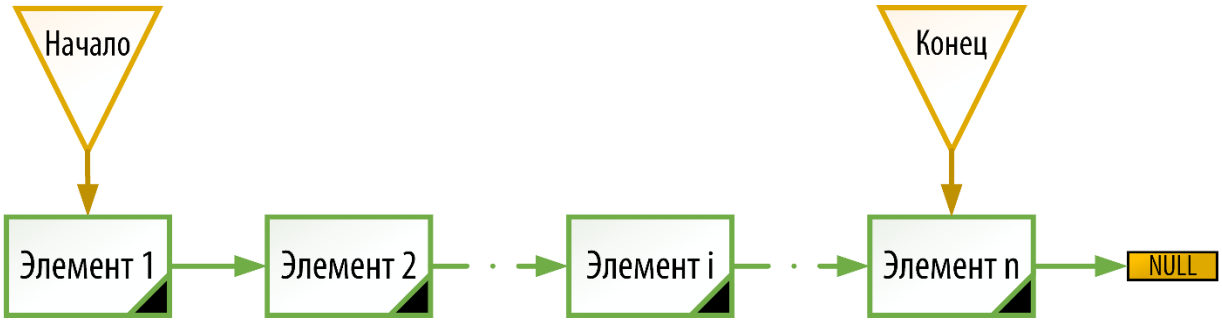


Рис. 2. Очередь

Существует также разновидность очереди, у которой элементы могут и включаться, и изыматься как с одного, так и с другого конца. Такая структура называется *деком* (DEQ – Double Ended Queue – двусторонняя очередь), однако ее рассмотрение выходит за рамки настоящего учебного пособия.

Для работы с очередью есть смысл реализовать функции добавления элемента в очередь и удаления его оттуда:

- **`enqueue`** (помещение нового элемента в очередь);
- **`dequeue`** (извлечение элемента из очереди с возвратом значения).

Также можно реализовать функцию удаления очереди, если не все элементы были из нее извлечены:

- **`clear`**.

При размещении нового элемента в очереди нужно проверять, не пуста ли она. При добавлении первого значения потребуется обновить указатели как на начало, так и на конец ДСД. При последующих вставках достаточно изменять только указатель на конец очереди.

Аналогично и при изъятии элемента проверяется, не был ли он последним. Если из очереди удаляется единственное значение, то, отдельно потребуется обнулить указатель на последний элемент (указатель на начало обнулится автоматически при выполнении стандартных действий алгоритма).

Ниже представлена программа, содержащая базовые функции для работы с очередью и способы их использования.

В качестве примера реализуем следующую программу. Пусть дан текстовый файл со списком лиц с известными ФИО и возрастом. Список хранится в алфавитном порядке. Требуется разделить людей из списка на две условные группы: трудоспособного (от 16 до 64 лет) и нетрудоспособного возраста и сохранить при этом исходный порядок в каждой из групп.

В демонстрационной программе, реализующей решение данной задачи, объявлены три структуры:

- **elem** – используется как основа для хранения элементов очереди;
- **man** – содержит в объединенном виде информационные поля, необходимые для реализации конкретного примера (см. ниже);
- **queue** – применяется для совместного хранения указателей на начало и конец очереди, а также счетчика ее элементов.

Для того, чтобы не отвлекаться от работы с очередью и облегчить восприятия примера, здесь и далее, будут использоваться только структуры и функции, предназначенные для хранения значений конкретных типов. При желании их можно самостоятельно переделать в шаблоны, по аналогии с примером для стека.

Файл «queue.h» содержит описание структур данных и основных функций.

```
1  #pragma once
2
3  /// <summary>
4  /// Структура для хранения данных о некотором человеке
5  /// (для примера будем хранить имя и возраст)
6  /// </summary>
7  struct man
8  {
9          /// <summary>
```

```
10          /// Имя человека
11          /// </summary>
12          char name[50];
13
14          /// <summary>
15          /// Возраст человека
16          /// </summary>
17          int age;
18  };
19
20  /// <summary>
21  /// Структура для хранения элементов очереди
22  /// </summary>
23  struct elem
24  {
25          /// <summary>
26          /// Информационное поле
27          /// </summary>
28          man m;
29
30          /// <summary>
31          /// Адресное поле
32          /// </summary>
33          elem* next = nullptr;
34  };
35
36  /// <summary>
37  /// Структура для работы с очередью, хранящая указатели на ее начало и конец
38  /// </summary>
39  struct queue
40  {
41          /// <summary>
42          /// Первый элемент очереди (голова)
43          /// </summary>
44          elem* head = nullptr;
45
46          /// <summary>
47          /// Последний элемент очереди (хвост)
48          /// </summary>
49          elem* tail = nullptr;
50
51          /// <summary>
52          /// Количество элементов в очереди
53          /// </summary>
54          size_t size = 0;
55  };
56
57  /// <summary>
58  /// Функция добавления элемента в очередь
59  /// </summary>
60  /// <param name="queue">Структура, определяющая очередь, куда добавляется элемент</param>
61  /// <param name="name">Имя человека, добавляемое в очередь</param>
62  /// <param name="age">Возраст человека, добавляемый в тот же элемент очереди</param>
63  void enqueue(queue& queue, const char* name, int age);
64
65  /// <summary>
66  /// Извлечение элемента из очереди
67  /// </summary>
```

```
68  /// <param name="queue">Структура, определяющая очередь откуда извлекается значение</param>
69  /// <param name="man">Структура, содержащая сведения о человеке</param>
70  /// <returns>
71  /// true, в случае, если значение было успешно извлечено из очереди и
72  /// false, в противном случае.
73  /// </returns>
74  bool dequeue(queue& queue, man& man);
75
76  /// <summary>
77  /// Очистка очереди
78  /// </summary>
79  /// <param name="queue">Структура, определяющая очередь, которая будет удалена</param>
80  void clear(queue& queue);
```

Файл «queue.cpp» с реализацией заявленных выше функций.

```
1   #include "queue.h"
2   #include <cstring>
3
4   /// <summary>
5   /// Вспомогательная функция для формирования элемента очереди.
6   /// Поскольку, ее вызов сторонними функциями не предусматривается,
7   /// ее описание отсутствует в заголовочном файле.
8   /// </summary>
9   /// <param name="name">Имя человека, сохраняемого в структуре</param>
10  /// <param name="age">Возраст человека, сохраняемый в структуре</param>
11  /// <returns>Структура, содержащие сведения о человеке</returns>
12  inline man create_elem(const char* name, int age)
13  {
14      man m;
15
16      // Если бы в структуре была строка в виде динамического массива, здесь нужно было бы
17      // предварительно выделить память под эту строку!
18
19      strcpy_s(m.name, name);
20      m.age = age;
21      return m;
22  }
23
24  void enqueue(queue& queue, const char* name, int age)
25  {
26      // Создаем новый элемент для размещения в очереди
27      man newm = create_elem(name, age);
28      elem* newel = new elem;
29      newel->m = newm;
30      if (!queue.tail)
31      {
32          // Если очереди еще не было, новый элемент становится единственным в ней
33          queue.head = queue.tail = newel;
34      } else
35      {
36          // Если очередь уже была, новый элемент помещается в конец:
37          queue.tail->next = newel;
38          queue.tail = newel;
39      }
40      queue.size++;
41  }
```

```
42
43  bool dequeue(queue& queue, man& m)
44  {
45        if (!queue.head) // Очереди нет
46              return false; // Вернуть значение невозможно
47        // Сохраняем возвращаемое значение
48        m = queue.head->m;
49        // Сохраняем указатель на начало очереди
50        elem* rem = queue.head;
51        // Изменяем адрес головного элемента
52        queue.head = queue.head->next;
53
54        // Если бы в структуре была строка в виде динамического массива, здесь нужно было бы
55        // предварительно очистить память из-под этой строки!
56
57        // Удаляем элемент с головы очереди
58        delete rem;
59        // Если элементов в очереди не осталось, обнуляем и указатель на ее конец.
60        if (!queue.head) queue.tail = nullptr;
61        queue.size--;
62        return true;
63  }
64
65  void clear(queue& queue)
66  {
67        // Проходим по всем элементам очереди, пока она не опустеет
68        while (queue.head)
69        {
70              elem* rem = queue.head; // Сохраняем ссылку на удаляемый элемент
71              queue.head = queue.head->next;// Переносим "голову" очереди вперед
72              delete rem; // Удаляем элемент по сохраненному указателю
73        }
74        queue.size = 0; // Обнуляем размер очереди
75        queue.tail = nullptr; // Обновляем указатель на последний элемент
76  }
```

Файл «main.cpp» с основной функцией **main()** и вспомогательными функциями для решения поставленной задачи.

```
1   #include <iostream>
2   #include <fstream>
3
4   #include "queue.h"
5   using namespace std;
6
7   /// Данная программа считывает данные о людях (имя и возраст) из файла и выводит на экран
8   /// сначала список лиц трудоспособного возраста, а затем - нетрудоспособного возраста,
9   /// сохраняя при этом исходный порядок расположения элементов хранимого в файле списка.
10  /// (Трудоспособным в программе считается возраст в диапазоне от 16 до 64 лет включительно).
11
12  /// <summary>
13  /// Функция для загрузки данных из файла и формирования очередей
14  /// </summary>
15  /// <param name="filename">Имя файла, содержащего сведения о людях (их имя и возраст)</param>
16  /// <param name="empl">Очередь из лиц трудоспособного возраста</param>
17  /// <param name="unempl">Очередь из лиц нетрудоспособного возраста</param>
```

```
18  void load_data(const char* filename, queue& empl, queue& unempl);
19
20  /// <summary>
21  /// Функция вывода данных из очереди
22  /// </summary>
23  /// <param name="q">Очередь, сведения из которой следуем отобразить на экране</param>
24  void show_data(queue& q);
25
26  void main()
27  {
28        setlocale(LC_ALL, "Rus");
29        // Создаем две очереди для работы с двумя категориями лиц:
30        queue employable;
31        queue unemployable;
32        // Загружаем данные в очереди
33        load_data("data.txt", employable, unemployable);
34        //Выводим список лиц трудоспособного возраста:
35        cout << "Лица трудоспособного возраста:\n";
36        show_data(employable);
37        //Выводим список лиц нетрудоспособного возраста:
38        cout << "Лица нетрудоспособного возраста:\n";
39        show_data(unemployable);
40        system("pause");
41  }
42
43  void load_data(const char* filename, queue& empl, queue& unempl)
44  {
45        ifstream f(filename);
46        if (f.is_open())
47        {
48              //До конца файла
49              while (!f.eof())
50              {
51                    char* man = new char[61];
52                    //Считываем очередную строку
53                    f.getline(man, 61);
54                    char *name = new char[50];
55                    int age;
56                    //Получаем имя и возраст человека в отдельных переменных
57                    sscanf_s(man, "%49[^0-9] %d", name, 50, &age);
58                    if (age<16 || age>=65) //Добавляем в список нетрудоспособных
59                          enqueue(unempl, name, age);
60                    else //Добавляем с писок трудоспособных
61                          enqueue(empl, name, age);
62              }
63              f.close();
64        }
65  }
66
67  void show_data(queue& q)
68  {
69        int i = 0;
70        while (q.size>0)
71        {
72              man m;
73              if (dequeue(q, m))
74              {
75              //Элемент получен, выводим данные
```

```
76              cout <<++i<<".\t"<<m.name<<"\tВозраст:"<<m.age<<"\n";
77              }
78      }
79  }
```

Определенный интерес в этой программе представляют параметры формата функции sscanf_s (строка 57). Само по себе ее применение обусловлено необходимостью отделения фамилии вместе с инициалами от возраста, а также преобразование данных о возрасте из строкового в числовой формат. Соответственно, конструкция **%49[^0-9]**, содержащая элементы так называемых регулярных выражений (используются для поиска в тексте по определенному шаблону), означает, что в данном месте могут располагаться любые символы *кроме* цифр от 0 до 9 в количестве до 49 штук (именно столько можно разместить в переменной **name**). Формат **%d**, в свою очередь, представляет собой указание на получение из строки целого числа (то есть в данном случае – возраста).

Для тестирования программы использовался файл с исходными данными «data.txt» следующего содержания:

```
Васильев П.Л. 45
Власов А.Ш. 39
Воронцова Ю.Е. 61
Еремеева А.П. 70
Иванов И.А. 14
Коршунова И.П. 15
Мечников И.Г. 33
Петрова А.Б. 22
Трусова В.В. 40
Хайруллин Р.Р. 29
```

В результате работы программы на экран будет выведено:

```
Лица трудоспособного возраста:
1.      Васильев П.Л.   Возраст: 45
2.      Власов А.Ш.     Возраст: 39
3.      Воронцова Ю.Е.  Возраст: 61
4.      Мечников И.Г.   Возраст: 33
5.      Петрова А.Б.    Возраст: 22
6.      Трусова В.В.    Возраст: 40
7.      Хайруллин Р.Р.  Возраст: 29
Лица нетрудоспособного возраста:
1.      Еремеева А.П.   Возраст: 70
2.      Иванов И.А.     Возраст: 14
3.      Коршунова И.П.  Возраст: 15
```

Можно заметить, что после разделения полного списка на две группы, исходный (алфавитный) порядок внутри каждой группы сохранен за счет создания в программе очередей без использования каких-либо методов сортировки.

Контрольные вопросы и задания для самостоятельного выполнения

1. Назовите принципы функционирования очереди.
2. С какой целью в программах выполняется проверка на пустоту очереди?
3. С какой целью в программах выполняется удаление очереди по окончанию работы с ней? Как изменится работа программы, если операцию удаления не выполнять?

---

4. Создать две очереди из двух целочисленных файлов. Затем объединить эти две очереди в одну, в которой элементы двух исходных очередей чередуются.
5. Создайте очередь из клиентов, список которых записан в файле. Распределите клиентов на две очереди (через одного).
6. Создайте очередь из клиентов, список которых записан в файле. Распределите клиентов на две очереди по цели обращения. (Информацию о цели обращения также предварительно добавить в исходный файл.)
7. Пусть имеется файл действительных чисел и некоторое число С. Используя очередь, напечатать сначала все элементы, меньшие числа С, а затем все остальные элементы, сохраняя в остальном исходный порядок их следования.
8. Реализовать очередь на базе двусвязного списка для решения задачи, разобранной в настоящем пособии выше.

9. Реализовать функции для работы с деком.
10. Реализовать очередь с приоритетом, то есть такую очередь, в которой элементы всегда забираются с головы, а добавляются в произвольное место, в зависимости от значения некоторого параметра *P* (приоритета), таким образом, чтобы его значения в очереди образовывали упорядоченную последовательность. Другой вариант реализации – зеркальный. Элементы всегда добавляются в конец, но извлекается из очереди элемент с наибольшим приоритетом.

СПИСОК

Базовая архитектура списков

Список *— это линейная динамическая структура данных, состоящая из узлов, каждый из которых содержит данные и адреса одного (в случае односвязного списка) или двух (в случае двусвязного списка) соседних узлов.*

Список является линейной конструкцией наиболее общего вида. Односвязный список (верхняя часть рис. 3) определяется одним своим концом, а двусвязный (нижняя часть рис. 3), соответственно, двумя концами.

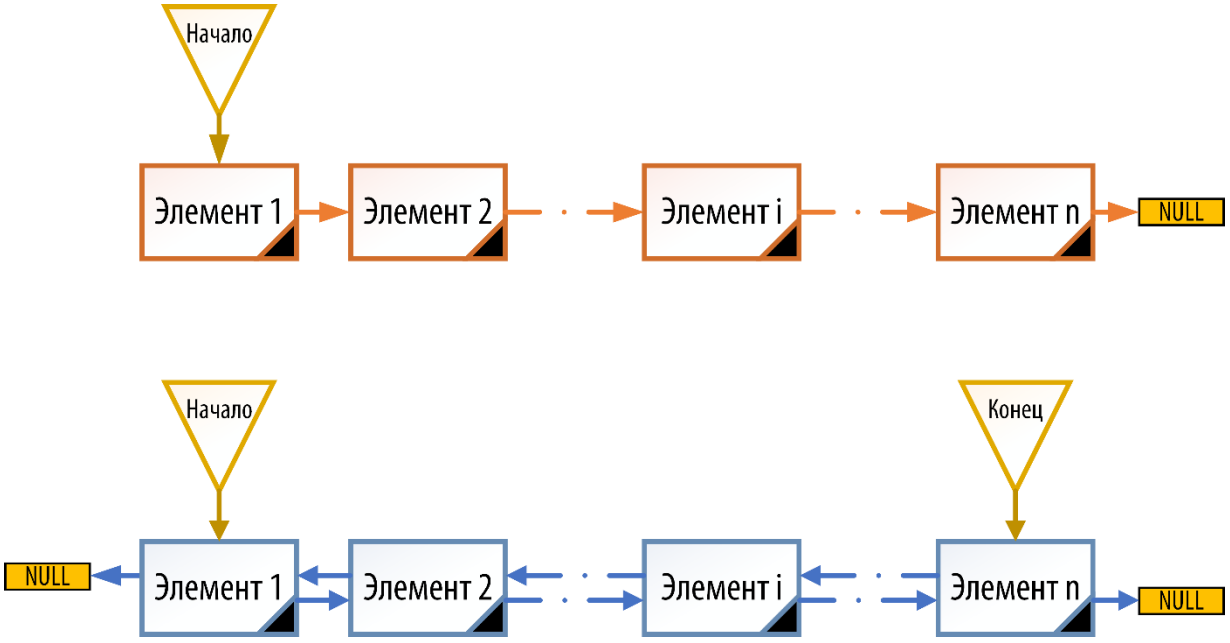


Рис. 3. Односвязный (сверху) и двусвязный список (снизу)

Элементы списка могут добавляться в любую из его позиций (в начало, конец или середину, без каких-либо ограничений).

На базе списков построены другие линейные виды динамических структур данных, в частности рассмотренные выше стек и очередь.

Работа с односвязным списком

В отличие от рассмотренных ранее динамических структур данных, при работе со списком нужно предусмотреть не только функцию добавления

элемента в его конец, но и вставки значения в произвольное место, указав его номер. При этом, в отличие от массива, в списке не так просто перейти к произвольной позиции – это возможно сделать только последовательно перебирая элементы, переходя по хранящимся в них ссылкам.

Аналогичная ситуация и с удалением значений. Эту операцию можно произвести с произвольным значением, указав либо его индекс, либо значение (в зависимости от варианта реализации функции удаления). При этом, алгоритм удаления будет работать по-разному, в зависимости от того, где расположен элемент: в начале списка или в середине. В первом случае необходимо обновить указатель на начало списка. Во втором – найти значение, предшествующее удаляемому, чтобы иметь возможность обновить связи.

Рассмотрим пример программы, демонстрирующей работу с односвязным списком. В ней будут реализованы функции для:

- добавления элемента в конец списка;
- вставки элемента в список на указанную позицию;
- получения значения элемента списка с заданным номером;
- вычисления длины списка;
- удаления элемента списка с заданным номером;
- очистки списка.

Кроме того, в программе реализованы функции, демонстрирующие работу со списком:

- формирование списка из данных, содержащихся в файле;
- вывод списка на экран;
- вывод некоторых значений из списка.

Файл «list1.h».

```
1   #pragma once
2
3   /// <summary>
4   /// Структура для реализации списка
5   /// </summary>
6   struct elem
7   {
```

```
8           /// <summary>
9           /// Поле для хранения данных
10          /// </summary>
11          int a;
12
13          /// <summary>
14          /// Указатель на следующий элемент.
15          /// </summary>
16          elem* next = nullptr;
17  };
18
19  /// <summary>
20  /// Функция для добавления элемента в конец списка
21  /// </summary>
22  /// <param name="list">Список, куда следует вставить новый элемент</param>
23  /// <param name="data">Добавляемое в список значение</param>
24  void add(elem*& list, int data);
25
26  /// <summary>
27  /// Функция для добавления элемента на указанное место в списке
28  /// (если в списке нет такого номера, элемент добавляется в его
29  /// начало или конец, в зависимости от того, что ближе).
30  /// </summary>
31  /// <param name="list">Список, в который нужно вставить значение</param>
32  /// <param name="data">Вставляемое в список значение</param>
33  /// <param name="position">Позиция, куда требуется вставить значение</param>
34  void insert(elem*& list, int data, int position);
35
36  /// <summary>
37  /// Функция поиска элемента по его номеру (позиции) в списке
38  /// </summary>
39  /// <param name="list">Список, в котором нужно производить поиск</param>
40  /// <param name="position">Номер получаемого элемента</param>
41  /// <returns>
42  /// Указатель на элемент списка, если в заданной позиции он существовал и
43  /// nullptr, если был задан некорректный номер позиции.
44  /// </returns>
45  const int* get(const elem* list, int position);
46
47  /// <summary>
48  /// Определение количества элементов списка
49  /// </summary>
50  /// <param name="list">Список, размер которого нужно вычислить</param>
51  /// <returns>Количество элементов списка</returns>
52  int count(const elem* list);
53
54  /// <summary>
55  /// Функция удаления элемента по его позиции
56  /// </summary>
57  /// <param name="list">Список, из которого нужно удалить элемент</param>
58  /// <param name="position">Позиция, из которой нужно удалить значение</param>
59  /// <returns>
60  /// true, если элемент успешно удален и
61  /// false, если элемент нельзя было удалить (была указана неверная позиция).
62  /// </returns>
63  bool remove(elem*& list, int position);
64
```

```
65  /// <summary>
66  /// Функция удаления списка
67  /// </summary>
68  /// <param name="list">Список, элементы которого нужно удалить из памяти</param>
69  void clear(elem*& list);
```

Файл «list1.cpp».

```
1   #include "list1.h"
2
3   void add(elem*& first, int data)
4   {
5           // Создание элемента списка
6           elem* newel = new elem;
7           newel->a = data;
8
9           // Определение наличия списка
10          if (first)
11          {
12                  // Список есть, добавляем элемент в конец.
13                  // Для этого пробегаем до последнего элемента списка.
14
15                  // Сначала устанавливаем указатель на первый элемент
16                  elem* curr = first;
17                  // Пока существует следующий элемент в списке
18                  while (curr->next)
19                  {
20                          // Перемещаем указатель на следующий элемент
21                          curr = curr->next;
22                  }
23                  //Указатель curr теперь находится на последнем элементе
24                  // устанавливаем у текущего элемента указатель next на новый элемент
25                  curr->next = newel;
26          } else
27          {
28                  // Списка раньше не было. Значит новый элемент будет первым
29                  first = newel;
30          }
31  }
32
33  void insert(elem*& first, int data, int pos)
34  {
35          // Добавление элемента в заданную позицию в списке.
36          // Если будет указано отрицательное или нулевое значение,
37          // элемент станет первым.
38          // Если будет указано значение, большее или равное длине списка,
39          // элемент станет последним.
40
41          // Создание элемента списка
42          elem* newel = new elem;
43          newel->a = data;
44
45          // Как и в функции add, проверяем наличие списка.
46          // Кроме того, проверяем, что позиция вставки >0, иначе
47          // новый элемент также оказывается первым.
48          if (first && pos>0)
49          {
```

```
50              elem* curr = first;// Указатель на текущий элемент
51              int p = 0;// Счетчик элементов
52              while (curr->next && p<pos-1)
53              {
54                      // Цикл работает либо пока не дойдем до последнего элемента,
55                      // либо пока не дойдем до позиции, предшествующей позиции вставки
56                      curr = curr->next;
57                      p++;
58              }
59              // curr теперь указывает на элемент, после которого нужно размещать новый
60
61              // Размещаем новый элемент в списке
62              newel->next = curr->next;
63              curr->next = newel;
64      } else
65      {
66              newel->next = first;
67              first = newel;
68      }
69  }
70
71
72  const int* get(const elem* first, int pos)
73  {
74      // Элемента с таким номером не существует
75      if (pos < 0 || !first) return nullptr;
76      // Находим элемент с указанным номером
77      int p = 0;
78      elem* curr = (elem*)first;
79      while (curr && p < pos)
80      {
81              curr = curr->next;
82              p++;
83      }
84      // В этот момент curr либо указывает на нужный элемент списка, либо равен nullptr,
85      // если элемента с нужным номером не существовало
86      if (curr)
87      {
88              // Элемент найден. Возвращаем указатель на значение
89              return &curr->a;
90      }
91      // Элемент не найден
92      return nullptr;
93  }
94
95  int count(const elem* first)
96  {
97      // Счетчик элементов списка
98      int cnt = 0;
99      elem* curr = (elem*)first;
100     while (curr)// Перебираем все элементы
101     {
102             cnt++;
103             curr = curr->next;
104     }
105     return cnt;
106 }
107
```

```
108 bool remove(elem*& first, int pos)
109 {
110       // Элемента с таким номером не существует – удалять нечего
111       if (pos < 0 || !first) return false;
112       // Если требуется удалить первый элемент, меняем указатель на начало списка
113       if (pos==0)
114       {
115             elem* rem = first;
116             first = first->next;
117             delete rem;
118             return true;
119       }
120       // Удаляется элемент не из начала.
121       // Находим элемент, предшествующий элементу с указанным номером
122       int p = 0;
123       elem* curr = first;
124       while (curr->next && p < pos-1)
125       {
126             curr = curr->next;
127             p++;
128       }
129       // Определяем причину выхода из цикла:
130       if (curr->next)
131       {
132             // Искомая позиция найдена, можно удалять следующий относительно curr элемент.
133             elem* rem = curr->next;
134             curr->next = curr->next->next;
135             delete rem;
136             return true;
137       }
138       // Элемент не найден.
139       return false;
140 }
141
142 void clear(elem*& first)
143 {
144       while (first)
145       {
146             // Запоминаем элемент для удаления
147             elem* del = first;
148             // Начало списка смещаем на следующий элемент
149             first = first->next;
150             // Удаляем сохраненный элемент.
151             delete del;
152       }
153 }
```

Файл «main.cpp».

Обратите внимание. Функция заполнения списка, расположенная здесь, рассчитана на работу с текстовым файлом с целочисленными значениями. В каждой его строке может находиться либо только одно число – значение элемента списка, либо два – значение и позиция, в которую его нужно поместить.

```cpp
#include <iostream>
#include <fstream>
using namespace std;

#include "list1.h"

/// <summary>
/// Пример функции для произвольного заполнения списка значениями
/// </summary>
/// <param name="list">Список, который требуется заполнить</param>
/// <returns>
/// true, если данные из файла успешно прочитаны и
/// false - при возникновении ошибок чтения из файла
/// </returns>
bool fill_list(elem*& list, const char* filename){
	// Загрузка элементов списка из файла
	// Если в строке файла одно число - оно добавляется в конец списка
	// Если в строке файла два числа, то второе означает позицию вставки
	ifstream f(filename);
	if (f.is_open())
	{
		// Файл найден и открыт
		// Создаем буфер для чтения данных
		char *data = new char[31];
		while (!f.eof())
		{
			// Читаем строку из файла
			f.getline(data, 31, '\n');
			if (f.fail())
			{
				// Ошибка чтения.
				f.close(); // Закрываем файл
				clear(list); // Очищаем часть уже созданного списка
				return false;
			}
			// Переменные для чтения значения и позиции
			int v, p;
			// Получаем числовые значения из прочитанной строки.
			// Функция возвращает количество успешно прочитанных значений.
			int cnt = sscanf_s(data, "%d %d", &v, &p);
			switch (cnt)
			{
			case 1:
				{
					// Есть только значение, добавляем его в список
					add(list, v);
					break;
				}
			case 2:
				{
					// Есть и значение, и его позиция.
					// Вставляем элемент на указанное место
					insert(list, v, p);
				}
			}
		}
		f.close();
		delete[] data;
```

```
59              return true;
60        }
61        return false;
62  }
63
64  /// <summary>
65  /// Отображение списка на экране
66  /// </summary>
67  /// <param name="list">Список, содержимое которого нужно показать</param>
68  void show_list(elem* list)
69  {
70        elem* curr = list;
71        if (!curr) cout << "EMPTY LIST" << endl;
72        else while (curr)
73        {
74              cout << curr->a << (curr->next ? " " : "\n");
75              curr = curr->next;
76        }
77  }
78
79  /// <summary>
80  /// Отображение элементов списка с нечетными номерами
81  /// </summary>
82  /// <param name="list">Список, элементы которого будут выводиться</param>
83  void show_odd(const elem* list)
84  {
85        // Определяем число элементов списка
86        int cnt = count(list);
87        for (int i = 1; i<cnt; i+=2)
88        {
89              // Получаем и выводим элемент
90              auto val = get(list, i);
91              cout << *val << ((i+2 < cnt) ? " " : "\n");
92        }
93  }
94
95  void main()
96  {
97        // Указатель на начало списка (изначально пуст)
98        elem* list1 = nullptr;
99        if (fill_list(list1, "data.txt"))
100       {
101             show_list(list1);// 300 -10 -1 30 1 10 100 200
102             remove(list1, 0);
103             show_list(list1);// -10 -1 30 1 10 100 200
104             remove(list1, 1);
105             show_list(list1);// -10 30 1 10 100 200
106             remove(list1, 100);
107             show_list(list1);// -10 30 1 10 100 200
108             remove(list1, -10);
109             show_list(list1);// -10 30 1 10 100 200
110             show_odd(list1); // 30, 10, 200
111             clear(list1);
112             show_list(list1);// EMPTY LIST
113       } else
114       {
115             cout << "Не удалось прочитать данные из файла :(";
116       }
```

```
117         system("pause");
118 }
```

Для испытания программы использовался файл с исходными данными «data.txt» следующего содержания:

```
1
10
100
-1 0
-10 0
30 2
200 30
300 -1
```

Результаты, полученные на экране после запуска программы, приведены в комментариях, находящихся в функции **main()**.

РАБОТА С ДВУСВЯЗНЫМ СПИСКОМ

При работе с односвязным списком можно было заметить, что при каждом добавлении элемента в конец списка, приходилось «пробегать» в цикле от начала до конца через все сохраненные значения. Соответственно, чем длиннее будет становиться список, тем медленнее будет выполняться эта функция. Использование двусвязного списка позволяет несколько сократить число операций перебора элементов при необходимости добавления нового числа в конец всей конструкции. Это становится возможным, поскольку двусвязный список содержит дополнительный указатель на последний его элемент. Также двусвязный список допускает перемещение по списку в двух направлениях. Однако при этом следует обратить внимание, что при выполнении операции вставки или удаления элемента, нужно обновлять большее количество указателей.

В следующем примере будет рассмотрена работа с двусвязным отсортированным списком. Отметим, что процесс упорядочивания элементов будет производиться сразу на этапе формирования списка посредством вставки элемента сразу в нужную позицию.

Список базируется на структуре **elem2**, для удобства работы со списком также введена структура **list**, в которой содержится совокупность указателей на первый и последний элемент списка, а также поле, хранящее длину списка.

В программе будут содержаться функции:

- добавления элемента в список с сохранением упорядоченности;
- удаления элемента из списка с указанной позиции;
- получение элемента по его номеру;
- удаления всего списка.

Файл «list2.h».

```
1   #pragma once
2
3   /// <summary>
4   /// Структура для хранения элементов списка
5   /// </summary>
6   struct elem
7   {
8           /// <summary>
9           /// Хранимое значение
10          /// </summary>
11          double x;
12
13          /// <summary>
14          /// Указатель на следующий элемент списка
15          /// </summary>
16          elem* next = nullptr;
17
18          /// <summary>
19          /// Указатель на предыдущий элемент списка
20          /// </summary>
21          elem* prev = nullptr;
22  };
23
24  /// <summary>
25  /// Структура для объединения указателей, определяющих один двусвязный список.
26  /// Предназначена для облегчения использования списка в сторонних функциях
27  /// </summary>
28  struct list2
29  {
30          /// <summary>
31          /// Указатель на первый элемент списка
32          /// </summary>
33          elem* first = nullptr;
34
35          /// <summary>
36          /// Указатель на последний элемент списка
37          /// </summary>
38          elem* last = nullptr;
```

```
39
40          /// <summary>
41          /// Поле для хранения размера списка
42          /// </summary>
43          int count = 0;
44  };
45
46  /// <summary>
47  /// Функция для добавления элемента в список
48  /// </summary>
49  /// <param name="list">Список, в который необходимо добавить элемент</param>
50  /// <param name="data">Размещаемое в списке значение</param>
51  void add(list2& list, double data);
52
53  /// <summary>
54  /// Функция для удаления элемента из списка по его индексу
55  /// </summary>
56  /// <param name="list">Список, из которого нужно удалить значение</param>
57  /// <param name="position">Позиция, с которой требуется удалить элемент</param>
58  /// <returns>
59  /// Признак успешности удаления элемента.
60  /// true - если удаление успешно выполнено
61  /// false - если указана неверная позиция и удалить элемент не удалось.
62  /// </returns>
63  bool remove (list2& list, int position);
64
65  /// <summary>
66  /// Функция получения элемента списка по индексу
67  /// </summary>
68  /// <param name="list">Список, из которого требуется получить значение</param>
69  /// <param name="position">Позиция элемента, значение которого нужно узнать</param>
70  /// <returns>
71  /// Указатель на значение в списке, если указана корректная позиция
72  /// или nullptr, если элемент получить не удалось.
73  /// </returns>
74  const double* get(list2 list, int position);
75
76  /// <summary>
77  /// Функция очистки списка
78  /// </summary>
79  /// <param name="list">Список, элементы которого нужно удалить</param>
80  void clear(list2& list);
```

Файл «list2.cpp».

```
1   #include "list2.h"
2
3   void add(list2& list, double data)
4   {
5           // Создание нового элемента списка
6           elem* newel = new elem;
7           newel->x = data;
8           // При вставке элемента увеличиваем длину списка
9           list.count++;
10          // Проверка наличия списка
11          if (!list.first)
```

```
12          {
13                // Если списка еще не было, новый элемент будет первым
14                list.first = list.last = newel;
15          } else
16          {
17                // Список уже существовал.
18                // Необходимо определить, куда именно вставить новый элемент
19                // для сохранения упорядоченности
20                if (newel->x <= list.first->x)
21                {
22                      // Элемент меньше всех ранее существовавших в списке. Делаем его первым.
23                      newel->next = list.first;
24                      list.first->prev = newel;
25                      list.first = newel;
26                } else if (newel->x >= list.last->x)
27                {
28                      // Элемент по значению больше всех, ранее существовавших в списке.
29                      // Делаем его последним
30                      newel->prev = list.last;
31                      list.last->next = newel;
32                      list.last = newel;
33                } else
34                {
35                      // Элемент необходимо расположить где-то в середине списка.
36                      // Определяем местоположение элемента.
37                      elem* curr = list.first->next;
38                      // Переходим по элементам списка, пока значения элементов меньше нового
39                      while (curr->x < newel->x)
40                      {
41                            curr = curr->next;
42                      }
43                      // Здесь curr указывает на элемент,
44                      // перед которым нужно расположить новый
45                      // Осуществляем вставку элемента с расстановкой указателей
46                      newel->next = curr;
47                      newel->prev = curr->prev;
48                      curr->prev->next = newel;
49                      curr->prev = newel;
50                }
51          }
52    }
53
54    bool remove(list2& list, int pos)
55    {
56          if (pos < 0 || pos>=list.count)
57                return false; // Такого элемента нет.
58          list.count--; // Количество элементов будет уменьшено.
59          if (pos == 0)
60          {
61                // Удаляется первый элемент
62                elem* rem = list.first;
63                // Изменяем указатель на начало списка
64                list.first = list.first->next;
65                if (list.first) list.first->prev = nullptr;
66                delete rem;
67                return true;
68          }
69          if (pos == list.count)
```

```
70          {
71                // Удаляется последний элемент
72                elem* rem = list.last;
73                // Изменяем указатель на конец списка
74                list.last = list.last->prev;
75                list.last->next = nullptr;
76                delete rem;
77                return true;
78          }
79          // Удаляется элемент из середины списка
80          //Определяем направление обхода списка
81          bool frw = (pos <= list.count / 2);
82          int p = 1;
83          elem* curr;
84          if (frw)
85          {
86                // Если удаляемый элемент из первой половины, пробегаем от начала...
87                curr = list.first->next;
88          } else
89          {
90                // ...иначе - с конца.
91                pos = list.count - pos;
92                curr = list.last->prev;
93          }
94          // Пробегаем по списку до искомого элемента, либо до конца списка
95          while (p < pos)
96          {
97                // Шагаем вперед или назад в зависимости от направления движения по списку
98                curr = frw ? curr->next : curr->prev;
99                p++;
100         }
101         // Корректируем указатель у предыдущего элемента
102         curr->prev->next = curr->next;
103         // Корректируем указатель у следующего элемента
104         curr->next->prev = rem->prev;
105         // Удаляем текущий элемент
106         delete curr;
107         return true;
108   }
109
110   const double* get(list2 list, int pos)
111   {
112         if (!list.first || pos < 0 || pos>=list.count)
113               return nullptr; // Такого элемента нет
114         if (!pos)return &list.first->x; // Возвращаем значение первого элемента
115         if (pos==list.count-1)
116               return &list.last->x; // Возвращаем значение последнего элемента
117         // Иначе ищем элемент с нужным номером
118         int p = 1;
119         // Определяем предпочтительное направление поиска
120         bool fwd = pos<=list.count/2;
121         elem* curr;
122         // Находим стартовый элемент
123         if (fwd)
124               curr = list.first->next;
125         else {
126               curr = list.last->prev;
127               // Корректируем номер искомого элемента в зависимости от направления поиска
```

```
128             pos = list.count - pos - 1;
129       }
130       while (curr && p<pos)
131       {
132             p++;
133             // Переходим вперед или назад в зависимости от направления поиска
134             curr = fwd ? curr->next : curr->prev;
135       }
136       // Здесь указатель находится на искомом элементе
137       return &curr->x;
138 }
139
140 void clear(list2& list)
141 {
142       // Функция очистки двусвязного списка практически не отличается от таковой
143       // для списка односвязного.
144       elem* rem;
145       while (list.first)
146       {
147             rem = list.first;
148             list.first = list.first->next;
149             delete rem;
150       }
151       list.last = nullptr; // Корректируем указатель на последний элемент
152       list.count = 0; // Корректируем количество элементов списка
153 }
```

Файл «main.cpp».

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  #include "list2.h"
6
7  bool fill_list(list2& list, const char* filename)
8  {
9        // Будем получать данные для списка из файла
10       ifstream f(filename);
11       if (f.is_open())
12       {
13             // Файл найден и открылся
14             while (!f.eof())
15             {
16                   double x;
17                   f >> x;
18                   if (f.fail()) { // Ошибка чтения из файла
19                         clear(list); // Очистка сформированной части списка
20                         return false;
21                   }
22                   add(list, x);
23             }
24             f.close();
25             return true;
26       }
27       return false;
28 }
```

```
29
30  /// <summary>
31  /// Отображение элементов списка на экране
32  /// </summary>
33  /// <param name="list">Список, значения которого требуется отобразить</param>
34  /// <param name="reversed">
35  ///       Определяет порядок вывода элементов:
36  ///       false - прямой (от первого до последнего) или
37  ///       true - обратный (от последнего к первому).
38  /// </param>
39  void show_list(const list2& list, bool reversed = false)
40  {
41        // Устанавливаем указатель на начало или конец списка в зависимости от порядка
42        // вывода его элементов, определяемого параметром reversed
43        elem* curr = reversed ? list.last : list.first;
44        if (!curr) cout << "Список пуст\n";// Список пуст
45        // Пробегаем по всем элементам списка и выводим значения
46        else while (curr)
47        {
48              elem* cmp = reversed ? list.first : list.last;
49              cout << curr->x << ((curr==cmp)?"\n":" ");
50              curr = reversed ? curr->prev : curr->next;
51        }
52  }
53
54  void main()
55  {
56        setlocale(LC_ALL, "RUS");
57        list2 list;
58        if (!fill_list(list, "data.txt"))
59        {
60              cout << "Не удалось прочитать данные из файла";
61              return;
62        }
63        show_list(list);
64        // Будет выведено: -88.12 -5.44 0 1 1 2.5 3.11 3.15 3.5 3.5 3.52 6.843 7.11 91.11
65
66        show_list(list, true);
67        // Будет выведено: 91.11 7.11 6.843 3.52 3.5 3.5 3.15 3.11 2.5 1 1 0 -5.44 -88.12
68
69        cout << *get(list, 0) << endl; // -88.12
70        cout << *get(list, 5) << endl; // 2.5
71
72        //Более корректный способ получения элемента с проверкой на непустоту значения
73        auto val = get(list, 10);
74        if (val) cout << *val << endl; // 3.52
75
76        remove(list, 1);
77        remove(list, 10);
78        show_list(list);
79        // Будет выведено: -88.12 0 1 1 2.5 3.11 3.15 3.5 3.5 3.52 7.11 91.11
80
81        cout <<  get(list, 10) << endl; // ######## (адрес значения элемента)
82        val = get(list, 10);
83        if (val) cout << *val << endl; // 7.11 (значение элемента)
84        val = get(list, 11);
85        if (val) cout << *val << endl; // 91.11 (значение элемента)
86        val = get(list, -3);
```

```
87          if (val) cout << *val << endl;
88          else cout << "Элемент отсутствует\n";
89          //Будет выведено: Элемент отсутствует
90
91          val = get(list, 100);
92          if (val) cout << *val << endl;
93          else cout << "Элемент отсутствует\n";
94          //Будет выведено: Элемент отсутствует
95
96          clear(list);
97          show_list(list); // Список пуст
98          system("pause");
99      }
```

Файл с исходными данными «data.txt»:

```
3.5 6.843 7.11 3.52 3.11 -5.44 -88.12 91.11 3.15 0 1 1 2.5 3.5
```

Результаты, получаемые для указанного файла приведены в комментариях, в функции **`main()`**.

В заключении отметим, что в качестве альтернативы двум указателям на начало и конец списка можно создать циклический список с барьерным элементом. В таком списке указатель на следующее значение у последнего элемента содержит адрес барьерного элемента, и наоборот – предыдущим элементом для барьерного является последний. Следовательно для работы с таким списком достаточно лишь одного указателя (см. рис. 4).

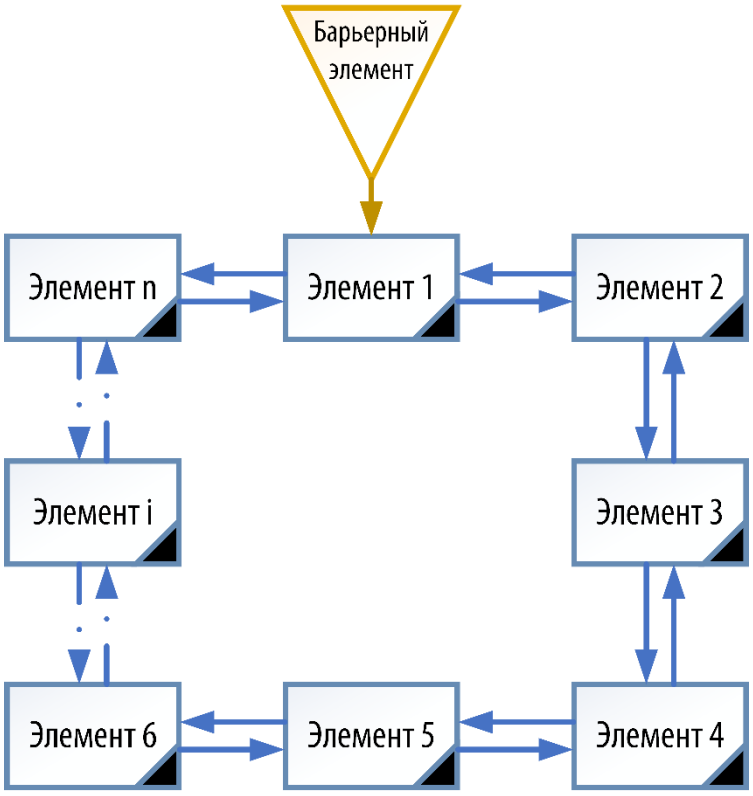


Рис. 4. Циклический список с барьерным элементом

Контрольные вопросы и задания для самостоятельного выполнения

1. Любой ли список является связным? Обоснуйте ответ.
2. В чем отличие первого элемента однонаправленного (двунаправленного) списка от остальных элементов этого же списка?
3. В чем отличие последнего элемента однонаправленного (двунаправленного) списка от остальных элементов этого же списка?
4. В чем принципиальные отличия выполнения добавления (удаления) элемента на первую и любую другую позиции в однонаправленном списке?
5. В чем принципиальные отличия выполнения основных операций в однонаправленных и двунаправленных списках?

---

6. Из одного односвязного списка с фамилиями студентов двух групп создать два односвязных списка, разделив студентов на группы и отсортировав их фамилии в алфавитном порядке.
7. Написать функцию для смены местами двух элементов односвязного списка по их номерам. Нумерацию элементов начинать с 0.
8. Реализовать метод добавления элемента в двусвязный список, так, чтобы в списке оставались только уникальные значения.
9. Загрузить из двух числовых файлов данные в два двусвязных циклических списка. Объединить списки в один, так, чтобы в нем были только различные элементы.
10. Используя двусвязный список, написать программу сложения длинных чисел. Числа разбить на группы по 9 цифр, которые будут являться элементами списка. Результат сложения занести в циклический список.

ДЕРЕВО

Базовая архитектура

Дерево *– это иерархическая (нелинейная) динамическая структура данных, которая состоит из набора вершин и ребер.*

Вершиной *или* ***узлом*** *называется каждый элемент дерева.* Каждый узел помимо полезной информации (информационных полей) содержит указатели на вершины следующего, более нижнего уровня (адресные поля). *Один из узлов дерева обычно выделяется в качестве основного, называемого* ***корнем****. Вершины, которые не имеют указателей на узлы нижних уровней называются* ***листьями****.* Общий вид дерева представлен на рис. 5.

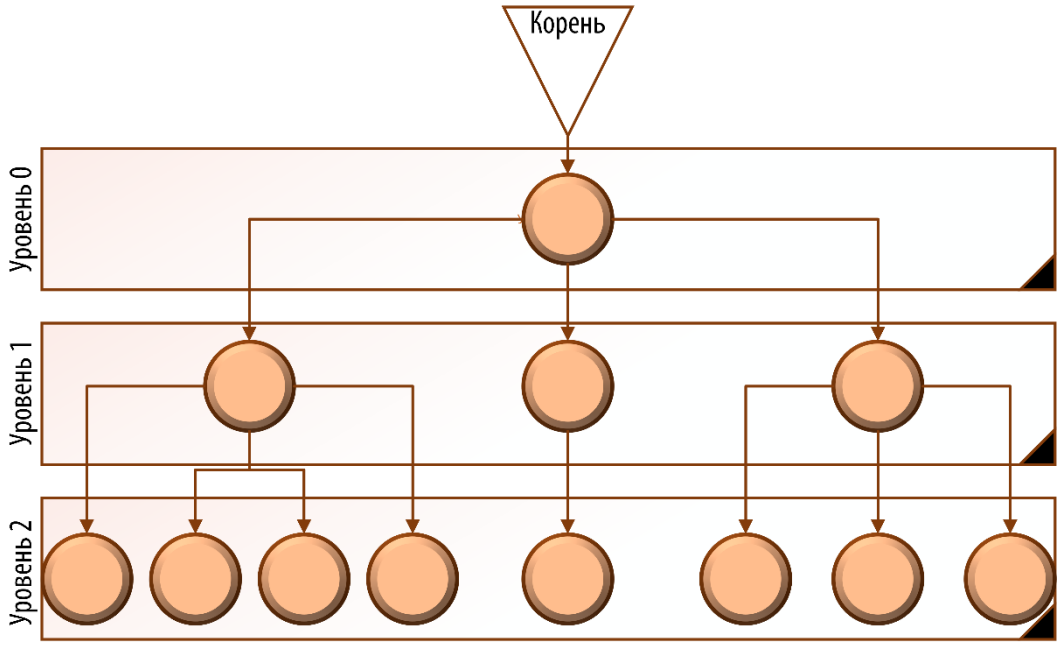


Рис. 5. Дерево общего вида

Все вершины дерева разбиты по уровням. На нулевом уровне расположен корень. Находящийся на уровне i узел x_i, имеет указатели на *дочерние* вершины (***потомки***) на уровне $i+1$: y_{i+1}^k, для которых он сам будет, соответственно, являться ***родительским узлом*** (***предком***).

Число равное количеству уровней в дереве называется его ***высотой****.*

Деревья, узлы которых отсортированы по какому-то признаку, называются ***упорядоченными****.*

Деревья, у которых каждый узел имеет ссылки не более чем на два других узла, называются ***двоичными*** *или* ***бинарными****.*

Бинарные деревья поиска

В настоящем пособии основное внимание будет уделено упорядоченным бинарным деревьям, которые также иногда называются ***бинарными деревьями поиска*** (см. рис. 6). Помимо хранения данных такие деревья позволяют уменьшать время поиска своих элементов по сравнению с линейными структурами. Для ускорения поиска элементов в таких деревьях, значения их узлов располагаются по определенному правилу: слева размещаются элементы меньшие, чем элемент верхнего уровня, а справа – большие. Отметим, что добавление одинаковых элементов в дерево, может приводить к негативным последствиям, внося сбои в работу ряда алгоритмов, поэтому такие элементы лучше не включать в дерево. При этом для сохранения общности конструкции, каждому узлу можно добавить дополнительное поле для хранения количества соответствующих элементов, встретившихся в исходном наборе данных.

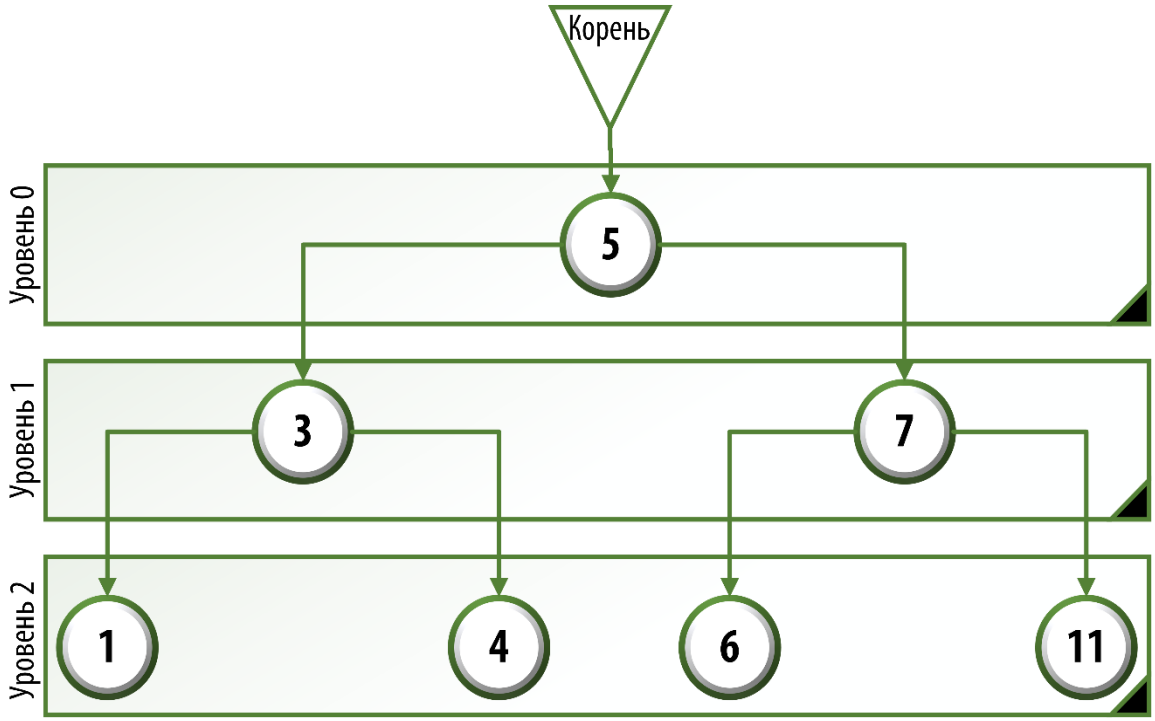


Рис. 6. Бинарное дерево поиска

Дерево, даже бинарное, – достаточно сложная динамическая структура данных. Поэтому в настоящем пособии будет рассмотрено несколько примеров для работы с ним (с нарастающей сложностью).

В первом примере содержится базовый набор функций для работы с бинарным деревом поиска:

- добавление элемента (**`add_to_tree`**);
- поиск элемента (**`find_in_tree`**);
- удаление элемента из дерева (**`remove_from_tree`**);
- удаление всего дерева (**`drop_tree`**).

Для простоты рассмотрим вариант дерева, в котором в качестве значений (**`value`**) узлов будут храниться целые числа.

Файл «tree.h» с объявлением структуры для хранения элементов (узлов) дерева, а также описанием базовых функций.

```
1   #pragma once
2
3   /// <summary>
4   /// Структура для хранения узлов дерева
5   /// </summary>
6   struct node
7   {
8           /// <summary>
9           /// Хранимое значение (информационное поле)
10          /// </summary>
11          int value;
12
13          /// <summary>
14          /// Количество соответствующих значений в исходном наборе (информационное поле)
15          /// </summary>
16          int count = 1;
17
18          /// <summary>
19          /// Указатель на левое поддерево (адресное поле)
20          /// </summary>
21          node* left = nullptr;
22
23          /// <summary>
24          /// Указатель на правое поддерево (адресное поле)
25          /// </summary>
26          node* right = nullptr;
27  };
28
29  /// <summary>
30  /// Структура для хранения информации о дереве в целом
31  /// </summary>
32  struct tree
```

```
33  {
34          /// <summary>
35          /// Корень, определяющий дерево
36          /// </summary>
37          node* root = nullptr;
38
39          /// <summary>
40          /// Общее число элементов в дереве
41          /// </summary>
42          size_t elem_count = 0;
43  };
44
45  /// <summary>
46  /// Функция добавления элемента в дерево
47  /// </summary>
48  /// <param name="tree">Дерево, где нужно разместить новый элемент</param>
49  /// <param name="value">Добавляемое в дерево значение</param>
50  void add_to_tree(tree& tree, int value);
51
52  /// <summary>
53  /// Функция удаления элемента из дерева
54  /// </summary>
55  /// <param name="tree">Дерево, откуда нужно удалить значение</param>
56  /// <param name="value">Удаляемое значение</param>
57  /// <returns>
58  /// true, если значение было найдено и успешно удалено
59  /// false, если указанного значения не нашлось.
60  /// </returns>
61  bool remove_from_tree(tree& tree, int value);
62
63  /// <summary>
64  /// Функция поиска элемента в дереве
65  /// </summary>
66  /// <param name="tree">Дерево, в котором осуществляется поиск</param>
67  /// <param name="value">Искомое значение</param>
68  /// <returns>
69  /// Если элемент найден, функция возвращает количество соответствующих значений
70  /// в исходном наборе данных.
71  /// Возвращает 0, если искомый элемент не обнаружен
72  /// </returns>
73  int find_in_tree(tree tree, int value);
74
75  /// <summary>
76  /// Функция удаления дерева
77  /// </summary>
78  /// <param name="tree">Дерево, все узлы которого нужно удалить</param>
79  void drop_tree(tree& tree);
```

Файл «tree.cpp» с реализацией соответствующих функций. Отметим, что большинство приведенных функций для работы с деревом являются рекурсивными, что существенно сокращает объем кода и облегчает чтение программы.

Обратите внимание! Некоторые функции реализованы в программе в двух вариантах: первый предназначен для вызова извне, например, из функции **main()**; другой, напротив, нужен для удобства внутреннего использования и, соответственно, эти варианты не описываются в заголовке «tree.h». Доступ к ним будет только в файле «tree.cpp».

```
1   #include "tree.h"
2
3   //-------------------------------------------------------------
4   //Объявления вспомогательных функций для внутреннего использования
5
6   /// <summary>
7   /// Добавление элемента в дерево. Вспомогательная (внутренняя) функция.
8   /// </summary>
9   /// <param name="root">Корень дерева, в которое нужно добавить новое значение</param>
10  /// <param name="value">Добавляемое в дерево значение</param>
11  /// <returns>
12  /// true, если в дереве был размещен новый узел и
13  /// false, если элемент с таким значение в дереве уже присутствовал и
14  /// было только увеличено поле количества значений исходной последовательности
15  /// </returns>
16  bool _add_to_tree(node*& root, int value);
17
18  /// <summary>
19  /// Удаление элемента из дерева. Вспомогательная (внутренняя) функция.
20  /// </summary>
21  /// <param name="root">Корень дерева, откуда требуется удалить узел</param>
22  /// <param name="value">Удаляемое значение</param>
23  /// <returns>
24  /// true, если элемент в дереве существовал и был удален функцией
25  /// false, если узла с указанным значением в дереве не было
26  /// </returns>
27  bool _remove_from_tree(node*& root, int value);
28
29  /// <summary>
30  /// Поиск элемента в дереве. Вспомогательная (внутренняя) функция.
31  /// </summary>
32  /// <param name="root">Корень дерева, в котором ищется элемент</param>
33  /// <param name="value">Искомое значение</param>
34  /// <returns>
35  /// Если элемент найден, функция возвращает количество соответствующих значений
36  /// в исходном наборе данных.
37  /// Возвращает 0, если искомый элемент не обнаружен
38  /// </returns>
39  int _find_in_tree(const node* root, int value);
40
41  /// <summary>
42  /// Удаление дерева. Вспомогательная (внутренняя) функция.
43  /// </summary>
44  /// <param name="root">Корень дерева, которое необходимо удалить</param>
45  void _drop_tree(node*& root);
46
47  /// <summary>
48  /// Перемещение значений из самого правого узла левого поддерева,
49  /// либо самого левого узла правого поддерева в узел, значение которого
50  /// нужно удалить.
```

```
51  /// Вспомогательная функция. Используется при избирательном удалении элементов дерева.
52  /// </summary>
53  /// <param name="rem">Указатель на удаляемый узел</param>
54  void _move_node(node* rem);
55
56  /// <summary>
57  /// Находит предка для элемента, являющегося ближайшим к удаляемому из дерева.
58  /// Вспомогательная функция.
59  /// </summary>
60  /// <param name="removing">Удаляемый элемент</param>
61  /// <returns>
62  /// Указатель на предка для элемента, являющегося ближайшим к удаляемому из дерева.
63  /// </returns>
64  node* _find_prev_nearest(node* removing);
65
66  /// <summary>
67  /// Определяет, с левым ли поддеревом необходимо работать при удалении элемента
68  /// </summary>
69  /// <param name="n"> Узел, для которого выполняется проверка</param>
70  /// <returns> true, если левое поддерево существует и можно с ним работать; иначе - false</returns>
71  inline bool _is_left(const node* n);
72
73  /// <summary>
74  /// Копирование значений основных информационных полей из одного узла в другой
75  /// </summary>
76  /// <param name="to">Узел, куда копируются значения</param>
77  /// <param name="from">Узел, из которого копируются значения</param>
78  inline void _copy_value(node* to, const node* from);
79
80  /// <summary>
81  /// Функция для удаления узла дерева
82  /// </summary>
83  /// <param name="n">Указатель на удаляемый узел</param>
84  inline void _del(node*& n);
85  //-------------------------------------------------------------
86
87  void add_to_tree(tree& t, int value)
88  {
89        if (_add_to_tree(t.root, value))
90              // Если добавился новый элемент, увеличиваем счетчик количества узлов дерева
91              t.elem_count++;
92  }
93
94  bool remove_from_tree(tree& t, int value)
95  {
96        auto res = _remove_from_tree(t.root, value);
97        if (res)
98              // Если элемент был удален из дерева, уменьшаем счетчик узлов в дереве
99              t.elem_count--;
100       return res;
101 }
102
103 int find_in_tree(tree t, int value)
104 {
105       return _find_in_tree(t.root, value);
106 }
107
108 void drop_tree(tree& t)
```

```
109 {
110       _drop_tree(t.root);
111       // В дереве больше нет ни одного узла
112       t.elem_count = 0;
113 }
114
115 bool _add_to_tree(node*& root, int value)
116 {
117       // Проверяем, существует ли дерево
118       if (root)
119       {
120             // Дерево уже существует. Определяем, куда разместить новый элемент
121             if (value == root->value)
122             {
123       // Если значение в новом элементе совпадает со значением в текущем элементе дерева
124       // Вместо добавления элемента, увеличиваем количество таких элементов
125                   root->count++;
126                   return false;
127             }
128             // В зависимости от значения нового элемента относительно текущего...
129             auto& branch =
130                   value < root->value ? root->left : root->right;
131             // ...рекурсивно размещаем элемент в левом или правом поддереве
132             bool r = _add_to_tree(branch, value);
133
134             return r;
135       }
136       // Если дерева нет, то новый элемент становится корневым
137       // Создаем новый узел для размещения в дереве
138       node* newnode = new node;
139       newnode->value = value;
140       // и помещаем в качестве текущего (корневого) узла
141       root = newnode;
142       return true;
143 }
144
145 bool _remove_from_tree(node*& root, int value)
146 {
147       if (root) {
148             if (value == root->value)
149             {
150                   // Удаляемый элемент найден.
151                   if (!root->left && !root->right)
152                         // Если это лист, просто удаляем его.
153                         _del(root);
154                   else
155                         // Иначе выполняем перенос
156                         _move_node(root);
157                   return true;
158             }
159             // Выполняем рекурсивный поиск удаляемого элемента в левом или правом поддереве
160             auto& subtree =
161               value < root->value ? root->left : root->right;
162             bool r = _remove_from_tree(subtree, value);
163             return r;
164       }
165       return false;
166 }
```

```
167
168 int _find_in_tree(const node* root, int value)
169 {
170       // Если корень пуст, значит искомое значение не найдено, количество искомых значений = 0
171       if (!root) return 0;
172       // Если значение имеется в корневом элементе, возвращаем количество
173       if (root->value == value) return root->count;
174       // Иначе продолжаем рекурсивный поиск в поддеревьях
175       if (value < root->value)
176             return _find_in_tree(root->left, value);
177       return _find_in_tree(root->right, value);
178 }
179
180 void _drop_tree(node*& root)
181 {
182       if (root)  // Если есть, что удалять
183       {
184             // Рекурсивно удаляется левое поддерево
185             _drop_tree(root->left);
186             // Рекурсивно удаляется правое поддерево
187             _drop_tree(root->right);
188             // Удаляется корневой элемент
189             _del(root);
190       }
191 }
192
193 void _move_node(node* rem)
194 {
195       auto parent = _find_prev_nearest(rem);
196       // Определяем, будем ли работать с левым поддеревом
197       auto is_lft = _is_left(rem);
198       // Если ближайший узел находится сразу за удаляемым,
199       // инверсируем признак поддерева
200       if (parent == rem) is_lft = !is_lft;
201       // Находим в одном из поддеревьев элемент, ближайший к удаляемому
202       // Он является самым правым в левом поддереве
203       // и самым левым – в правом.
204       auto& to_remove = is_lft ? parent->right : parent->left;
205       // Если есть продолжение дерева после фактически удаляемого элемента
206       node* child = nullptr;
207       if (to_remove)
208             child = is_lft ? to_remove->right : to_remove->left;
209       // Копируем значения(!) из ближайшего (к удаляемому) узла в удаляемый узел
210       // При этом сам удаляемый узел останется в памяти (но с новым значением),
211       // Лишний элемент будет удален у дерева снизу.
212       _copy_value(rem, to_remove);
213       // Выполняем удаление лишнего узла
214       _del(to_remove);
215       // Прицепляем «остаток» дерева к родительскому узлу
216       to_remove = child;
217 }
218
219 node* _find_prev_nearest(node* rem)
220 {
221       // Если удаляемого узла нет, сразу выходим
222       if (!rem) {
223             return rem;
224       }
```

```
225         // Устанавливаем удаляемый элемент в качестве родительского
226         auto prev = rem;
227         // Определяем с каким поддеревом работать
228         if (_is_left(rem))
229         {
230               // Поиск в левом поддереве крайнего правого элемента и его предка.
231               // Предок необходим для дальнейшей настройки связей между оставшимися узлами
232               auto last = rem->left;
233               while (last && last->right) {
234                     prev = last;
235                     last = last->right;
236               }
237         }
238         else
239         {
240               // Поиск в правом поддереве крайнего левого элемента и его предка.
241               // Предок необходим для дальнейшей настройки связей между оставшимися узлами
242               auto last = rem->right;
243               while (last && last->left) {
244                     prev = last;
245                     last = last->left;
246               }
247         }
248         return prev;
249   }
250
251   inline bool _is_left(const node* n)
252   {
253         //Если левое поддерево есть, работаем с ним
254         return (bool)n->left;
255   }
256
257   inline void _copy_value(node* to, const node* from)
258   {
259         // Если оба узла корректные (не равны nullptr)
260         if (to && from)
261         {
262               // Выполняем копирование значений основных информационных полей
263               to->value = from->value;
264               to->count = from->count;
265         }
266   }
267
268   inline void _del(node*& n)
269   {
270         delete n;
271         n = nullptr;
272   }
```

Обратите внимание! Стоит отдельно отметить небольшую особенность, встречающуюся в коде, без которой программа не будет функционировать корректно. Посмотрите, например, на строку 204. В ней объявляется переменная автоматически выводимого типа *со ссылкой*. Наличие здесь ссылки обеспечивает сохранение изменений в структуре дерева, в родительском узле (**parent**). В противном случае (при отсутствии ссылки) структура дерева разрушится.

Файл «main.cpp»

```
1   #include <iostream>
2   #include "tree.h"
3   using namespace std;
4
5   // Вспомогательные функции для создания и отображения дерева
6
7   /// <summary>
8   /// Функция создает дерево, последовательно заполняя его значениями из указанного массива
9   /// </summary>
10  /// <param name="t">Дерево, которое требуется сформировать</param>
11  /// <param name="arr">Массив элементов, которые требуется разместить в дереве</param>
12  /// <param name="elem_count">Количество элементов массива</param>
13  void create_tree(tree& t, const int arr[], size_t elem_count)
14  {
15        for (auto i = 0; i < elem_count; i++)
16        {
17              add_to_tree(t, arr[i]);
18        }
19  }
20  //------------------------------------------------------------
21
22  void main()
23  {
24        setlocale(LC_ALL, "Rus");
25        // Вспомогательный массив, содержащий элементы, которые будут помещены в дерево
26        int arr[] = { 5, 2, 8, 1, 4, 0, 6, 9, 10,
27                      7, 1, 1, 7, 7, 0, 4, 5, 5, 10 };
28        tree tree;
29        create_tree(tree, arr, sizeof(arr) / sizeof(arr[0]));
30        // Сформированное дерево изображено на рис. 7 (см. ниже)
31        remove_from_tree(tree, 5);   // Дерево рис. 8 (см. ниже)
32        remove_from_tree(tree, 9);   // Дерево рис. 9 (см. ниже)
33        remove_from_tree(tree, 0);   // Дерево рис. 10 (см. ниже)
34        remove_from_tree(tree, 8);   // Дерево рис. 11 (см. ниже)
35        remove_from_tree(tree, -1); // Такого элемента не было
36        drop_tree(tree);
37        system("pause");
38  }
```

В результате работы программы по массиву элементов, заданных в строках 26-27, будет создано дерево (рис. 7 – представление, имеющееся до первого удаления, то есть до выполнения строки 31).

Далее, начиная с 31 строки из дерева последовательно удаляются элементы. В результате будут получаться варианты, представленные на рис. 8 – 11.

Обратите внимание, что при таком способе объединения ветвей после удаления узлов, сохраняется общий принцип расположения элементов в дереве

поиска: меньшие значения (относительно узла-предка) находятся слева, а большие – справа.

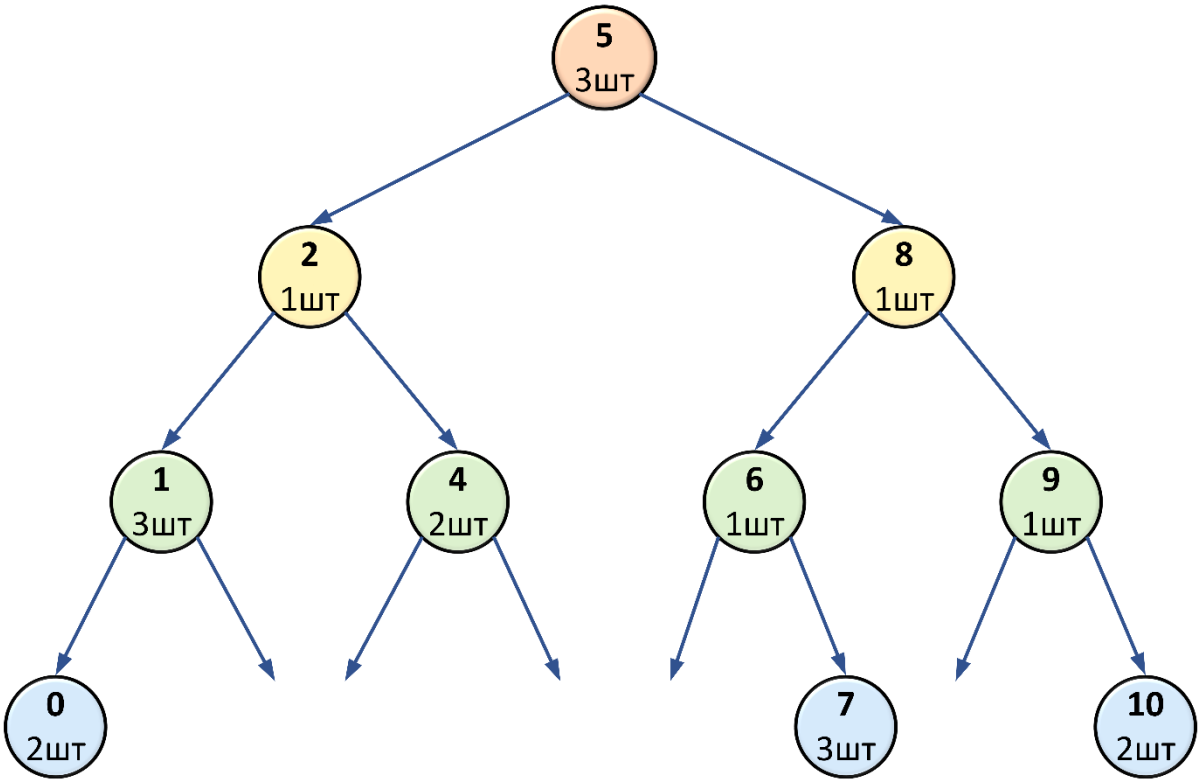


Рис. 7. Исходное дерево

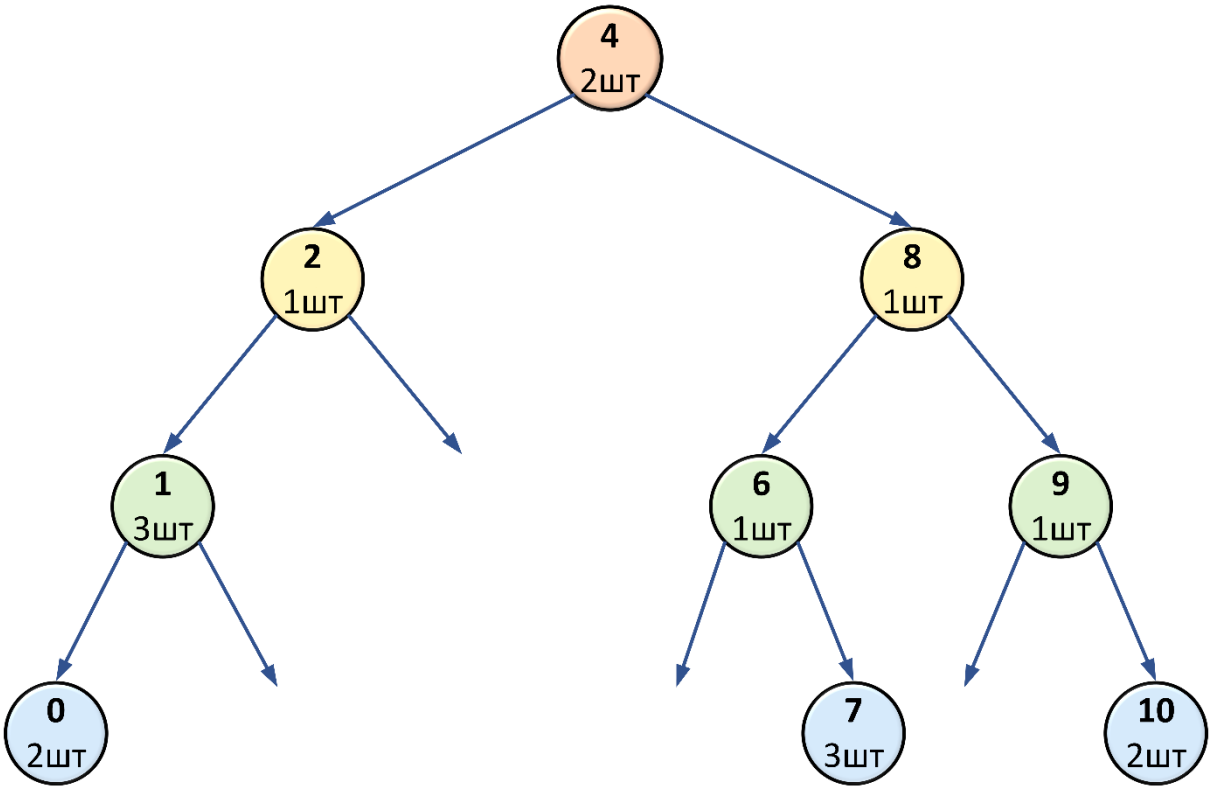


Рис. 8. Дерево после удаления элемента «5» (корня)

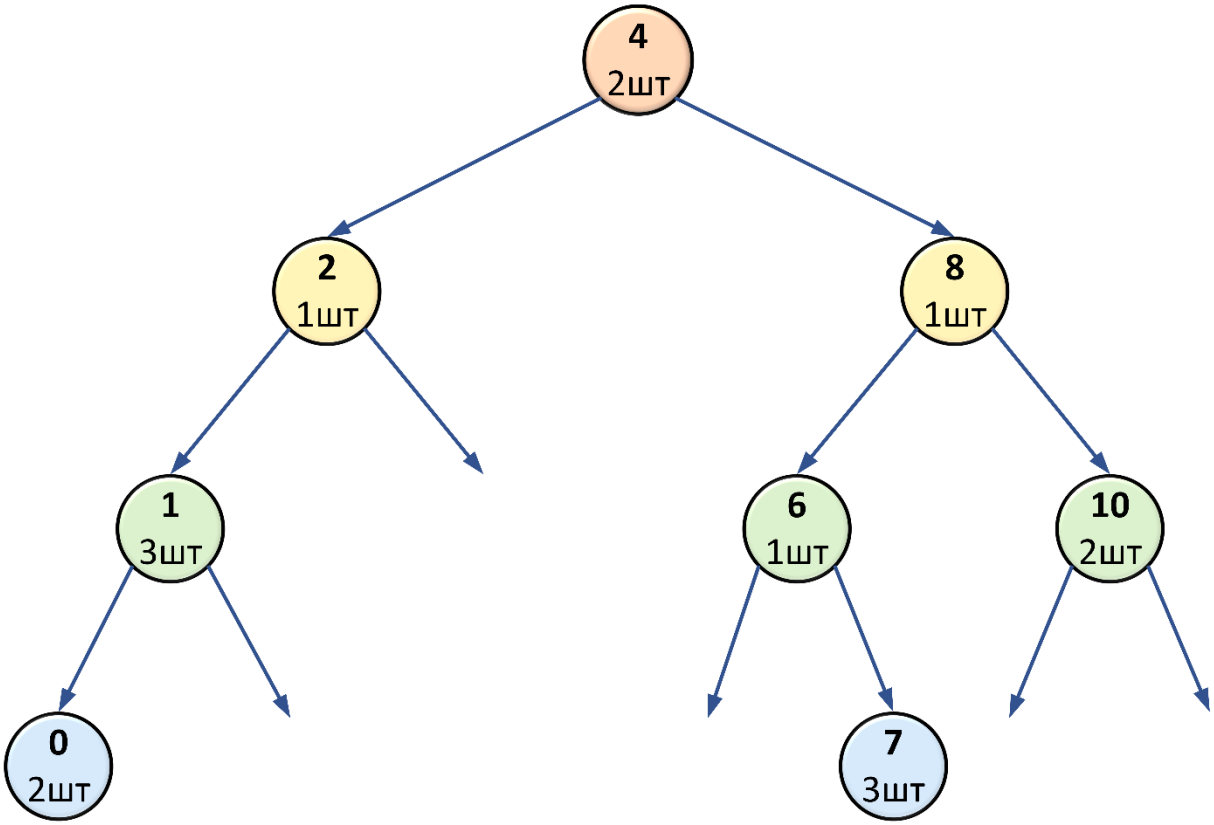


Рис. 9. Дерево после удаления элемента «9»,
у которого есть один единственный узел-потомок

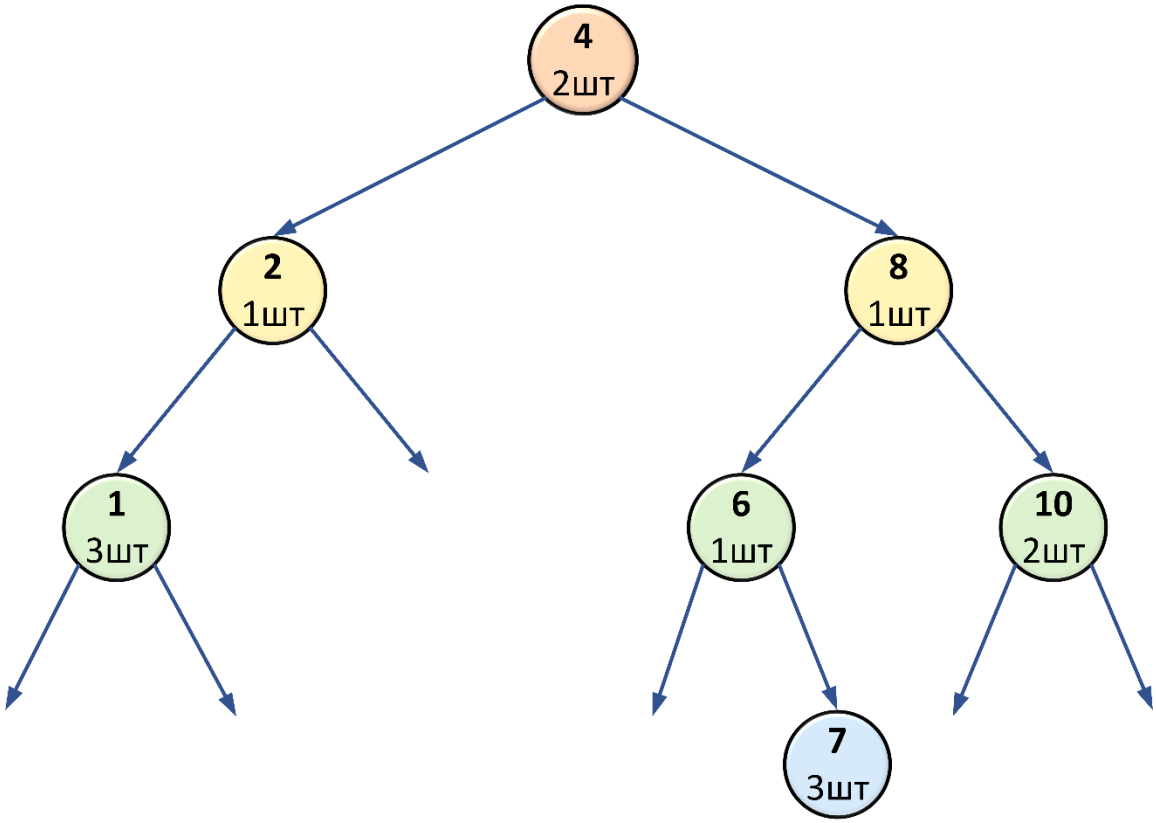


Рис. 10. Дерево после удаления элемента «0» – листа

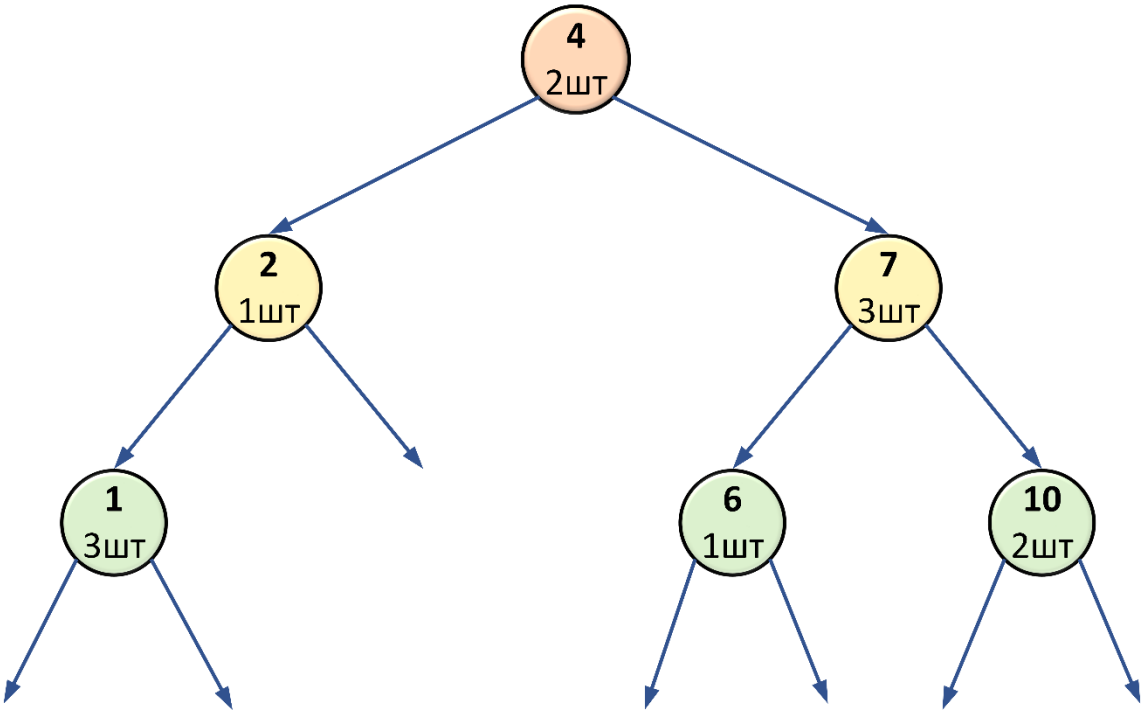


Рис. 11. Дерево после удаления элемента «8» – промежуточного узла, после которого расположено несколько других

Приведенный выше пример демонстрирует основы работы с динамической структурой данной «бинарное дерево». Однако для того, чтобы проверить расположенные в нем элементы, потребуется использовать отладчик и отслеживать в нем состояние различных переменных и указателей. В следующем примере в рассмотренное приложение будет добавлен ряд функций для получения элементов дерева и отображения их на экране.

Обход деревьев

Поскольку узлы в дереве расположены нелинейно, возникает вопрос, каким именно образом можно перебрать все его элементы.

Существует несколько способов перебора всех значений в дереве – так называемых вариантов обхода дерева. Во-первых, обход элементов можно осуществлять вглубь или вширь. При обходе вглубь порядок обхода узлов также может варьироваться. Здесь выделяют три способа: префиксный (прямой), инфиксный (центрированный) и постфиксный (обратный). Общий вид классификации приведен на рис. 12.

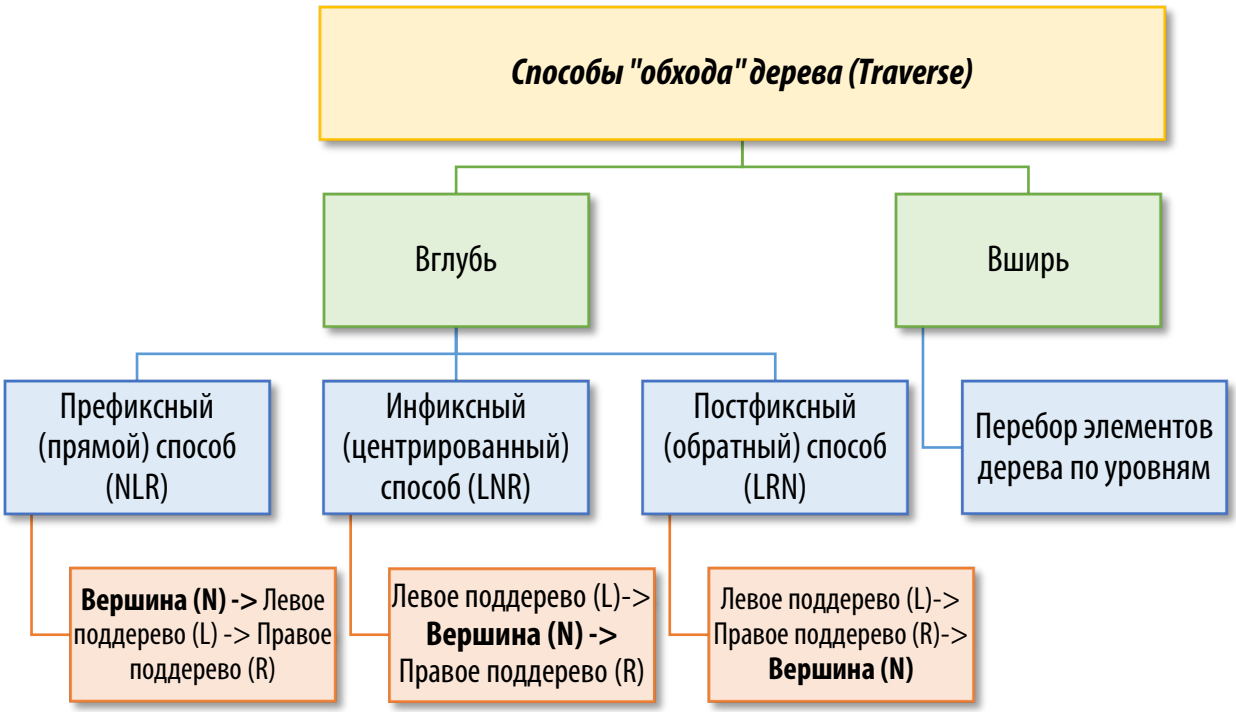


Рис. 12. Классификация способов обхода элементов в бинарном дереве

Ниже будут приведены вариант кода, содержащий функции для выполнения каждого из четырех видов обхода и размещения элементов в очереди для последующего использования при выводе.

Обход дерева вглубь

Для формирования определенной последовательности обхода элементов дерева, последние будут размещаться в очереди в нужном порядке. Это потребует использования функций и структур для работы с очередями. С этой целью добавим в проект файлы «queue.h» и «queue.cpp», аналогичные тем, что были описаны в одном из предыдущих разделов настоящего пособия, за исключением того, что вместо структуры **`man`**, использованной в примере ранее, везде в очереди будет использована указатель на структуру **`node`** дерева.

При этом может возникнуть неприятность, связанная с появлением циклической зависимости между файлами «queue.h» и «tree.h». Действительно, в первый из них необходимо вставить инструкцию **`#include "tree.h"`**, поскольку в очереди будет использована структура

дерева. В то же время, внутри «tree.h» необходимо включить инструкцию **`#include "queue.h"`**, поскольку функции обхода дерева будут формировать очереди, а значит использовать соответствующую структуру.

Для того, чтобы избежать подобных конфликтов между различными частями проекта, разместим все структуры (как для дерева, так и для очереди) в отдельном файле, который назовем «tree_struct.h».

```
1   #pragma once
2
3   /// <summary>
4   /// Структура для хранения узлов дерева
5   /// </summary>
6   struct node
7   {
8           /// <summary>
9           /// Хранимое значение (информационное поле)
10          /// </summary>
11          int value;
12
13          /// <summary>
14          /// Количество соответствующих значений в исходном наборе (информационное поле)
15          /// </summary>
16          int count = 1;
17
18          /// <summary>
19          /// Указатель на левое поддерево (адресное поле)
20          /// </summary>
21          node* left = nullptr;
22
23          /// <summary>
24          /// Указатель на правое поддерево (адресное поле)
25          /// </summary>
26          node* right = nullptr;
27  };
28
29  /// <summary>
30  /// Структура для хранения информации о дереве в целом
31  /// </summary>
32  struct tree
33  {
34          /// <summary>
35          /// Корень, определяющий дерево
36          /// </summary>
37          node* root = nullptr;
38
39          /// <summary>
40          /// Общее число элементов в дереве
41          /// </summary>
42          size_t elem_count = 0;
43  };
44
45  /// <summary>
46  /// Структура для хранения элементов очереди
```

```
47  /// </summary>
48  struct elem
49  {
50          /// <summary>
51          /// Информационное поле – указатель на узлы дерева
52          /// </summary>
53          node* node;
54
55          /// <summary>
56          /// Адресное поле – указатель на следующий элемент очереди
57          /// </summary>
58          elem* next = nullptr;
59  };
60
61  /// <summary>
62  /// Структура для работы с очередью, хранящая указатели на ее начало и конец
63  /// </summary>
64  struct queue
65  {
66          /// <summary>
67          /// Первый элемент очереди (голова)
68          /// </summary>
69          elem* head = nullptr;
70
71          /// <summary>
72          /// Последний элемент очереди (хвост)
73          /// </summary>
74          elem* tail = nullptr;
75
76          /// <summary>
77          /// Количество элементов в очереди
78          /// </summary>
79          size_t size = 0;
80  };
```

Тогда файл «queue.h» будет выглядеть следующим образом.

```
1   #pragma once
2   #include "tree_struct.h"
3
4   /// <summary>
5   /// Добавление элемента в очередь
6   /// </summary>
7   /// <param name="queue">Очередь, в которую вносится новый элемент</param>
8   /// <param name="tree_node">Узел дерева, добавляемый в очередь</param>
9   void enqueue(queue& queue, node* tree_node);
10
11  /// <summary>
12  /// Извлечение элемента из очереди
13  /// </summary>
14  /// <param name="queue">Очередь, из которой извлекается очередной элемент</param>
15  /// <param name="tree_node">Узел дерева, извлекаемый из очереди</param>
16  /// <returns>
17  /// true, если очередь не была пустой и элемент успешно извлечен
18  /// false - в противном случае.
19  /// </returns>
20  bool dequeue(queue& queue, node*& tree_node);
```

```
21
22  /// <summary>
23  /// Очистка очереди
24  /// </summary>
25  /// <param name="queue">Очередь, из которой необходимо удалить все элементы</param>
26  void clear(queue& queue);
```

Также приведем и содержание претерпевшего изменение файла «queue.cpp».

```
1
2   #include "queue.h"
3   #include <cstring>
4
5   void enqueue(queue& queue, node* tree_node)
6   {
7       // Создаем новый элемент для размещения в очереди
8       elem* newel = new elem;
9       newel->node = tree_node;
10      if (!queue.tail)
11      {
12          // Если очереди еще не было, новый элемент становится единственным в ней
13          queue.head = queue.tail = newel;
14      }
15      else
16      {
17          // Если очередь уже была, новый элемент помещается в конец:
18          queue.tail->next = newel;
19          queue.tail = newel;
20      }
21      queue.size++;
22  }
23
24  bool dequeue(queue& queue, node*& tree_node)
25  {
26      if (!queue.head)   // Очереди нет
27          return false;  // Вернуть значение невозможно
28      // Сохраняем возвращаемое значение
29      tree_node = queue.head->node;
30      // Сохраняем указатель на начало очереди
31      elem* rem = queue.head;
32      // Изменяем адрес головного элемента
33      queue.head = queue.head->next;
34      // Удаляем элемент с головы очереди
35      delete rem;
36      // Если элементов в очереди не осталось, обнуляем и указатель на ее конец.
37      if (!queue.head) queue.tail = nullptr;
38      queue.size--;
39      return true;
40  }
41
42  void clear(queue& queue)
43  {
44      // Проходим по всем элементам очереди, пока она не опустеет
45      while (queue.head)
46      {
```

```
47                  // Сохраняем ссылку на удаляемый элемент
48                  elem* rem = queue.head;
49                  // Переносим указатель вперед
50                  queue.head = queue.head->next;
51                  // Удаляем элемент по сохраненному указателю
52                  delete rem;
53          }
54          // Обнуляем размер очереди
55          queue.size = 0;
56          // Обновляем указатель на последний элемент
57          queue.tail = nullptr;
58  }
```

Файл «tree.h» претерпит следующие изменения по сравнению с первой версией.

Обратите внимание! Здесь и далее удаляемые строки будут зачеркнуты (нумерация у таких строк отсутствует), а строки, которые не изменяются будут отмечены серым цветом фона.

```
1   #pragma once
2   #include "tree_struct.h"

    /// <summary>
    /// Структура для хранения узлов дерева
    /// </summary>
    struct node
    {
            /// <summary>
            /// Хранимое значение (информационное поле)
            /// </summary>
            int value;

            /// <summary>
            /// Количество соответствующих значений в исходном наборе (информационное поле)
            /// </summary>
            int count = 1;

            /// <summary>
            /// Указатель на левое поддерево (адресное поле)
            /// </summary>
            node* left = nullptr;

            /// <summary>
            /// Указатель на правое поддерево (адресное поле)
            /// </summary>
            node* right = nullptr;
    };

    /// <summary>
    /// Структура для хранения информации о дереве в целом
    /// </summary>
    struct tree
    {
            /// <summary>
```

```
        /// Корень, определяющий дерево
        /// </summary>
        node* root = nullptr;

        /// <summary>
        /// Общее число элементов в дереве
        /// </summary>
        size_t elem_count = 0;
};

/// Функция добавления элемента в дерево
/// </summary>
/// <param name="tree">Дерево, где нужно разместить новый элемент</param>
/// <param name="value">Добавляемое в дерево значение</param>
void add_to_tree(tree& tree, int value);

/// <summary>
/// Функция удаления элемента из дерева
/// </summary>
/// <param name="tree">Дерево, откуда нужно удалить значение</param>
/// <param name="value">Удаляемое значение</param>
/// <returns>
/// true, если значение было найдено и успешно удалено
/// false, если указанного значения не нашлось.
/// </returns>
bool remove_from_tree(tree& tree, int value);

/// <summary>
/// Функция поиска элемента в дереве
/// </summary>
/// <param name="tree">Дерево, в котором осуществляется поиск</param>
/// <param name="value">Искомое значение</param>
/// <returns>
/// Если элемент найден, функция возвращает количество соответствующих значений
/// в исходном наборе данных.
/// Возвращает 0, если искомый элемент не обнаружен
/// </returns>
int find_in_tree(tree tree, int value);

/// <summary>
/// Функция удаления дерева
/// </summary>
/// <param name="tree">Дерево, все узлы которого нужно удалить</param>
void drop_tree(tree& tree);

/// <summary>
/// Префиксный (прямой) обход дерева вглубь (NLR)
/// </summary>
/// <param name="tree">Дерево, которое необходимо обойти</param>
/// <returns>Очередь элементов дерева в необходимом порядке</returns>
queue prefix_traverse(tree tree);

/// <summary>
/// Инфиксный (центрированный) обход дерева вглубь (LNR)
/// </summary>
/// <param name="tree">Дерево, которое необходимо обойти</param>
/// <returns>Очередь элементов дерева в необходимом порядке</returns>
queue infix_traverse(tree tree);
```

```
52
53  /// <summary>
54  /// Постфиксный (обратный) обход дерева вглубь (LRN)
55  /// </summary>
56  /// <param name="tree">Дерево, которое необходимо обойти</param>
57  /// <returns>Очередь элементов дерева в необходимом порядке</returns>
58  queue postfix_traverse(tree tree);
```

Изменения файла «tree.cpp».

Обратите внимание! В следующих листингах часть кода, которая не изменилась по сравнению с прошлым примером, может быть пропущена в целях экономии места. Следите за нумерацией строк!

```
1   #include "tree.h"
2   #include "queue.h"
3
4   //-------------------------------------------------------------
5   //Объявления вспомогательных функций для внутреннего использования
                // Часть кода, которая не изменилась, была пропущена для экономии места
                                                    //...
85  inline void _del(node*& n);
86
87  /// <summary>
88  /// Перечисление возможных видов обхода дерева вглубь. Вспомогательный элемент.
89  /// </summary>
90  enum _traverse_type { infix, prefix, postfix };
91
92  /// <summary>
93  /// Функция обхода дерева вглубь. Вспомогательная (внутренняя) функция
94  /// </summary>
95  /// <param name="q">Очередь из элементов дерева, формируемая функцией</param>
96  /// <param name="root">Корень дерева, для которого выполняется обход</param>
97  /// <param name="type">Тип обхода (инфиксный, префиксный или постфиксный)</param>
98  void _traverse(queue& q, node* root, _traverse_type type);
99  //-------------------------------------------------------------
                // Часть кода, которая не изменилась, была пропущена для экономии места
287
288 void _traverse(queue& q, node* root, _traverse_type type)
289 {
290       // Реализация обхода дерева вглубь
291       if (root)
292       {
293             switch (type)
294             {
295             case infix: // Обход вида LNR
296                   // Рекурсивный вывод значений левого поддерева (L)
297                   if (root->left) _traverse(q, root->left, type);
298                   // Вывод значений узла (N)
299                   enqueue(q, root);
300                   // Рекурсивный вывод значений правого поддерева (R)
301                   if (root->right)traverse(q, root->right, type);
302                   break;
303             case prefix: // Обход вида NLR
304                   // Вывод значения узла (N)
```

```
305                     enqueue(q, root);
306                     // Рекурсивный вывод значений левого поддерева (L)
307                     if (root->left)traverse(q, root->left, type);
308                     // Рекурсивный вывод значений правого поддерева (R)
309                     if (root->right)traverse(q, root->right, type);
310                     break;
311             case postfix: // Обход вида LRN
312                     // Рекурсивный вывод значений левого поддерева (L)
313                     if (root->left) _traverse(q, root->left, type);
314                     // Рекурсивный вывод значений правого поддерева (R)
315                     if (root->right)traverse(q, root->right, type);
316                     // Вывод значения узла (N)
317                     enqueue(q, root);
318             }
319     }
320 }
321
322 queue infix_traverse(tree t)
323 {
324     queue q;
325     traverse(q, t.root, infix);
326     return q;
327 }
328
329 queue prefix_traverse(tree t)
330 {
331     queue q;
332     traverse(q, t.root, prefix);
333     return q;
334 }
335
336 queue postfix_traverse(tree t)
337 {
338     queue q;
339     traverse(q, t.root, postfix);
340     return q;
341 }
```

Наличие функций обхода дерева позволит выводить его элементы на экран. Для этого можно добавить в «main.cpp» функцию отображения **`show_tree()`** и вызвать ее, указывая при этом тип обхода дерева, который необходимо произвести. Также разместим в файле «main.cpp» перечисление **`enum traverse_type`** со списком различных видов обходов дерева.

```
1  #include <iostream>
2  #include "queue.h"
3  #include "tree.h"
4  using namespace std;
5
6  // Вспомогательные функции для создания и отображения дерева
7
```

```
8   /// <summary>
9   /// Перечисление возможных видов обхода дерева.
10  /// </summary>
11  enum traverse_type { prefix, infix, postfix };
12
13  /// <summary>
14  /// Функция отображает на экране элементы из очереди, составленной из узлов дерева
15  /// </summary>
16  /// <param name="tree">Дерево, которое необходимо вывести на экран</param>
17  /// <param name="tt">
18  /// Указывает на тип выполнения обхода дерева.
19  /// </param>
20  void show_tree(tree tree, traverse_type tt)
21  {
22        // Формирование очереди для выполнения обхода определенного вида
23        queue tq;
24        switch (tt)
25        {
26        case prefix:    tq = prefix_traverse(tree);     break;
27        case infix:     tq = infix_traverse(tree);      break;
28        case postfix:   tq = postfix_traverse(tree);    break;
29        }
30        // Получение элемента с головы очереди для начала обхода
31        auto got = tq.head;
32        // Пока в очереди есть элементы (полученный элемент не равен nullptr)
33        while (got)
34        {
35              auto n = got->node;
36              // Если элемент не пуст, выводится его значение
37              // и в скобках количество таких значений в исходном наборе
38              if (n) cout << n->value << "(*"
39                    << n->count << ")";
40              // Если элемент пуст, вместо его значения выводится прочерк
41              else cout << "-";
42              // Переходим к следующему по очереди элементу
43              got = got->next;
44              // Выводим символ-разделитель: пробел, если в очереди еще есть элементы
45              // или символ перехода на новую строку, если элемент был последним.
46              cout << (got ? " " : "\n");
47        }
48        // Удаление вспомогательной очереди
49        clear(tq);
50  }
51
52  /// <summary>
53  /// Функция создает дерево, последовательно заполняя его значениями из указанного массива
54  /// </summary>
55  /// <param name="t">Дерево, которое требуется сформировать</param>
56  /// <param name="arr">Массив элементов, которые требуется разместить в дереве</param>
57  /// <param name="elem_count">Количество элементов массива</param>
58  void create_tree(tree& t, const int arr[], size_t elem_count)
59  {
60        for (auto i = 0; i < elem_count; i++)
61        {
62              add_to_tree(t, arr[i]);
63        }
64  }
65  //-----------------------------------------------------------
```

```
66
67  void main()
68  {
69        setlocale(LC_ALL, "Rus");
70        // Вспомогательный массив, содержащий элементы, которые будут помещены в дерево
71        int arr[] = { 5, 2, 8, 1, 4, 0, 6, 9, 10,
72                      7, 1, 1, 7, 7, 0, 4, 5, 5, 10 };
73        tree tree;
74        create_tree(tree, arr, sizeof(arr) / sizeof(arr[0]));
          // Сформированное дерево изображено на рис. 7 (см. ниже)
          remove_from_tree(tree, 5);  // Дерево рис. 8 (см. ниже)
          remove_from_tree(tree, 9);  // Дерево рис. 9 (см. ниже)
          remove_from_tree(tree, 0);  // Дерево рис. 10 (см. ниже)
          remove_from_tree(tree, 8);  // Дерево рис. 11 (см. ниже)
          remove_from_tree(tree, -1); // Такого элемента не было
75        // Выполняем центрированный обход
76        cout << "Инфиксный (LNR) обход\n";
77        show_tree(tree, infix);
78        // Выполняем прямой обход
79        cout << "Префиксный (NLR) обход\n";
80        show_tree(tree, prefix);
81        // Выполняем обратный обход
82        cout << "Постфиксный (LRN) обход\n";
83        show_tree(tree, postfix);
84        drop_tree(tree);
85        system("pause");
86  }
```

В результате работы программы на экране будет отображаться:

```
Инфиксный (LNR) обход
0(*2) 1(*3) 2(*1) 4(*2) 5(*3) 6(*1) 7(*3) 8(*1) 9(*1) 10(*2)
Префиксный (NLR) обход
5(*3) 2(*1) 1(*3) 0(*2) 4(*2) 8(*1) 6(*1) 7(*3) 9(*1) 10(*2)
Постфиксный (LRN) обход
0(*2) 1(*3) 4(*2) 2(*1) 7(*3) 6(*1) 10(*2) 9(*1) 8(*1) 5(*3)
```

Обход дерева вширь

Приведенный пример демонстрирует выполнение методов обхода дерева вглубь. Однако в нем отсутствует алгоритм обхода дерева вширь, то есть вывода его элементов по уровням. Этот способ является несколько более сложным и потребует внесения дополнений в применяемые структуры.

Прежде всего, в структуру **node** (файл «tree_struct.h») потребуется добавить дополнительное поле **height**. Это значение представляет собой целое число, которое соответствует высоте узла. Для листа дерева (узла, который не содержит дочерних элементов) положим высоту равной 1. Эту величину можно будет назначить и в качестве значения по умолчанию для поля

height. Чем больше уровней расположено под данным узлом, тем большее значение будет содержать рассматриваемое поле.

```
#pragma once

/// <summary>
/// Структура для хранения узлов дерева
/// </summary>
struct node
{
	/// <summary>
	/// Хранимое значение (информационное поле)
	/// </summary>
	int value;

	/// <summary>
	/// Количество соответствующих значений в исходном наборе (информационное поле)
	/// </summary>
	int count = 1;

	/// <summary>
	/// Высота поддерева, с корнем в данном узле
	/// </summary>
	int height = 1;

	/// <summary>
	/// Указатель на левое поддерево (адресное поле)
	/// </summary>
	node* left = nullptr;

	/// <summary>
	/// Указатель на правое поддерево (адресное поле)
	/// </summary>
	node* right = nullptr;
};
                        // Последующие строки кода в файле не меняются
```

В файл «tree.h» добавляются описание функции обхода дерева вширь.

```

/// <summary>
/// Обход дерева вширь
/// </summary>
/// <param name="tree">Дерево, которое необходимо обойти</param>
/// <param name="include_empty">
/// Параметр, определяющий необходимость включения в очередь пустых элементов из дерева.
/// true - пустые (отсутствующие) элементы включаются в очередь;
/// false - отсутствующие элементы пропускаются.
/// </param>
/// <returns>Очередь элементов дерева в необходимом порядке</returns>
queue wide_traverse(tree tree, bool include_empty = true);
```

Реализация последней предполагает использование вспомогательной функции с тем же названием. Кроме того, для вычисления и получения высоты также потребуется как вспомогательный функционал, так и некоторое изменение имеющегося кода. Таким образом, в файл «tree.cpp» следует включить описания вспомогательных функций.

```
99
100  /// <summary>
101  /// Безопасное получение высоты поддерева для заданного узла и
102  /// сохранение (обновление) этой информации в узле
103  /// </summary>
104  /// <param name="n">Корень дерева, для которого будет вычисляться высота</param>
105  /// <returns>Значение высоты для указанного узла дерева</returns>
106  int _get_height(node* n);
107
108  /// <summary>
109  /// Функция обхода дерева вширь
110  /// </summary>
111  /// <param name="q">Очередь, формируемая функцией, содержащая узлы дерева в нужном по-
     рядке</param>
112  /// <param name="root">Корень дерева, для которого выполняется обход</param>
113  /// <param name="include_empty">
114  /// Указывает на необходимость включения (true) в результат отсутствующих узлов дерева,
115  /// либо на отсутствие такой необходимости (false).
116  /// </param>
117  void _wide_traverse(queue& q, node*& root, bool include_empty);
118  //--------------------------------------------------------------
```

И далее.

```
147
148  bool _add_to_tree(node*& root, int value)
149  {
150        // Проверяем, существует ли дерево
151        if (root)
152        {
153              // Дерево уже существует. Определяем, куда разместить новый элемент
154              if (value == root->value)
155              {
156        // Если значение в новом элементе совпадает со значением в текущем элементе дерева
157        // Вместо добавления элемента, увеличиваем количество таких элементов
158                    root->count++;
159                    return false;
160              }
161              // В зависимости от значения нового элемента относительно текущего...
162              auto& branch =
163                    value < root->value ? root->left : root->right;
164              // ...рекурсивно размещаем элемент в левом или правом поддереве
165              bool r = _add_to_tree(branch, value);
166
167              // Если добавлялся новый элемент, пересчитываем высоту для данного узла
168              if (r) _get_height(root);
169
170              return r;
```

```
171         }
172         // Если дерева нет, то новый элемент становится корневым
173         // Создаем новый узел для размещения в дереве
174         node* newnode = new node;
175         newnode->value = value;
176         // И помещаем в качестве текущего (корневого) узла
177         root = newnode;
178         return true;
179   }
180
181   bool _remove_from_tree(node*& root, int value)
182   {
183         if (root) {
184               if (value == root->value)
185               {
186                     // Удаляемый элемент найден.
187                     if (!root->left && !root->right)
188                           // Если это лист, просто удаляем его.
189                           _del(root);
190                     else
191                           // Иначе выполняем перенос
192                           _move_node(root);
193                     return true;
194               }
195               // Выполняем рекурсивный поиск удаляемого элемента в левом или правом поддереве
196               auto& subtree =
197                     value < root->value ? root->left : root->right;
198               bool r = _remove_from_tree(subtree, value);
199
200               // Если элемент был удален, пересчитываем высоту для данного узла
201               if (r) _get_height(root);
202               return r;
203         }
204         return false;
205   }
206
207   int _find_in_tree(const node* root, int value)
208   {
209         // Если корень пуст, значит искомое значение не найдено, количество = 0
210         if (!root) return 0;
211         // Если значение имеется в корневом элементе, возвращаем количество
212         if (root->value == value) return root->count;
213         // Иначе продолжаем рекурсивный поиск в поддеревьях
214         if (value < root->value)
215               return _find_in_tree(root->left, value);
216         return _find_in_tree(root->right, value);
217   }
218
219   void _drop_tree(node*& root)
220   {
221         if (root) // Если есть, что удалять
222         {
223               // Рекурсивно удаляется левое поддерево
224               _drop_tree(root->left);
225               // Рекурсивно удаляется правое поддерево
226               _drop_tree(root->right);
227               // Удаляется корневой элемент
228               _del(root);
```

```
229         }
230     }
231
232     void _move_node(node* rem)
233     {
234         auto parent = _find_prev_nearest(rem);
235         // Определяем, будем ли работать с левым поддеревом
236         auto is_lft = _is_left(rem);
237         // Если ближайший узел находится сразу за удаляемым,
238         // инверсируем признак поддерева
239         if (parent == rem) is_lft = !is_lft;
240         // Находим в одном из поддеревьев элемент, ближайший к удаляемому
241         // Он является самым правым в левом поддереве
242         // и самым левым – в правом.
243         auto& to_remove = is_lft ? parent->right : parent->left;
244         // Если есть продолжение дерева после фактически удаляемого элемента
245         node* child = nullptr;
246         if (to_remove)
247             child = is_lft ? to_remove->right : to_remove->left;
248         // Копируем значения(!) из ближайшего (к удаляемому) узла в удаляемый узел
249         // При этом сам удаляемый узел останется в памяти (с новым значением),
250         // Лишний элемент будет удален у дерева снизу.
251         _copy_value(rem, to_remove);
252         // Выполняем удаление лишнего узла
253         _del(to_remove);
254         // Прицепляем «остаток» дерева к родительскому узлу
255         to_remove = child;
256         // Выполняем пересчет высот
257         _get_height(parent);
258     }
```

Также с появлением значения высоты, стало возможным несколько оптимизировать функцию выбора поддерева, из которого выбирается узел для переноса на место удаляемого значения (**`_is_left()`**). Ее можно переписать следующим образом.

```
291
292     inline bool _is_left(const node* n)
293     {
294         auto lh = _get_height(n->left);
295         auto rh = _get_height(n->right);
296         return lh >= rh;
297     }
```

В заключении, добавляем реализации функций подсчета высоты и собственно выполнения обхода вширь.

```
370
371     int _get_height(node* n)
372     {
373         // Если узел пуст, сразу прерываем вычисления и возвращаем высоту = 0.
```

```
374         if (!n) return 0;
375         // Определяем сохраненные высоты левого и правого поддеревьев (если они существуют)
376         auto lh = n->left ? n->left->height : 0;
377         auto rh = n->right ? n->right->height : 0;
378         // Высоту текущего узла рассчитываем как наибольшую из высот поддеревьев + 1
379         return n->height = 1 + (lh > rh ? lh : rh);
380 }
381
382 queue wide_traverse(tree t, bool include_empty)
383 {
384         queue q;
385         wide_traverse(q, t.root, include_empty);
386         return q;
387 }
388
389 void _wide_traverse(queue& queue, node*& r, bool include_empty)
390 {
391         // Проверка того, что дерево существует
392         if (!r) return;
393         // Корень дерева добавляется в очередь
394         enqueue(queue, r);
395         // Получение указателя на начало очереди
396         auto q = queue.head;
397         // Вычисление количества максимального числа узлов для дерева данной высоты
398         // (Операция «<<» - побитовый сдвиг влево - используется для получения нужной степени числа 2.)
399         const auto cnt = 1 << (r->height - 1);
400         // Производится проход по очереди
401         for (int i = 1; i < cnt && q; i++)
402         {
403                 // Получаем указатели на левый и правый подузлы для текущего
404                 auto ln = q->node ? q->node->left : nullptr;
405                 auto rn = q->node ? q->node->right : nullptr;
406                 // Добавляем узлы в очередь если они не пустые, либо
407                 // параметр include_empty позволяет сохранять там пустые значения.
408                 // (Последнее необходимо для организации вывода на экран структуры дерева)
409                 if (ln || include_empty) enqueue(queue, ln);
410                 if (rn || include_empty) enqueue(queue, rn);
411                 q = q->next;
412         }
413 }
```

Функция обхода вширь поддерживает два режима работы: без включения пустых узлов и с добавлением пропусков. Последний вариант будет особенно полезен для вывода дерева по уровням.

Для просмотра результатов обхода дерева с помощью только что добавленных функций, следует внести ряд изменений в «main.cpp»:

- добавить дополнительный тип обхода дерева в **`enum _traverse_type`**;
- добавить новый вариант обхода дерева в функцию **`show_tree()`**.

Также в конце программы приводится пример построения дерева по упорядоченному набору значений.

```
1   #include <iostream>
2   #include "queue.h"
3   #include "tree.h"
4   using namespace std;
5
6   //Вспомогательные функции для создания и отображения дерева
7
8   /// <summary>
9   /// Перечисление возможных видов обхода дерева.
10  /// </summary>
    enum traverse_type { prefix, infix, postfix };
11  enum traverse_type { prefix, infix, postfix, wide };
12
13  /// <summary>
14  /// Функция отображает на экране элементы из очереди, составленной из узлов дерева
15  /// </summary>
16  /// <param name="tree">Дерево, которое необходимо вывести на экран</param>
17  /// <param name="tt">
18  /// Указывает на тип выполнения обхода дерева. По умолчание выбран обход по уровням (вширь).
19  /// </param>
20  /// <param name="show_empty">
21  /// Указывает на необходимость отображения пустых узлов дерева (true, по умолчанию),
22  /// либо вывода только существующих элементов (false).
23  /// Применяется только в совокупности с параметром tt = wide
24  /// </param>
25  void show_tree(tree tree,traverse_type tt = wide,
26                 bool show_empty = true)
27  {
28      // Формирование очереди для выполнения обхода определенного вида
29      queue tq;
30      switch (tt)
31      {
32      case prefix:    tq = prefix_traverse(tree);     break;
33      case infix:     tq = infix_traverse(tree);      break;
34      case postfix:   tq = postfix_traverse(tree);    break;
35      case wide: tq = wide_traverse(tree, show_empty); break;
36      }
37      // Получение элемента с головы очереди для начала обхода
38      auto got = tq.head;
39      // Пока в очереди есть элементы (полученный элемент не равен nullptr)
40      while (got)
41      {
42          auto n = got->node;
43          // Если элемент не пуст, выводится его значение
44          // и в скобках количество таких значений в исходном наборе
45          if (n) cout << n->value << "(*"
46              << n->count << ")";
47          // Если элемент пуст, вместо его значения выводится прочерк
48          else cout << "-";
49          // Переходим к следующему по очереди элементу
50          got = got->next;
51          // Выводим символ-разделитель: пробел, если в очереди еще есть элементы
```

```
52                  // или символ перехода на новую строку, если элемент был последним.
53                  cout << (got ? " " : "\n");
54          }
55          // Удаление вспомогательной очереди
56          clear(tq);
57  }
58
59  /// <summary>
60  /// Функция создает дерево, последовательно заполняя его значениями из указанного массива
61  /// </summary>
62  /// <param name="t">Дерево, которое требуется сформировать</param>
63  /// <param name="arr">Массив элементов, которые требуется разместить в дереве</param>
64  /// <param name="elem_count">Количество элементов массива</param>
65  void create_tree(tree& t, const int arr[], size_t elem_count)
66  {
67          for (auto i = 0; i < elem_count; i++)
68          {
69                  add_to_tree(t, arr[i]);
70          }
71  }
72  //--------------------------------------------------------------
73
74  void main()
75  {
76          setlocale(LC_ALL, "Rus");
77          // Вспомогательный массив, содержащий элементы, который будут помещены в дерево
78          int arr[] = { 5, 2, 8, 1, 4, 0, 6, 9, 10,
79                        7, 1, 1, 7, 7, 0, 4, 5, 5, 10 };
80          tree tree;
81          create_tree(tree, arr, sizeof(arr) / sizeof(arr[0]));
82          // Выполняем центрированный обход
83          cout << "Инфиксный (LNR) обход\n";
84          show_tree(tree, infix);
85          // Выполняем прямой обход
86          cout << "Префиксный (NLR) обход\n";
87          show_tree(tree, prefix);
88          // Выполняем обратный обход
89          cout << "Постфиксный (LRN) обход\n";
90          show_tree(tree, postfix);
91          // Выполняем обход вширь с добавлением всех узлов, в том числе пустых
92          cout << "Обход вширь\n";
93          show_tree(tree);
94          // Выполняем обход вширь с размещением в очереди только существующих элементов дерева
95          cout << "Обход вширь (без пустых)\n";
96          show_tree(tree, wide, false);
97
98          cout << "Удаление числа 5\n";
99          remove_from_tree(tree, 5);   // Дерево рис. 8 (см. выше)
100         show_tree(tree);
101         cout << "Удаление числа 9\n";
102         remove_from_tree(tree, 9);   // Дерево рис. 9 (см. выше)
103         show_tree(tree);
104         cout << "Удаление числа 0\n";
105         remove_from_tree(tree, 0);   // Дерево рис. 10 (см. выше)
106         show_tree(tree);
107         cout << "Удаление числа 8\n";
108         remove_from_tree(tree, 8);   // Дерево рис. 11 (см. выше)
109         show_tree(tree);
```

```
110         cout << "Удаление несуществующего числа -1\n";
111         remove_from_tree(tree, -1); // Такого элемента не было
112         show_tree(tree);
113         drop_tree(tree);
114         system("pause");
115 }
```

На экран при этом будет выведена следующая информация.

```
Инфиксный (LNR) обход
0(*2) 1(*3) 2(*1) 4(*2) 5(*3) 6(*1) 7(*3) 8(*1) 9(*1) 10(*2)
Префиксный (NLR) обход
5(*3) 2(*1) 1(*3) 0(*2) 4(*2) 8(*1) 6(*1) 7(*3) 9(*1) 10(*2)
Постфиксный (LRN) обход
0(*2) 1(*3) 4(*2) 2(*1) 7(*3) 6(*1) 10(*2) 9(*1) 8(*1) 5(*3)
Обход вширь
5(*3) 2(*1) 8(*1) 1(*3) 4(*2) 6(*1) 9(*1) 0(*2) - - - - 7(*3) - 10(*2)
Обход вширь (без пустых)
5(*3) 2(*1) 8(*1) 1(*3) 4(*2) 6(*1) 9(*1) 0(*2) 7(*3) 10(*2)
Удаление числа 5
4(*2) 2(*1) 8(*1) 1(*3) - 6(*1) 9(*1) 0(*2) - - - - 7(*3) - 10(*2)
Удаление числа 9
4(*2) 2(*1) 8(*1) 1(*3) - 6(*1) 10(*2) 0(*2) - - - - 7(*3) - -
Удаление числа 0
4(*2) 2(*1) 8(*1) 1(*3) - 6(*1) 10(*2) - - - - - 7(*3) - -
Удаление числа 8
4(*2) 2(*1) 7(*3) 1(*3) - 6(*1) 10(*2) - - - - - - - - -
Удаление несуществующего числа -1
4(*2) 2(*1) 7(*3) 1(*3) - 6(*1) 10(*2) - - - - - - - - -
```

При сравнении полученного результата обхода дерева вширь (до удаления из него элементов) с графическим представлением дерева (см. рис. 13), легко заметить, что перебор узлов действительно осуществляется по уровням.

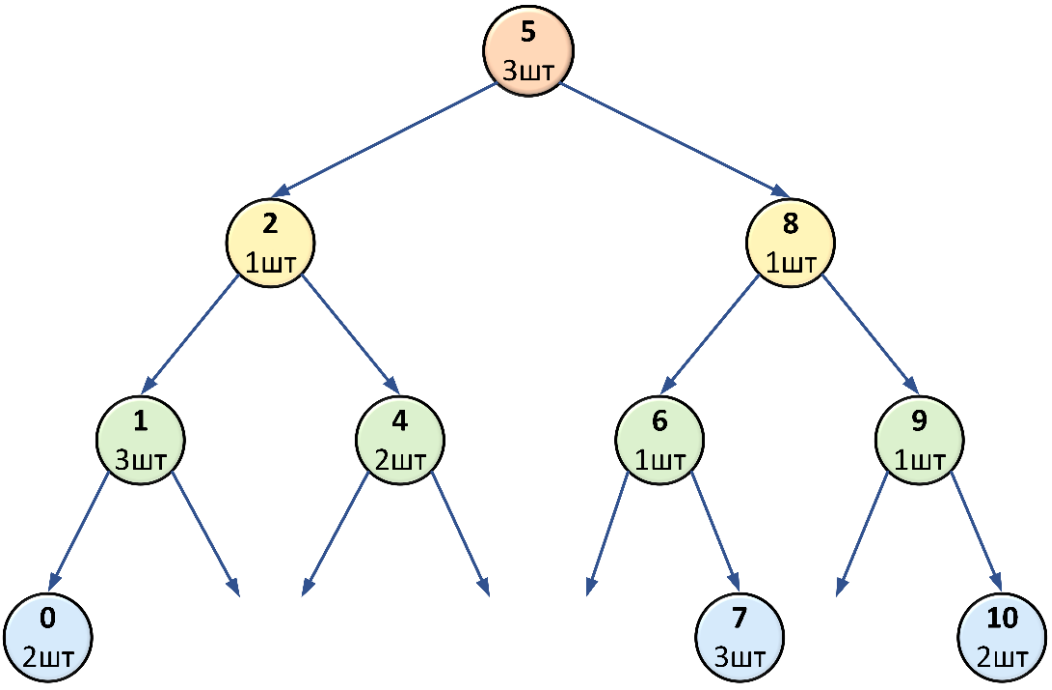


Рис. 13. Структура созданного дерева

Балансировка бинарного дерева (АВЛ-деревья)

К сожалению, при построении бинарных деревьев поиска указанным выше способом, не всегда можно рассчитывать на то, что элементы в нем будут расположены оптимальным образом.

Для демонстрации возможных проблем добавим в конце функции **`main()`** следующий блок кода.

```
113         drop_tree(tree);
114
115         // Тестирование работы с деревом на упорядоченном наборе значений
116         int arr2[] = { 1, 1, 1, 2, 3, 4, 5, 6, 6, 7, 7, 7 };
117         // Формирование дерева:
118         create_tree(tree, arr2, sizeof arr2 / sizeof arr2[0]);
119         // Просмотр дерева по уровням
120         cout << "Просмотр дерева, построенного "
121              << "по упорядоченной последовательности\n";
122         show_tree(tree);
123         drop_tree(tree);
124         system("pause");
125 }
```

Здесь строится дерево по упорядоченному набору значений. Можно заметить, что в этом случае все вершины оказываются справа от предка и динамическая структура данных фактически превращается в список.

Действительно, при использовании упорядоченного массива **`int arr2[] = { 0, 1, 1, 1, 2, 3, 4, 5, 6, 6, 7, 7, 7 };`** на экране будет отображаться:

```
Просмотр дерева, построенного по упорядоченной последовательности
1(*3) - 2(*1) - - - 3(*1) - - - - - - - 4(*1) - - - - - - - - - - - -
- - - - 5(*1) - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - 6(*2) - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - 7(*3)
```

Это соответствует дереву на рис. 14.

В таком виде, дерево уже не подходит для выполнения своей основной задачи – ускорения поиска, поскольку в худшем случае (например, при попытке поиска отсутствующего элемента 10) придется проверять все его узлы.

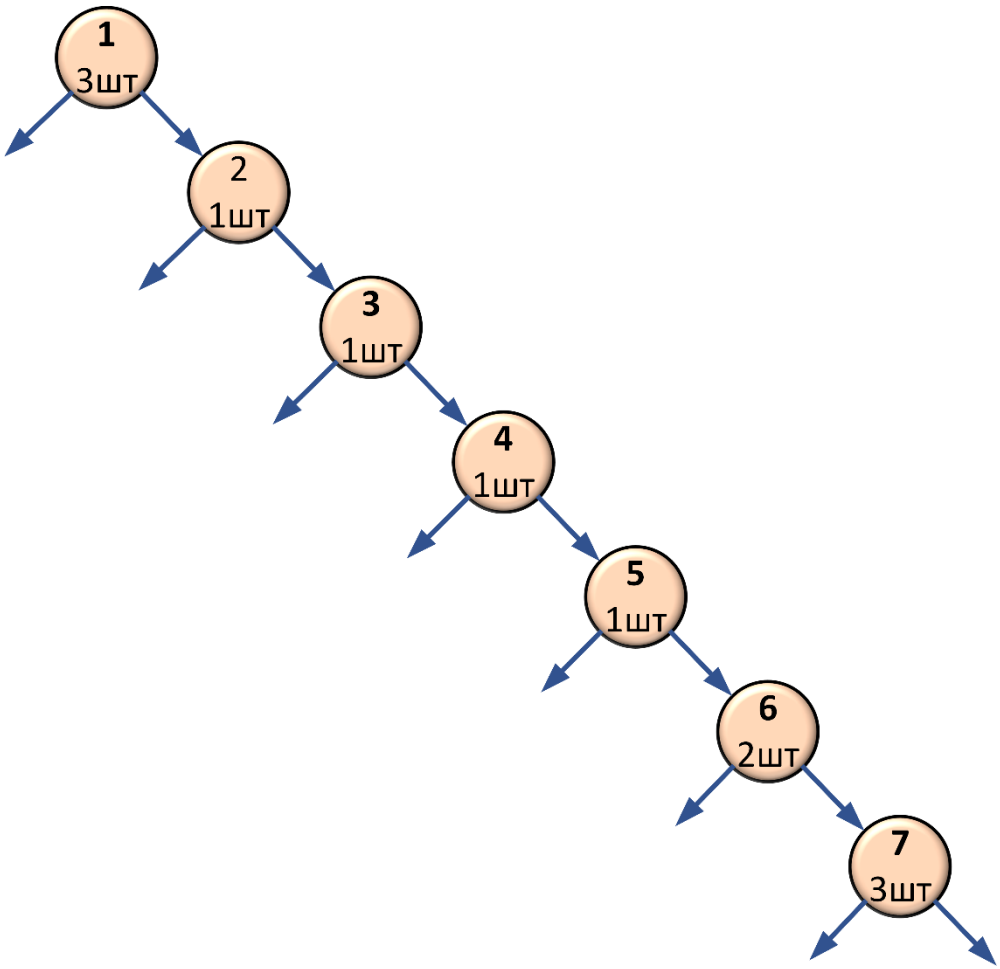


Рис. 14. Дерево, построенное по упорядоченному набору значений

Получилось, что всего лишь 7 существующих в дереве элементов заняли в нем целых 7 уровней.

Для того, чтобы уменьшить высоту дерева, потребуется применить специальные алгоритмы, направленные на перераспределение элементов в динамической структуре, при сохранении основного правила их расположения (значения, меньшие корня – слева от него, большие – справа). Такой алгоритм называется алгоритмом балансировки дерева и применяется, как правило, в момент изменения конструкции, то есть при добавлении или удалении элементов.

*Двоичное дерево поиска называется **АВЛ-деревом**, когда для каждой его вершины высота ее двух поддеревьев отличается не более, чем на 1.* Аббревиатура АВЛ образована первыми буквами фамилий создателей алгоритма – советских ученых Г. М. **А**дельсон-**В**ельского и Е. М. **Л**андиса.

Рассмотрим алгоритм балансировки на следующем примере. Пусть дано дерево (или фрагмент дерева), представленное на рис. 15. В скобках рядом с названиями вершин расположено значение высоты для данного узла.

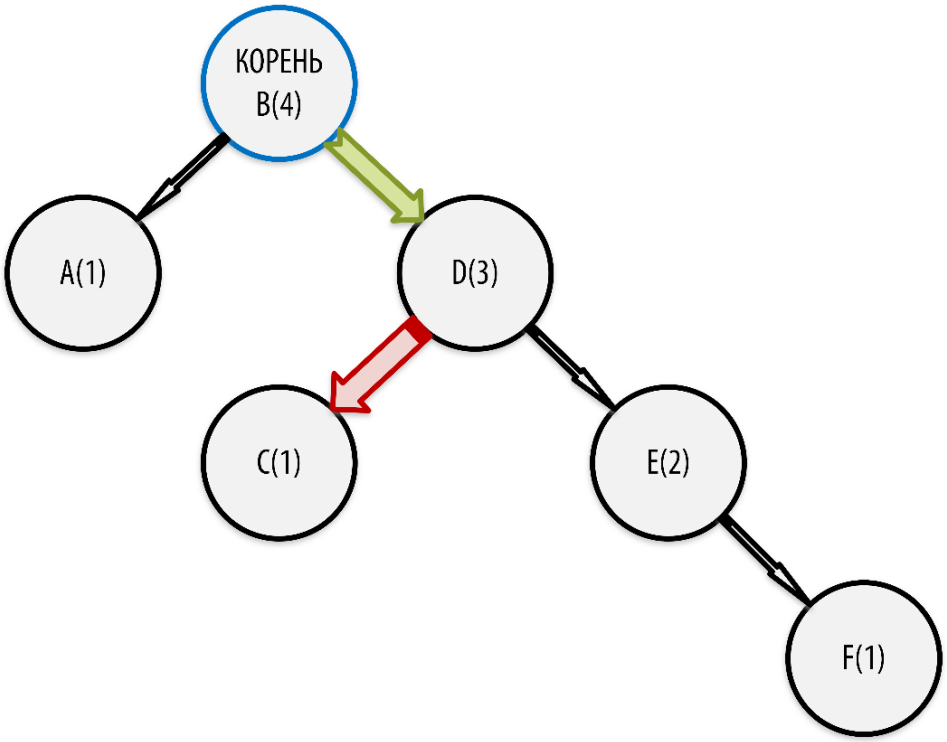


Рис. 15. Исходное дерево 1

Поскольку для корня (B) высота левого поддерева (A – 1) отличается от высоты правого поддерева (D – 3) на 2 единицы, данное дерево имеет неоптимальную высоту и требует балансировки. Так как правая ветвь длиннее левой, к дереву применяется так называемый малый левый поворот. Для этого выполняются следующие операции (см. рис. 16).

1. Дочерний узел относительно корня со стороны более длинной ветви (в данном случае – D) объявляется новым корнем.
2. Указатель на левое поддерево для нового корня (D) устанавливается на узел, который являлся корневым ранее (B).
3. Указатель на правое поддерево у бывшего корня (B) перемещается на левое поддерево для узла D – узел C.

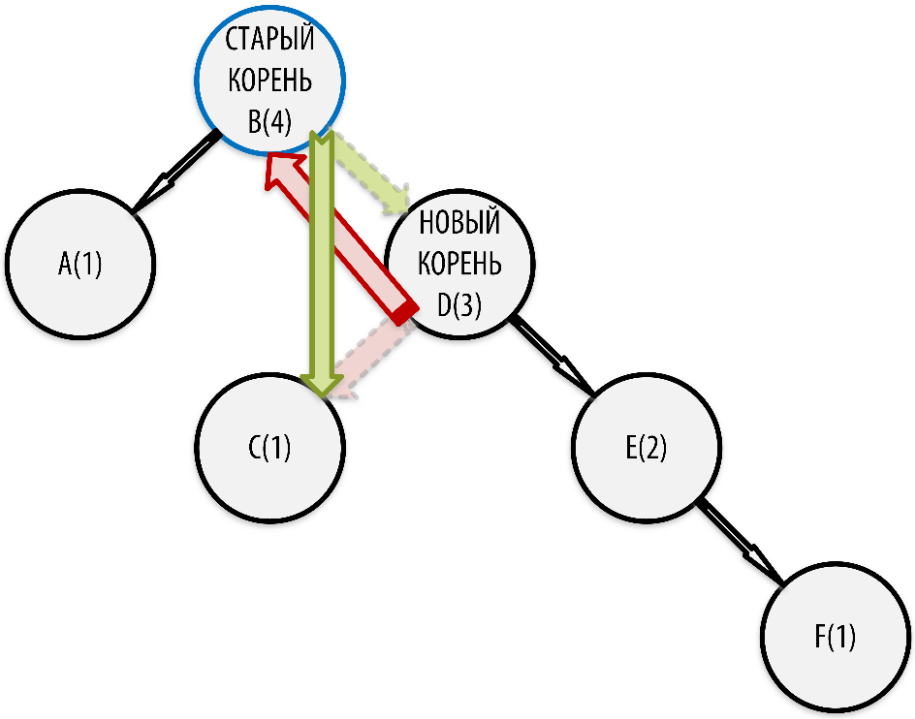


Рис. 16. Изменение корня и связей между узлами

В результате будет получена структура, показанная на рис. 17.

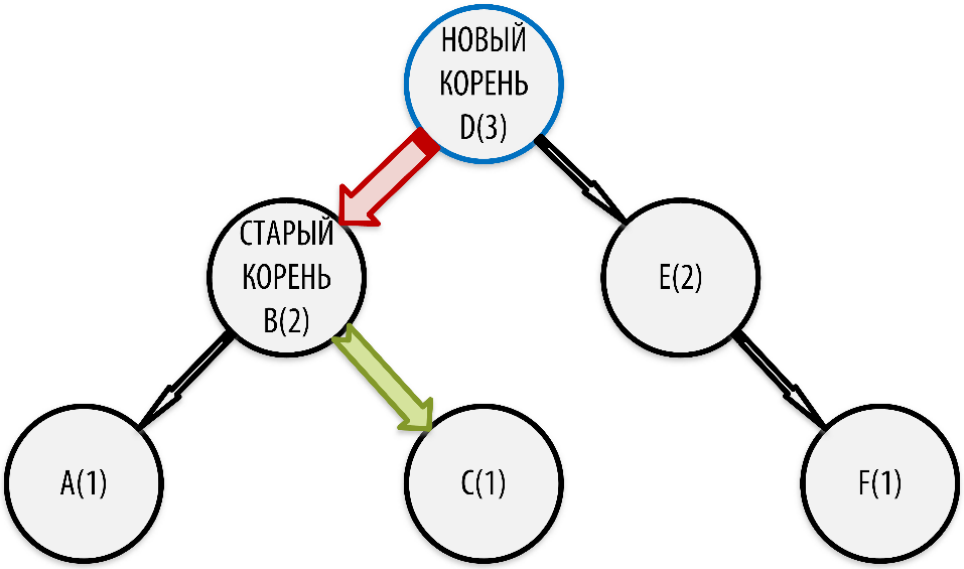


Рис. 17. Сбалансированное дерево поиска

В новом варианте дерева количество уровней уменьшилось на 1 и нет узлов, нарушающих требование сбалансированности. Кроме того, сохранилось и основное свойство деревьев поиска: значения узлов слева меньше корня, а справа – больше.

В случае, когда левая ветвь дерева длиннее правой, выполняется правый поворот, по аналогии с описанным выше методом, лишь меняя местами левую и правую стороны.

В ряде случаев, тем не менее, выполнение малых поворотов не приводит к желаемым результатам.

Рассмотрим пример (рис. 18).

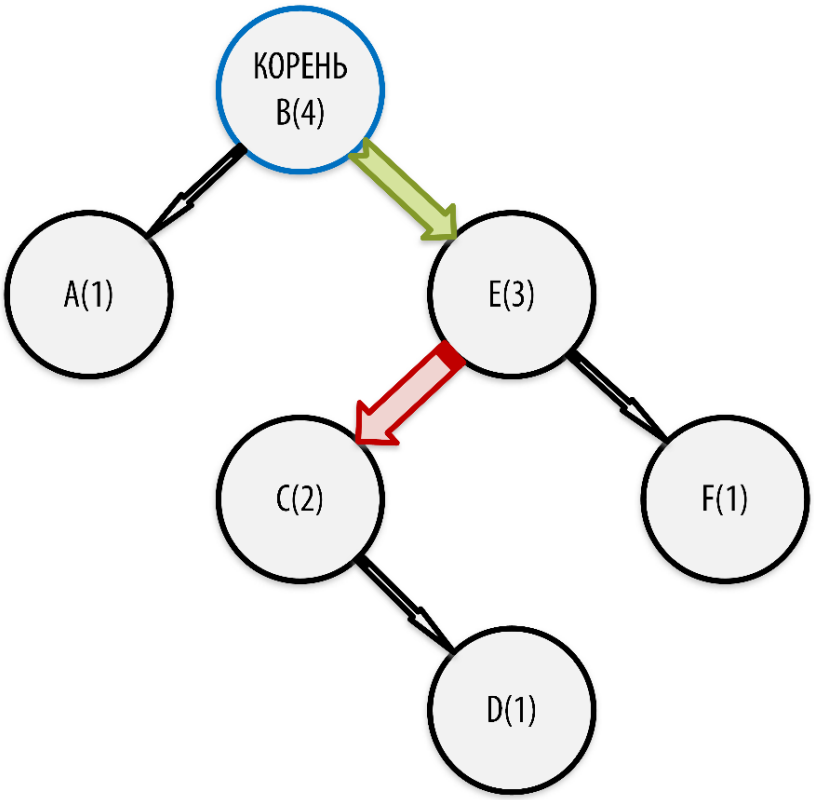
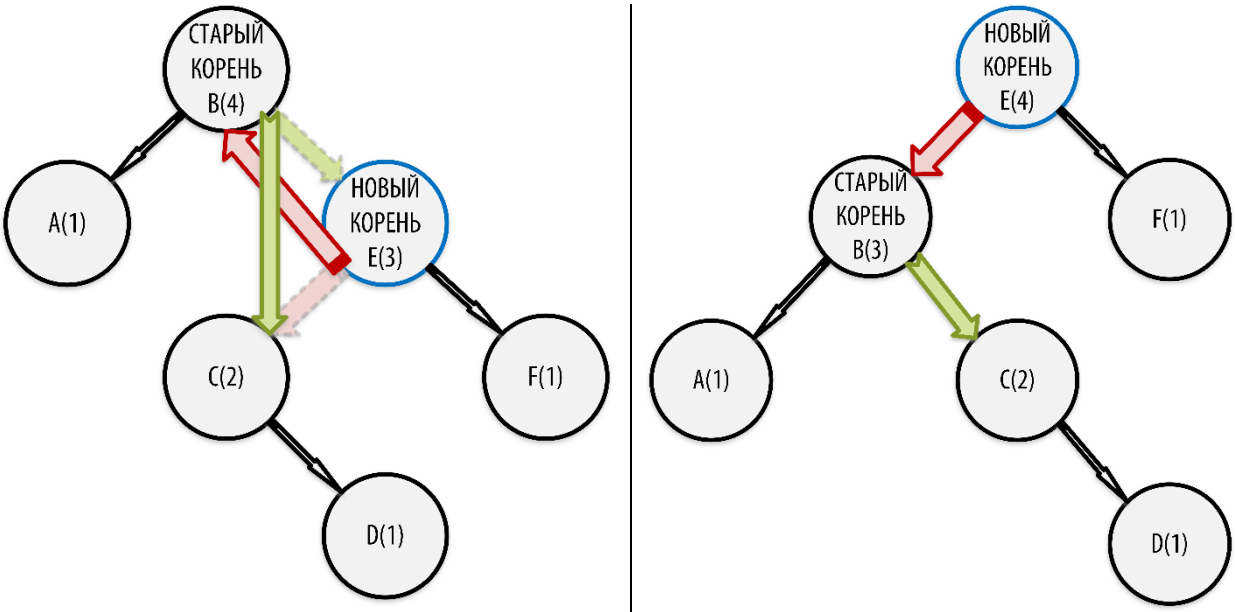


Рис. 18. Исходное дерево 2

У него, как и в предыдущем случае, имеется дисбаланс между ветвями, и правая ветвь (E – 3) длиннее левой (A – 1). Однако в отличие от исходного дерева 1, у более длинной ветви, начинающейся с узла E второго дерева, уже левая ветвь оказывается длиннее правой (C – 2 > F – 1).

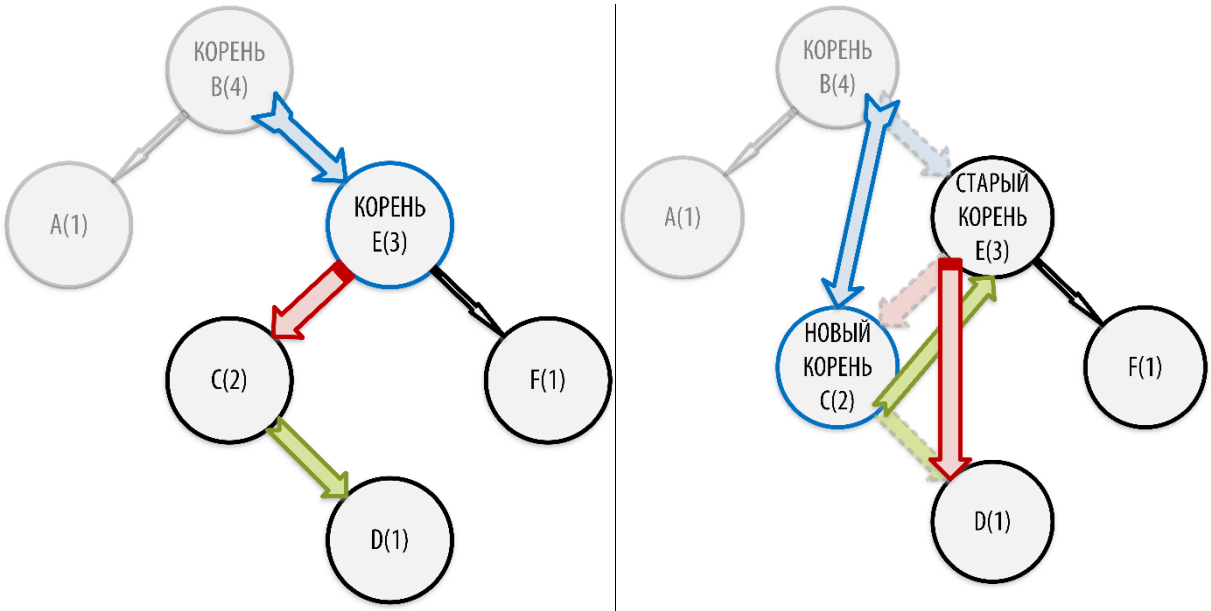
Если мы применим к исходному дереву 2 малый левый поворот (рис. 19а), окажется, что полученная конструкция, по-прежнему не является сбалансированной (рис. 19б).



а) изменение корня и связей | б) полученная конструкция

Рис. 19. Выполнение малого левого поворота

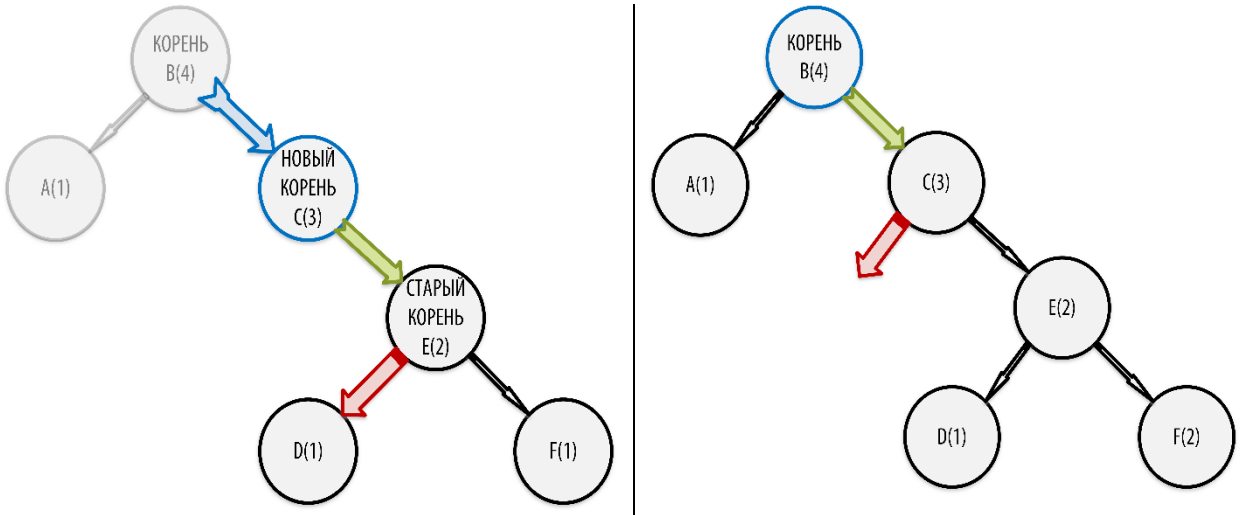
Для исходных деревьев второго вида, необходимо использовать большой поворот, состоящий из двух разнонаправленных малых поворотов, примененных к разным узлам. Рассмотрим алгоритм более подробно. Пусть опять имеется дерево, представленное на рис. 18. Рассмотрим его правое (более высокое) поддерево, начиная с узла E (рис. 20а) и выполним для него операцию малого *правого* поворота (рис. 20б).



а) выбор поддерева от узла E | б) изменение связей и корня

Рис. 20. Малый правый поворот поддерева

В результате будет получена структура, представленная на рис. 21а, идентичная исходному дереву первого вида (рис. 15).



а) результат относительно узла C | б) результат относительно узла B

Рис. 21. Результат малого правого поворота для поддерева

Далее для полученного дерева можно применять описанный выше алгоритм левого поворота, начиная с корня B (рис. 21б).

Промежуточные этапы выполнения поворота практически полностью идентичны показанным на рис. 15–17, поэтому приведем только конечный результат балансировки на рис. 22.

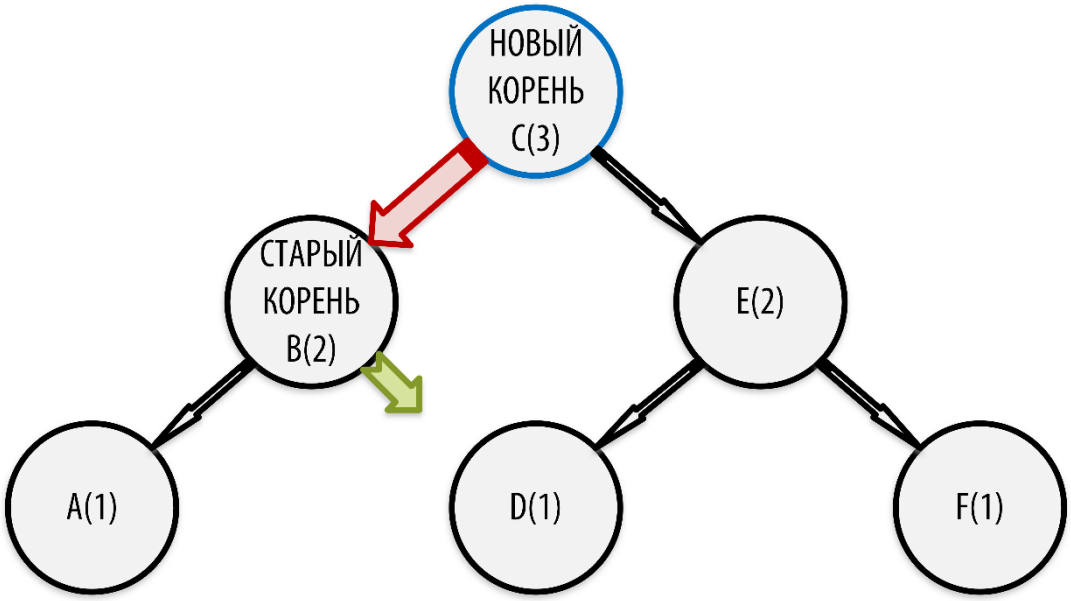


Рис. 22. Сбалансированное дерево

Рассмотрим теперь реализацию алгоритма балансировки дерева. Для этого необходимо будет внести изменения в файл «tree.cpp». Во-первых, потребуется добавить заголовки вспомогательных функций.

```
114  /// <summary>
115  /// Безопасное получение высоты поддерева для заданного узла и
116  /// сохранение (обновление) информации в узле
117  /// </summary>
118  /// <param name="n">Корень дерева, для которого будет вычисляться высота</param>
119  /// <returns>Значение высоты для указанного узла дерева</returns>
120  int _get_height(node* n);
121
122  /// <summary>
123  /// Функция балансировки дерева или его части
124  /// </summary>
125  /// <param name="n">Вершина дерева, для которой выполняется балансировка</param>
126  void _balance(node*& n);
127
128  /// <summary>
129  /// Функция выполнения малого левого поворота
130  /// </summary>
131  /// <param name="n">Вершина дерева, для которой выполняется поворот</param>
132  void _small_left_rotate(node*& n);
133
134  /// <summary>
135  /// Функция выполнения малого правого поворота
136  /// </summary>
137  /// <param name="n">Вершина дерева, для которой выполняется поворот</param>
138  void _small_right_rotate(node*& n);
139
140  /// <summary>
141  /// Функция выполнения большого левого поворота
142  /// </summary>
143  /// <param name="n">Вершина дерева, для которой выполняется поворот</param>
144  void _big_left_rotate(node*& n);
145
146  /// <summary>
147  /// Функция выполнения большого правого поворота
148  /// </summary>
149  /// <param name="n">Вершина дерева, для которой выполняется поворот</param>
150  void _big_right_rotate(node*& n);
151  //-----------------------------------------------------------
```

Во-вторых, необходимо вызвать функцию балансировки дерева при вставке, и удалении его узлов.

```
180
181  bool _add_to_tree(node*& root, int value)
182  {
183      // Проверяем, существует ли дерево
184      if (root)
185      {
186          // Дерево уже существует. Определяем, куда разместить новый элемент
187          if (value == root->value)
```

```
188                 {
189         // Если значение в новом элементе совпадает со значением в текущем элементе дерева
190         // Вместо добавления элемента, увеличиваем количество таких элементов
191                     root->count++;
192                     return false;
193                 }
194                 // В зависимости от значения нового элемента относительно текущего…
195                 auto& branch =
196                     value < root->value ? root->left : root->right;
197                 // …рекурсивно размещаем элемент в левом или правом поддереве
198                 bool r = _add_to_tree(branch, value);
199
                    // Если добавлялся новый элемент, пересчитываем высоту для данного узла
                    if (r) _get_height(root);

200                 // Выполнение балансировки дерева после добавления нового узла
201                 // Пересчет высоты узлов внесен в функцию балансировки
202                 if (r) _balance(root);
203                 return r;
204         }
205         // Если дерева нет, то новый элемент становится корневым
206         // Создаем новый узел для размещения в дереве
207         node* newnode = new node;
208         newnode->value = value;
209         // И помещаем в качестве текущего (корневого) узла
210         root = newnode;
211         return true;
212 }
213
214 bool _remove_from_tree(node*& root, int value)
215 {
216         if (root) {
217             if (value == root->value)
218             {
219                 // Удаляемый элемент найден.
220                 if (!root->left && !root->right)
221                     // Если это лист, просто удаляем его.
222                     _del(root);
223                 else
224                     // Иначе выполняем перенос
225                     _move_node(root);
226                 return true;
227             }
228             // Выполняем рекурсивный поиск удаляемого элемента в левом или правом поддереве
229             auto& subtree =
230                 value < root->value ? root->left : root->right;
231             bool r = _remove_from_tree(subtree, value);
232
                // Если элемент был удален, пересчитываем высоту для данного узла
                if (r) _get_height(root);
233             // Если элемент был удален, выполняем балансировку для корневого узла
234             if (r) _balance(root);
235             return r;
236         }
237         return false;
238 }
239
240 int _find_in_tree(const node* root, int value)
```

```
241 {
242       // Если корень пуст, значит искомое значение не найдено, количество = 0
243       if (!root) return 0;
244       // Если значение имеется в корневом элементе, возвращаем количество
245       if (root->value == value) return root->count;
246       // Иначе продолжаем рекурсивный поиск в поддеревьях
247       if (value < root->value)
248             return _find_in_tree(root->left, value);
249       return _find_in_tree(root->right, value);
250 }
251
252 void _drop_tree(node*& root)
253 {
254       if (root)  // Если есть, что удалять
255       {
256             // Рекурсивно удаляется левое поддерево
257             _drop_tree(root->left);
258             // Рекурсивно удаляется правое поддерево
259             _drop_tree(root->right);
260             // Удаляется корневой элемент
261             delete root;
262             root = nullptr;
263       }
264 }
265
266 void _move_node(node* rem)
267 {
268       auto parent = _find_prev_nearest(rem);
269       // Определяем, будем ли работать с левым поддеревом
270       auto is_lft = _is_left(rem);
271       // Если ближайший узел находится сразу за удаляемым,
272       // инверсируем признак поддерева
273       if (parent == rem) is_lft = !is_lft;
274       // Находим в одном из поддеревьев элемент, ближайший к удаляемому
275       // Он является самым правым в левом поддереве
276       // и самым левым – в правом.
277       auto& to_remove = is_lft ? parent->right : parent->left;
278       // Если есть продолжение дерева после фактически удаляемого элемента
279       node* child = nullptr;
280       if (to_remove)
281             child = is_lft ? to_remove->right : to_remove->left;
282       // Копируем значения(!) из ближайшего (к удаляемому) узла в удаляемый узел
283       // При этом сам удаляемый узел останется в памяти (с новым значением),
284       // Лишний элемент будет удален у дерева снизу.
285       copy_value(rem, to_remove);
286       // Выполняем удаление лишнего узла
287       del(to_remove);
288       // Прицепляем остаток дерева к родительскому узлу
289       to_remove = child;
          // Выполняем пересчет высот
          get_height(parent);
290       // Выполняем балансировку дерева относительно вершины parent
291       balance(parent);
292 }
```

В-третьих, остается разместить реализацию новых функций.

```
448
449 void _balance(node*& n)
450 {
451         // Проверяем, что переданный узел существует
452         if (!n) return;
453         // Определяем высоты левого и правого поддеревьев
454         auto lh = _get_height(n->left);
455         auto rh = _get_height(n->right);
456         // Определяем разность высот двух поддеревьев
457         auto dh = lh - rh;
458         // Находим абсолютную величину разности
459         dh = dh >= 0 ? dh : -dh;
460         // Если разность высот 2 или более, требуется балансировка
461         if (dh >= 2)
462         {
463                 // Если правая ветвь длиннее, делаем левый поворот
464                 if (rh > lh)
465                 {
466                         // Определим высоты поддеревьев в правом поддереве
467                         auto rlh = _get_height(n->right->left);
468                         auto rrh = _get_height(n->right->right);
469                         // Если правое под-поддерево больше, делаем малый поворот
470                         if (rrh > rlh) _small_left_rotate(n);
471                         // иначе - большой
472                         else _big_left_rotate(n);
473                 }
474                 // Будем делать правый поворот, если левая ветвь больше
475                 else
476                 {
477                         // Определим высоты поддеревьев в левом поддереве
478                         auto llh = _get_height(n->left->left);
479                         auto lrh = _get_height(n->left->right);
480                         // Если левое под-поддерево больше, делаем малый поворот
481                         if (llh > lrh) _small_right_rotate(n);
482                         // иначе - большой
483                         else _big_right_rotate(n);
484                 }
485         }
486         else
487         {
488                 // Разбалансировки нет. Просто уточняем высоту текущего узла
489                 get_height(n);
490         }
491 }
492
493 void _small_left_rotate(node*& n)
494 {
495         // Узел правого поддерева будет новым корнем
496         node* new_root = n->right;
497         // Указатель на правое поддерево смещаем на узел, который стоит слева от нового корня
498         n->right = new_root->left;
499         // Указатель на левое поддерево для нового корня устанавливаем на старый корень
500         new_root->left = n;
501         // Выполняем пересчет высоты для старого корня
502         get_height(n);
503         // Выполняем фактический перенос корня
504         n = new_root;
505         // Выполняем пересчет высоты для нового корня
```

```
506         get_height(n);
507     }
508
509     void _small_right_rotate(node*& n)
510     {
511         // Данная функция является полностью зеркальной относительно предыдущей
512         // Узел левого поддерева будет новым корнем
513         node* new_root = n->left;
514         // Указатель на левое поддерево смещаем на узел, который стоит справа от нового корня
515         n->left = new_root->right;
516         // Указатель на правое поддерево для нового корня устанавливаем на старый корень
517         new_root->right = n;
518         // Выполняем пересчет высоты для старого корня
519         get_height(n);
520         // Выполняем фактический перенос корня
521         n = new_root;
522         // Выполняем пересчет высоты для нового корня
523         get_height(n);
524     }
525
526     void _big_left_rotate(node*& n)
527     {
528         small_right_rotate(n->right);
529         small_left_rotate(n);
530     }
531
532     void _big_right_rotate(node*& n)
533     {
534         // Данная функция является полностью зеркальной относительно предыдущей
535         small_left_rotate(n->left);
536         small_right_rotate(n);
537     }
```

При сохранении функции **`main()`** в неизменном относительно предыдущего примера виде, на экране получим следующий вывод.

```
Инфиксный (LNR) обход
0(*2) 1(*3) 2(*1) 4(*2) 5(*3) 6(*1) 7(*3) 8(*1) 9(*1) 10(*2)
Префиксный (NLR) обход
5(*3) 2(*1) 1(*3) 0(*2) 4(*2) 8(*1) 6(*1) 7(*3) 9(*1) 10(*2)
Постфиксный (LRN) обход
0(*2) 1(*3) 4(*2) 2(*1) 7(*3) 6(*1) 10(*2) 9(*1) 8(*1) 5(*3)
Обход вширь
5(*3) 2(*1) 8(*1) 1(*3) 4(*2) 6(*1) 9(*1) 0(*2) - - - - 7(*3) - 10(*2)
Обход вширь (без пустых)
5(*3) 2(*1) 8(*1) 1(*3) 4(*2) 6(*1) 9(*1) 0(*2) 7(*3) 10(*2)
Удаление числа 5
4(*2) 2(*1) 8(*1) - - 6(*1) 9(*1) - - - - - 7(*3) - 10(*2)
Удаление числа 9
6(*1) 4(*2) 8(*1) 2(*1) - 7(*3) 10(*2)
Удаление числа 0
6(*1) 4(*2) 8(*1) 2(*1) - 7(*3) 10(*2)
Удаление числа 8
6(*1) 4(*2) 7(*3) 2(*1) - - 10(*2)
Удаление несуществующего числа -1
6(*1) 4(*2) 7(*3) 2(*1) - - 10(*2)
Просмотр дерева, построенного по упорядоченной последовательности
4(*1) 2(*1) 6(*2) 1(*3) 3(*1) 5(*1) 7(*3)
```

Легко заметить, что теперь обозначаемые прочерками пропуски отсутствуют, а значит, элементы заняли все возможные места и расположились всего на трех уровнях – это минимально возможное значение для данного числа узлов. Это значит, что дерево точно получилось сбалансированным (см. рис. 23 и сравните с рис. 14).

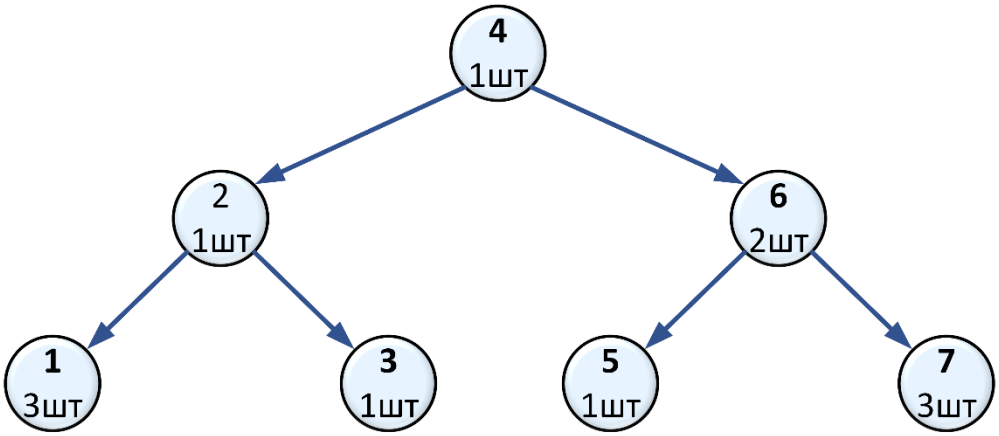


Рис. 23. Сбалансированное дерево поиска

Отметим, что в определенных случаях, даже при использовании приведенного алгоритма, элементы могут занять и большее число уровней, однако при этом все равно будет соблюдаться условие сбалансированности, то есть ни для каких двух поддеревьев одного узла, разность высот не будет превышать 1.

В заключении проведем эксперимент и удалим из дерева узлы, что теоретически может привести к разбалансировке.

Добавим в конец функции **`main()`** файла «main.cpp», следующий код.

```
114
115         // Тестирование работы с деревом на упорядоченном наборе значений
116         int arr2[] = { 1, 1, 1, 2, 3, 4, 5, 6, 6, 7, 7, 7 };
117         // Формирование дерева:
118         create_tree(tree, arr2, sizeof arr2 / sizeof arr2[0]);
119         // Просмотр дерева по уровням
120         cout << "Просмотр дерева, построенного "
121              << "по упорядоченной последовательности\n";
122         show_tree(tree);
123         cout << "Удаление числа 4 (корень)\n";
124         remove_from_tree(tree, 4);
125         show_tree(tree);
126         cout << "Удаление числа 2 (промежуточный узел)\n";
127         remove_from_tree(tree, 2);
128         show_tree(tree);
129         cout << "Удаление числа 1 (лист)\n";
130         remove_from_tree(tree, 1);
```

```
131         show_tree(tree);
132         drop_tree(tree);
133         system("pause");
134 }
```

В результате его выполнения в конце вывода будут показаны следующие результаты.

```
Просмотр дерева, построенного по упорядоченной последовательности
4(*1) 2(*1) 6(*2) 1(*3) 3(*1) 5(*1) 7(*3)
Удаление числа 4 (корень)
3(*1) 2(*1) 6(*2) 1(*3) - 5(*1) 7(*3)
Удаление числа 2 (промежуточный узел)
3(*1) 1(*3) 6(*2) - - 5(*1) 7(*3)
Удаление числа 1 (лист)
5(*1) 3(*1) 6(*2) - - - 7(*3)
```

Нетрудно заметить, что удаление элементов не приводит к возникновению дисбаланса в дереве, что могло произойти после последней операции.

Бинарное дерево выражений

Бинарное дерево выражений *представляет собой особый вид двоичного дерева, используемого для представления алгебраических и логических выражений.* Эти деревья могут представлять выражения, содержащие как унарные, так и бинарные операторы.

Операнды выражения в двоичном дереве всегда являются листьями, при этом остальные узлы содержат операторы. Соответственно, бинарные операторы будут содержать указатели на двух потомков, а унарные – только на одного.

Как правило, дерево выражений строится по выражению, записанному в обратной польской нотации (в постфиксной форме).

Рассмотрим пример работы с деревом выражений.

Пусть имеется выражение:

$$(a + b^2)^3(c - d)^2.$$

Для построения дерева необходимо преобразовать это выражение в постфиксную форму. Это можно сделать при помощи стека (см. задачи для

самостоятельного решения по теме «Стек»). В результате преобразования получим запись вида:

$$a\ b\ 2\ \hat{}\ +\ 3\ \hat{}\ c\ d\ -\ 2\ \hat{}\ *.$$

Теперь можно приступить к построению дерева. Можно разбирать данную строку справа налево и пользоваться следующим алгоритмом.

Первый встреченный оператор добавляем в дерево в качестве корня, последующие операторы рекурсивно добавляем в правое поддерево до тех пор, пока не встретится операнд. В случае, если последний добавленный оператор является бинарным, добавляем символы в левое поддерево до тех пор, пока не встретится операнд. Как только в конце и левого и правого поддеревьев окажутся операнды (либо после унарного оператора заполненным окажется правое поддерево), поднимаемся на уровень вверх и продолжаем процесс дальше.

При применении данного алгоритма будет построено дерево, показанное на рис. 24.

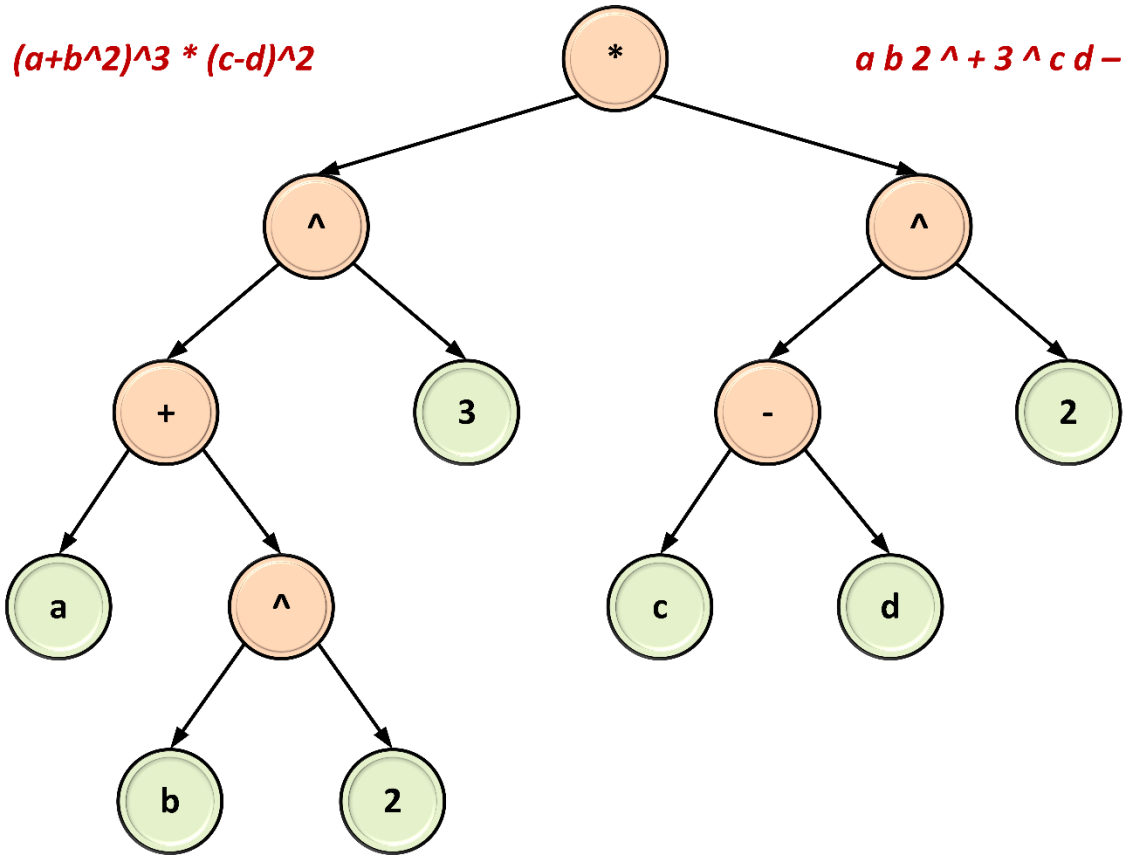


Рис. 24. Пример построения дерева выражений

Из уже готового дерева можно получить как постфиксную, так и инфиксную запись выражения. Так, для восстановления выражения в обратной польской нотации достаточно произвести постфиксный обход дерева вглубь.

Стандартная же, инфиксная, форма записи может быть получена центрированным обходом дерева вглубь с заключением в скобки как левого, так и правого поддерева.

Дерево общего вида

В некоторых практических задачах бывает удобно применять для хранения данных деревья общего вида. Например, в такую структуру организуется информация о файловой системе – список папок и файлов, расположенных на носителе данных.

Поскольку количество дочерних элементов для каждого узла в дереве общего вида не ограничено, возникает вопрос, каким образом хранить в каждом узле ссылки на дочерние элементы. В бинарных деревьях для этого было достаточно двух переменных (**`left`** и **`right`**). Для полных же деревьев вместо них потребуется использовать списки.

В следующем примере будет построено дерево, способное хранить список файлов и папок. При этом, чтобы не усложнять программный код очередным описанием уже рассмотренных ранее конструкций, в проекте будут использованы некоторые стандартные классы языка C++. Часть из них реализуют ряд динамических структур данных: **`list`** (для работы со списками), **`queue`** (для создания очередей), **`stack`** (для организации работы со стеком). Другие – **`string`** и **`stringstream`** используются соответственно для организации работы со строками и выполнения некоторых специальных операций с текстом. Эти классы находятся в стандартном пространстве имен, поэтому перед их именами в коде либо должно присутствовать уточнение **`std::`**, либо потребуется использовать конструкцию **`using namespace std;`**.

В состав разрабатываемого проекта войдут 3 файла: «main.cpp», содержащий функции конкретного приложения, а также «tree.h» и «tree.cpp» соответственно с заголовками и реализацией функций для формирования полного дерева.

Файл «tree.h».

```
1   #pragma once
2   #include <list>
3   #include <queue>
4   #include <string>
5   using namespace std;
6
7   /// <summary>
8   /// Структура для работы с узлами дерева
9   /// </summary>
10  struct node
11  {
12          /// <summary>
13          /// Название хранимого элемента (файла, папки или буквы диска)
14          /// Каждый узел предназначен для хранения короткого имени файла или папки
15          /// </summary>
16          std::string name = "";
17
18          /// <summary>Уровень, на котором расположен элемент файловой структуры</summary>
19          int level = -1;
20
21          ///<summary>Список дочерних элементов</summary>
22          list<node*> list;
23  };
24
25  /// <summary>Структура для работы с деревом в целом</summary>
26  struct tree
27  {
28          /// <summary>Указатель на корень дерева</summary>
29          node* root = new node;
30
31          /// <summary>
32          /// Инициализация корня значением обратного слеша (с этого символа
33          /// будет начинаться файловая структура устройства с несколькими дисками).
34          /// Уровень, на котором располагается корень файловой структуры - 0
35          /// </summary>
36          tree()
37          {
38                  root->name = "\\";
39                  root->level = 0;
40          }
41
42          /// <summary>Удаление корня при удалении переменной типа tree</summary>
43          ~tree()
44          {
45                  delete root;
46                  root = nullptr;
47          }
48  };
```

```
49
50  /// <summary>
51  /// Функция размещает объект с указанным полным именем в дереве.<br><br>
52  /// Например, если в качестве параметра full_name было передано значение
53  /// C:\\Windows\\System32\\driver.dll
54  /// и функция вызывается впервые, то в дерево будет добавлено 4 новых элемента :
55  /// C:, Windows, System32 и driver.dll <br>
56  /// При повторном вызове функции с параметром full_name равным, например,
57  /// C:\\Windows\\notepad.exe,
58  /// в дерево будет добавлен только один новый элемент :
59  /// notepad.exe
60  /// Два других(C: и Windows) в нем уже присутствуют.
61  /// </summary>
62  /// <param name="t">Дерево, к которое добавляется новый(ые) элементы</param>
63  /// <param name="full_name">Полный путь к элементу файловой структуры</param>
64  void add_to_tree(tree& t, string full_name);
65
66  /// <summary>
67  /// Удаление элемента из дерева
68  /// </summary>
69  /// <param name="t">Дерево, из которого нужно удалить узел</param>
70  /// <param name="full_name">Полный путь к элементу файловой структуры</param>
71  /// <returns>true, если удаление успешно и false в противном случае</returns>
72  bool delete_from_tree(tree& t, string full_name);
73
74  /// <summary>
75  /// Функция обхода дерева для внешнего вызова
76  /// </summary>
77  /// <param name="t">Дерево, элементы которого требуется получить</param>
78  /// <returns>Очередь из элементов дерева</returns>
79  queue<const node*> traverse(const tree& t);
80
81  /// <summary>
82  /// Удаление дерева.
83  /// </summary>
84  /// <param name="t">Ссылка на удаляемое дерево</param>
85  void drop_tree(tree& t);
```

Файл «tree.cpp».

```
1   #include "tree.h"
2   #include <sstream>
3
4   // ------------------------------------------------------------
5   // Список вспомогательных функций
6
7   /// <summary>
8   /// Вспомогательная рекурсивная функция добавления узла в дерево
9   /// </summary>
10  /// <param name="n">Узел в котором будет размещен новый элемент</param>
11  /// <param name="name">Короткое имя элемента файловой структуры</param>
12  void _add_to_tree(node*& n, string name);
13
14  // Удаление узла из дерева
15  bool _delete_from_tree(node*& n, string name);
16
17  /// <summary>
```

```
18  /// Внутренняя основная рекурсивная функция обхода дерева.
19  /// </summary>
20  /// <param name="q">Очередь из элементов дерева</param>
21  /// <param name="n">Узел, для которого выполняется обход</param>
22  void _traverse(queue<const node*>& q, const node* n);
23
24  /// <summary>
25  /// Встраиваемая вспомогательная функция создания нового узла дерева
26  /// </summary>
27  /// <param name="name">Короткое имя создаваемого узла</param>
28  /// <returns>Указатель на новый узел для размещения в дереве</returns>
29  inline node* _create_new_node(string name);
30
31  /// <summary>
32  /// Внутренняя функция удаления дерева
33  /// </summary>
34  /// <param name="t">Корневой узел удаляемого дерева или его части</param>
35  void _drop_tree(node*& t);
36
37  /// <summary>
38  /// Вспомогательная функция поиска указателя на элемент файловой структуры,
39  /// как дочернего элемента заданного узла
40  /// </summary>
41  /// <param name="parent">Родительский элемент, в котором выполняется поиск </param>
42  /// <param name="name">Короткое  название искомого узла</param>
43  /// <returns>Указатель на искомый узел дерева</returns>
44  node* _get_node_in_list_by_name(const node* parent, string name);
45
46  /// <summary>
47  /// Вспомогательная функция определения порядкового номера дочернего узла из списка,
48  /// соответствующего указанной вершине дерева t.
49  /// </summary>
50  /// <param name="t">Узел дерева, в котором выполняется поиск</param>
51  /// <param name="name">Короткое имя искомого узла</param>
52  /// <returns>Номер узла в списке дочерних элементов, либо -1, если узел не найден</returns>
53  int _get_node_list_index_by_name(const node* t, string name);
54
55  /// <summary>
56  /// Получение узла в родительском элементе по полному имени
57  /// </summary>
58  /// <param name="full_name">Полный путь к элементу файловой структуры</param>
59  /// <param name="parent">Родительский узел, в котором размещен искомый элемент</param>
60  /// <returns>Указатель на искомый узел файловой структуры или nullptr,
61  /// если таковой не найден</returns>
62  node* _get_node_by_full_name(string full_name, node*& parent);
63  //-----------------------------------------------------------------
64
65  void add_to_tree(tree& t, string full_name)
66  {
67          // Деление строки на части по символу "\"
68          // Для выполнения операции используется класс stringstream,
69          // сформированный по строке с полным именем файла
70          stringstream ss(full_name);
71          string item;
72          node* n = t.root;
73          // Бежим по всем элементам и либо находим в дереве
74          // существующий указатель,
75          // либо добавляем новый элемент в дерево
```

```
76          while (getline(ss, item, '\\'))
77          {
78                  // Функция getline на каждой итерации цикла
79                  // помещает часть строкового потока, размещенного между обратными слешами,
80                  // в переменную item, которая представляет собой короткое имя файла или папки
81                  if (!item.empty()) // если имя не пустое
82                          // Выполняется вызов вспомогательной (перегруженной) функции
83                          // add_to_tree с первым аргументом типа node*
84                          _add_to_tree(n, item);
85          }
86  }
87
88  void _add_to_tree(node*& n, string name)
89  {
90          // Если корень не пуст
91          if (!n) return;
92          // Получение родительского узла для нового
93          auto container = _get_node_in_list_by_name(n, name);
94          // Если родительский узел найден
95          if (!container)
96          { // Добавляем в него новый узел
97                  container = new node;
98                  container->name = name;
99                  container->level = n->level + 1;
100                 n->list.push_back(container);
101         }
102         n = container;
103 }
104
105 bool _delete_from_tree(tree& t, string full_name)
106 {
107         // Параметр full_name содержит полное имя удаляемого объекта
108         node* parent = t.root;
109         // Находим узел (n), соответствующий полному имени, а также его предка (parent)
110         node* n = _get_node_by_full_name(full_name, parent);
111         if (parent && n){
112                 // Если узел и его предок найдены,
113                 // удаляем из списка родительского элемента ссылку требуемый узел
114                 parent->list.remove(n);
115                 // Удаляем сам узел из дерева
116                 _drop_tree(n);
117                 return true;
118         }
119         return false;
120 }
121
122 queue<const node*> traverse(const tree& t)
123 {
124         queue<const node*> q;
125         const node* root = t.root;
126         _traverse(q, root);
127         return q;
128 }
129
130 void _traverse(queue<const node*>& q, const node* n)
131 {
132         if (n){ // Если уровень записи отличен от корневого
133                 q.push(n); //Добавляем узел в очередь
```

```
134             for (auto e : n->list)
135             {
136                     // Для всех дочерних узлов текущего узла выполняем
137                     // добавление их в очередь.
138                     _traverse(q, e);
139             }
140     }
141 }
142
143 void drop_tree(tree& t)
144 {
145     _drop_tree(t.root);
146 }
147
148 void drop_tree(node*& t)
149 {
150     // Выполнение рекурсивного удаления всех поддеревьев
151     // по списку дочерних узлов
152     for (auto el : t->list)
153     {
154         _drop_tree(el);
155     }
156     // Очистка списка дочерних узлов
157     t->list.clear();
158     // Удаление узлов, кроме корневого.
159     // (Корневой узел удаляется внутри самой структуры tree)
160     if (t->level > 0) {
161         delete t;
162         t = nullptr;
163     }
164 }
165
166 inline node* _create_new_node(string name)
167 {
168     node* n = new node;
169     n->name = name;
170     return n;
171 }
172
173 int _get_node_list_index_by_name(const node* t, string name)
174 {
175     int i = 0;
176     // Данный цикл позволяет пробежать по всем элементам списка
177     for (auto el : t->list)
178     {
179         // Если находим элемент с требуемым именем
180         if (el->name == name)
181             // возвращаем его порядковый номер
182             return i;
183         i++;
184     }
185     // Если цикл ни разу не прервался, значит файла с указанным
186     // именем в списке на обнаружено
187     return -1;
188 }
189
190 node* _get_node_in_list_by_name(const node* parent, string name)
191 {
```

```
192         if (parent)
193               for (auto n : parent->list)
194                     if (n && n->name==name) return n;
195         return nullptr;
196   }
197
198   node* _get_node_by_full_name(string full_name, node*& parent)
199   {
200         // Для удаления элемента из файловой структуры это имя нужно разделить
201         // на части по символу "\". См. аналогичный прием в функции _add_to_tree
202         stringstream ss(full_name);
203         string item;
204         node* n = parent;
205         while (getline(ss, item, '\\'))
206         {// Получаем очередную часть имени файла
207               // Запоминаем текущий узел дерева
208               parent = n;
209               // Пытаемся найти текущий элемент (item) из узла n
210               n = _get_node_in_list_by_name(n, item);
211               if (!n) return nullptr; //поиск не удался
212         }
213         return n;
214   }
```

Для демонстрации возможностей работы с небинарным деревом воспользуемся функциями из файла «main.cpp».

```
1   #include <iostream>
2   #include <stack>
3   #include <Windows.h>
4   #include "tree.h"
5
6   using namespace std;
7
8   //-------------------------------------------------------
9   //Вспомогательные функции
10
11  /// <summary>
12  /// Показ элементов дерева либо в виде древовидной структуры, либо списком
13  /// </summary>
14  /// <param name="t">Отображаемое на экране дерево</param>
15  /// <param name="as_tree">
16  /// Параметр определяет тип представления содержимого:
17  /// Если равен true - выводит элементы в виде дерева
18  /// Если равен false - выводит элементы файловой структуры списком
19  /// </param>
20  void show_tree(const tree& t, bool as_tree = true);
21
22  /// <summary>
23  /// Функция отображения элементов дерева в древовидной структуре
24  /// </summary>
25  /// <param name="q">Очередь из элементов выводимого дерева</param>
26  void _show_as_tree(const queue<const node*>& q);
27
28  /// <summary>
29  /// Функция отображения элементов дерева в форме списка
```

```
30  /// </summary>
31  /// <param name="q">Очередь из элементов выводимого дерева</param>
32  void _show_as_list(const queue<const node*>& q);
33
34  /// <summary>
35  /// Рекурсивная функция отображения полного пути к элементу файловой структуры
36  /// </summary>
37  /// <param name="s">Стек, содержащий элементы файловой структуры</param>
38  /// <param name="last_item">
39  /// Признак вывода последнего элемента пути. <br>
40  /// Данный параметр влияет на используемый символ разделитель между элементами
41  /// Если равен true, будет выведен конец строки ("\n"), в противном случае - символ "\"
42  /// </param>
43  void _show_stack(stack<const node*> s, bool last_item = true);
44
45  /// <summary>
46  /// Вспомогательная функция, используемая для формирования полных имен всех возможных
47  /// элементов файловой структуры, имеющихся в стеке
48  /// </summary>
49  /// <param name="st">Стек элементов файловой структуры</param>
50  /// <param name="cnt">
51  /// Количество выводимых элементов или
52  /// -1 для вывода всех возможных значений
53  /// </param>
54  void _show_all_records(stack<const node*>& st, int cnt = -1);
55  //-------------------------------------------------------------
56
57  /// <summary>
58  /// Основная функция программы
59  /// </summary>
60  void main()
61  {
62      setlocale(LC_ALL, "Rus");
63      SetConsoleCP(1251);
64      SetConsoleOutputCP(1251);
65      //Создание дерева папок и файлов
66      tree t;
67      //Добавление элементов структуры дерева
68      add_to_tree(t, "C:\\Windows\\etc\\drivers\\hosts");
69      add_to_tree(t, "C:\\Windows\\notepad.exe");
70      add_to_tree(t, "C:\\boot.data");
71      add_to_tree(t, "C:\\Program Files");
72      add_to_tree(t, "C:\\Windows\\System32\\system.dll");
73      add_to_tree(t, "D:\\Users\\User1");
74      add_to_tree(t, "D:\\Users\\User1\\Desktop");
75      add_to_tree(t, "D:\\Users\\User1\\Desktop\\Chrome.lnk");
76      //Отображение всех элементов файловой структуры в виде списка
77      show_tree(t, false);
78      //Отображение всех элементов файловой структуры в виде дерева
79      show_tree(t);
80
81      //Далее выполняются изменения структуры дерева и отображение результатов
82      delete_from_tree(t, "C:\\Windows\\etc\\drivers\\hosts");
83      show_tree(t);
84      delete_from_tree(t, "D:");
85      show_tree(t);
86      add_to_tree(t, "C:\\Users\\User1");
87      show_tree(t);
```

```
88      delete_from_tree(t, "C:\\Users");
89      show_tree(t);
90      delete_from_tree(t, "C:\\Windows");
91      show_tree(t);
92      drop_tree(t);
93      show_tree(t);
94      system("pause");
95  }
96
97  void show_tree(const tree& t, bool as_tree)
98  {
99      //Создание очереди из элементов файловой структуры
100     const queue<const node*> q = traverse(t);
101     //Выбор способа отображения элементов
102     if (as_tree)
103         _show_as_tree(q);
104     else
105         _show_as_list(q);
106     cout << endl;
107 }
108
109 void _show_as_tree(const queue<const node*>& queue)
110 {
111     //Формирование копии очереди
112     auto q = queue;
113     //Пока очередь содержит какие-либо элементы
114     while (q.size() > 0)
115     {
116         //Получение указателя на первый элемент в очереди
117         auto el = q.front();
118         //Удаление первого элемента из головы очереди
119         q.pop();
120         //Вывод соединительных линий в зависимости от числа уровней
121         auto level = el->level;
122         for (int i = 0; i < (level - 1) * 2; i++)
123         {
124             cout << (i % 2 == 0 ? "|" : " ");
125         }
126         //Вывод заключительной линии перед элементом файловой структуры
127         if (level > 0) cout << "|-";
128         //Вывод короткого названия элемента
129         cout << el->name << endl;
130     }
131 }
132
133 void _show_as_list(const queue<const node*>& queue)
134 {
135     // Формирование копии очереди
136     auto q = queue;
137     // Создание стека элементов файловой структуры
138     stack<const node*> st;
139     // Сохранение уровня последнего элемента
140     auto last_lvl = 0;
141     // Для всех элементов очереди
142     while (q.size() > 0)
143     {
144         // Получение первого элемента из очереди
145         auto el = q.front();
```

```
146             if (el->level > 0) { // Пропуск корня файловой структуры
147                   if (el->level <= last_lvl) {
148                   // Если следующий элемент файловой структуры оказался на том же
149                   // или более высоком уровне, что и предыдущий,
150                   // отображаем содержимое, которое накопилось в стеке
151                         _show_all_records(st, last_lvl - el->level);
152                   }
153                   //Добавляем очередной элемент в стек
154                   st.push(el);
155             }
156             //Запоминаем уровень последнего элемента
157             last_lvl = el->level;
158             //Извлекаем рассмотренный элемент из очереди
159             q.pop();
160       }
161       //Показываем элементы, накопившиеся в стеке, в конце обработки очереди
162       _show_all_records(st);
163 }
164
165 void _show_all_records(stack<const node*>& st, int cnt)
166 {
167       int i = 0;
168       // Выводим указанное количество или все элементы из стека
169       // Например, для стека, содержащего элементы:
170       // - notepad.exe
171       // - Windows
172       // - C:
173       // при параметре cnt=-1 будут выведены записи:
174       // C:\Windows\notepad.exe
175       // C:\Windows
176       // C:
177       while (st.size() > 0 && (i++ <= cnt || cnt < 0)) {
178             _show_stack(st);
179             st.pop();
180       }
181 }
182
183 void _show_stack(stack<const node*> s, bool last_item)
184 {
185       // Формирование полного пути по элементам, расположенным в стеке
186       string n = s.top()->name;
187       s.pop();
188       if (s.size()>0) _show_stack(s, false);
189       cout << n << (last_item ? "\n" : "\\");
190 }
```

В результате работы программы на экране будут выведены представления элементов файловой системы в виде списка элементов, а также в формате древовидной структуры.

После выполнения группы функций **`add_to_tree()`**, в строке 77 выводится список всех элементов файловой структуры.

```
C:\Windows\etc\drivers\hosts
C:\Windows\etc\drivers
C:\Windows\etc
C:\Windows\notepad.exe
C:\Windows\System32\system.dll
C:\Windows\System32
C:\Windows
C:\boot.data
C:\Program Files
C:
D:\Users\User1\Desktop\Chrome.lnk
D:\Users\User1\Desktop
D:\Users\User1
D:\Users
D:
```

Затем те же самые элементы выводятся в виде древовидной структуры в строке 79.

```
\
|-C:
| |-Windows
| | |-etc
| | | |-drivers
| | | | |-hosts
| | |-notepad.exe
| | |-System32
| | | |-system.dll
| |-boot.data
| |-Program Files
|-D:
| |-Users
| | |-User1
| | | |-Desktop
| | | | |-Chrome.lnk
```

При удалении файла из дерева (строка 82) указывается полный путь к нему, но удаления промежуточных узлов, как и ожидается, не происходит.

```
\
|-C:
| |-Windows
| | |-etc
| | | |-drivers
| | |-notepad.exe
| | |-System32
| | | |-system.dll
| |-boot.data
| |-Program Files
|-D:
| |-Users
| | |-User1
| | | |-Desktop
| | | | |-Chrome.lnk
```

В то же время, при удалении промежуточного элемента файловой системы (например, папки, содержащей другие папки и файлы), выполняется удаление как самого элемента, так и всех его дочерних узлов. Например, при извлечении диска D: (строка 84), происходит полное удаление из файловой системы всего его содержимого.

```
\
|-C:
| |-Windows
| | |-etc
| | | |-drivers
| | |-notepad.exe
| | |-System32
| | | |-system.dll
| |-boot.data
| |-Program Files
```

Контрольные вопросы и задания для самостоятельного выполнения

1. Можно ли список назвать разновидностью дерева? Ответ обоснуйте.
2. Чем отличаются, с точки зрения реализации алгоритма, префиксный, постфиксный и инфиксный обходы бинарного дерева?
3. Назовите примеры практических задач, которые могут быть эффективно решены с использованием деревьев.

---

4. По возрастающей последовательности, содержащей большое число значений, создать несбалансированное и сбалансированное деревья поиска. Оценить скорость создания деревьев. Оценить скорость выполнения поиска несуществующего элемента в дереве (так, чтобы для доступа к несуществующему элементу потребовалось максимальное для данного дерева число проверок).
5. Загрузить из файла информацию об успеваемости студентов за последнюю сессию (в файле указаны фамилии, инициалы и средний

балл). Сформировать дерево для поиска по фамилии. Вывести из дерева баллы всех студентов с фамилией на заданную букву.

6. Загрузить из файла информацию об успеваемости студентов за последнюю сессию (в файле указаны фамилии, инициалы и средний балл). Сформировать дерево для поиска по среднему баллу. Вывести из дерева информацию о студентах, со средним баллом больше заданного значения.
7. Создать дерево по имеющемуся математическому выражению, представленному в постфиксной форме записи. Восстановить по созданному дереву обратную польскую запись данного выражения, а также его инфиксную (стандартную, скобочную) форму записи.
8. По имеющемуся дереву выражений, содержащему только числа и знаки операций (см. задачу 7), вычислить значение этого выражения.
9. По имеющемуся дереву выражений, содержащему числа, буквенные константы и знаки операций (см. задачу 7), вычислить значение этого выражения, запросив у пользователя значения для констант, имеющихся в дереве.
10. Написать реализацию небинарного дерева для разбора текстового файла в формате JSON. Дерево должно позволять получать значение или список значений по заданному ключу.

    JSON – это текстовый формат обмена данными. JSON-текст представляет собой одну из двух структур:

    - набор пар **ключ:значение**, где ключом может быть только строка, а значением – любая форма;
    - упорядоченный набор значений.

    В качестве значений в JSON могут быть использованы:

    - записи – заключенное в фигурные скобки неупорядоченное множество пар ключ:значение, разделенных запятыми, заключённое в фигурные скобки;

- одномерные массивы – это заключенное в квадратные скобки упорядоченное (возможно пустое) множество значений (в том числе разнотипных), разделенных запятыми;
- числа (целые или вещественные);
- литералы true (логическое значение «истина»), false (логическое значение «ложь») и null;
- строки – упорядоченные множества из нуля или более символов юникода, заключённые в двойные кавычки.

Ниже представлен пример JSON-файла.

```
{
   "firstName": "Иван",
   "lastName": "Иванов",
   "address": {
       "streetAddress": "Кремлёвская, 18",
       "city": "Казань",
       "postalCode": 420000
   },
   "phone": [
       "+7 843 233-7109",
       "+7 927 111-0500"
   ]
}
```

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Белов, В. В. Алгоритмы и структуры данных: учебник / В. В. Белов, В. И. Чистякова. – Москва: КУРС : ИНФРА-М, 2020. – 240 с. – (Высшее образование: Бакалавриат). – ISBN 978-5-906818-25-6. – Текст : электронный. – URL: https://znanium.com/catalog/product/1057212 (дата обращения: 15.03.2021). – Режим доступа: по подписке.
2. Колдаев, В. Д. Основы алгоритмизации и программирования: учебное пособие / В.Д. Колдаев ; под ред. проф. Л.Г. Гагариной. – Москва : ИД «ФОРУМ» : ИНФРА-М, 2019. – 414 с. – (Среднее профессиональное образование). – ISBN 978-5-8199-0733-7. – Текст : электронный. – URL: https://znanium.com/catalog/product/980416 (дата обращения: 15.03.2021). – Режим доступа: по подписке.
3. Колдаев, В. Д. Структуры и алгоритмы обработки данных : учебное пособие / В. Д. Колдаев. - Москва : РИОР : ИНФРА-М, 2020. – 296 с. – (Высшее образование: Бакалавриат). – ISBN 978-5-369-01264-2. – Текст : электронный. – URL: https://znanium.com/catalog/product/1054007 (дата обращения: 25.02.2021). – Режим доступа: по подписке.
4. Царев, Р. Ю. Алгоритмы и структуры данных (CDIO): Учебник / Царев Р.Ю., Прокопенко А.В. – Краснояр.:СФУ, 2016. – 204 с.: ISBN 978-5-7638-3388-1. – Текст : электронный. – URL: https://znanium.com/catalog/product/967108 (дата обращения: 25.02.2021). – Режим доступа: по подписке.

Учебное пособие

Программные модели на основе динамических структур данных

Маклецов Сергей Владиславович