

Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Клоунов Юрий Сергеевич
Должность: Ректор
Дата подписания: 23.09.2026 10:01:15
Уникальный программный ключ:
3beb265d5d589e7ff4c954946f3ad99a1e70ac12

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«ТЮМЕНСКИЙ ИНДУСТРИАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра «Кибернетических систем»

ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания
по выполнению курсовой работы
дисциплины «Технологии разработки программного обеспечения»
для обучающихся направления подготовки
27.03.04 - Управление в технических системах всех форм обучения

Составитель ***И.О. Лозикова,***
старший преподаватель

Тюмень
ТИУ
2025

Технологии разработки программного обеспечения: Методические указания по выполнению курсовой работы дисциплины «Технологии разработки программного обеспечения» для обучающихся направления подготовки 27.03.04 - Управление в технических системах всех форм обучения всех форм обучения

Методические указания рассмотрены и рекомендованы к изданию на заседании кафедры «Кибернетических систем»
«__» ноября 2025 года, протокол №

Аннотация

Методические указания по выполнению курсовой работы дисциплины «Технологии разработки программного обеспечения» для обучающихся направления подготовки 27.03.04 - Управление в технических системах всех форм обучения. Для выполнения курсовой работы обучающиеся должны быть знакомы с математическими основами программирования, структурами данных, теорией и практикой объектно-ориентированного и многопоточного программирования.

Приведено содержание основных этапов выполнения курсовой работы, требования к решению, оформлению и защите курсовой работы, варианты курсовых заданий. Даны методические указания и рекомендации по выполнению работы.

Методические указания могут быть рекомендованы для выполнения самостоятельной и курсовой работы по дисциплинам «Технологии программирования», «Структуры данных в технических системах».

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1. ЦЕЛЬ И ЗАДАНИЯ.....	5
Задание вариантов 1-10	5
Задания вариантов 11-37	8
2. ОПИСАНИЕ СТРУКТУР ДАННЫХ	10
2.1 Бинарное дерево	10
2.2 Бинарное дерево поиска.....	18
2.3 Сортирующее дерево, пирамида.....	26
3. ТРЕБОВАНИЯ К ПРОГРАММЕ.....	30
4. ТИПОВАЯ СТРУКТУРА ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ.....	30
5. РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ РАЗДЕЛОВ КУРСОВОЙ РАБОТЫ	32
6. ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА	33
7. ПОРЯДОК ЗАЩИТЫ РАБОТЫ.....	33
Библиографический список литературы.....	35
Приложение 1 Описания информационных классов для вариантов 11-37.....	36
Приложение 2 Образец оформления титульного листа	40
Приложение 3 Пример описания интерфейса программы.....	41
Приложение 4 Пример диаграммы классов.....	42
Приложение 5 Пример описания классов.....	43
Приложение 6 Пример спецификации состава программы.....	44
Приложение 7 Пример описания модулей программы	45

ВВЕДЕНИЕ

Методические указания по выполнению курсовой работы дисциплины «Технологии разработки программного обеспечения» для обучающихся направления подготовки 27.03.04 - Управление в технических системах всех форм обучения.

Курсовая работа является частью программы изучения дисциплины, проводится согласно графику учебного процесса и имеет целью оценить:

- теоретические знания, практические умения и навыки в области объектно-ориентированного программирования, структур данных, технологий программирования;
- умение творчески решать поставленную задачу, применяя теоретические, практические знания и умения;
- умение выполнить самостоятельно и в срок заданную работу;
- умение самостоятельно работать со специальной литературой.

Для выполнения курсовой работы обучающиеся должны быть знакомы с математическими основами программирования, структурами данных, теорией и практикой объектно-ориентированного и многопоточного программирования.

В рамках выполнения курсовой работы необходимо разработать программу в соответствии с предложенным вариантом задания, подготовить пояснительную записку о проделанной работе, и защитить работу перед преподавателем.

График выполнения курсовой работы соответствует графику аттестации семестра. Оценка за курсовую работу выставляется в зачетную книжку.

1. ЦЕЛЬ И ЗАДАНИЯ

Цель работы: разработать программное обеспечение с графическим/консольным интерфейсом, применяя объектно-ориентированные методы и многопоточные технологии.

Задание вариантов 1-10

Разработать систему классов, состоящую из абстрактных базовых классов и соответствующих им производных классов. В абстрактном классе описать абстрактные (чисто виртуальные) методы. Эти функции определяются в производных классах. При необходимости в разрабатываемые классы добавляются дополнительные методы (все виды конструкторов, методы установки и вывода значения полей, операция присваивания, сравнения и т. п.) для обеспечения надлежащего функционирования системы классов.

Объекты классов организовать набором объектов базового класса. Набор объектов – это список, массив, множество, контейнерные классы STL и т.д. Программа-клиент выполняет с набором объектов следующие операции: создать объект, удалить объект, показать значение объекта и прочие дополнительные операции, определенные в иерархии классов согласно варианту. Обязательно должен быть использован виртуальный вызов.

Предусмотреть графический интерфейс в программе-клиенте системы классов, позволяющий продемонстрировать все методы объектов иерархии классов.

В таблицах 1 и 2 перечислены классы объектов и обязательные операции с ними. Таблица 3 содержит спецификации вариантов заданий.

Таблица 1

Перечень классов объектов структур данных

Класс	Описание
dataArray	Массив данных, абстрактный, контейнер, шаблон
ListPointer	Список данных на базе адресных указателей, абстрактный, контейнер, шаблон
ArrOfSize	Массив заданного размера, контейнер, шаблон
ArraySafe	Надежный массив, контейнер, шаблон Надёжный массив в отличие от статического массива может изменять свой размер во время выполнения программы
SListOnArr	Односвязная структура данных (список), на базе массива с индексными указателями, контейнер, шаблон
DListOnArr	Двусвязная структура данных, на базе массива с индексными указателями, контейнер, шаблон

DListPointer	Двусвязная структура данных, на базе адресных указателей, контейнер, шаблон
SListPointer	Односвязная структура данных, на базе массива с индексными указателями, контейнер, шаблон
RSListPointer	Кольцевая односвязная структура данных, на базе адресных указателей, контейнер, шаблон
RDListPointer	Кольцевая двусвязная структура данных, на базе адресных указателей, контейнер, шаблон

Таблица 2

Спецификация операций (методов)

Операция (метод)	Описание
BasicOperations	Базовые операции ⊕ опрос размера списка, ⊕ очистка списка, ⊕ проверка списка на пустоту, ⊕ опрос наличия элемента с заданным значением, ⊕ доступ к значению элемента с заданным номером в списке, ⊕ получение позиции в списке элемента с заданным значением, ⊕ включение нового элемента в позицию с заданным номером, ⊕ удаление элемента из позиции с заданным номером, ⊕ итератор для доступа к элементам списка с операциями: 1) установка на начало списка, 2) проверка конца списка, 3) доступ к значению текущего элемента, 4) переход к следующему элементу списка, 5) переход к предыдущему элементу списка (для списков на базе массива или двусвязных структур)
LSDSort()	алгоритм LSD-поразрядной сортировки
MSDSort()	алгоритм MSD-поразрядной сортировки
PyramidalSort()	алгоритм пирамидальной сортировки
ShellSort()	алгоритм сортировки Шелла

operator +(&, &)	конкатенация списков
value operator [](int pos)	доступ к значению элемента по номеру позиции в списке
bool operator ==(&)	сравнение списков на равенство
bool operator > (&)	сравнение списков
bool operator < (&)	сравнение списков

Таблица 3

Спецификации вариантов 1-10

Вари- ант	Базовые классы	Производные классы	Обязательная операция (метод)
1	DataArray	ArrOfSize	BasicOperations, LSDSort(), operator +(dataArray&, dataArray &), bool operator ==(dataArray &)
2	ListPointer	DListPointer	BasicOperations, value operator [](int pos) bool operator ==(ListPointer &) bool operator > (ListPointer &)
3	ListPointer	RDListPointer	BasicOperations, value operator [](int pos) bool operator ==(ListPointer &) bool operator < (ListPointer &)
4	DataArray	ArraySafe	BasicOperations, MSDSort(), bool operator ==(dataArray &) bool operator > (dataArray &)
5	ListPointer	ArrOfSize	BasicOperations, value operator [](int pos) operator + (ListPointer &, ListPointer &) bool operator ==(ListPointer &)
6	ListPointer	ArraySafe	BasicOperations, value operator [](int pos) operator + (ListPointer &, ListPointer &) bool operator >(ListPointer &)
7	DataArray	SListOnArr	BasicOperations, PyramidalSort(), operator +(dataArray&, dataArray &), bool operator < (dataArray&)
8	ListPointer	RSListPointer	BasicOperations, value operator [](int pos) operator + (ListPointer &, ListPointer &)

			bool operator == (ListPointer &)
9	ListPointer	SListPointer	BasicOperations, value operator [](int pos) operator + (ListPointer &, ListPointer &) bool operator >(ListPointer &)
10	DataArray	DListOnArr	BasicOperations, ShellSort(), bool operator == (DataArray &), bool operator < (DataArray&)

Задания вариантов 11-37

Создать шаблон класса структуры данных по варианту Таблицы 4. Реализовать в нем следующие методы:

- все виды конструкторов – по умолчанию, с параметрами, копирования;
- деструктор;
- добавление в конец и в начало списка; вставка элемента в список по правилу;
- удаление элемента в списке заданного или по правилу; очистка списка;
- присваивание нового значения элементу списка и/или перегрузка операции присваивания (=);
- поиск элемента в списке по ключу - перегрузка операции ([]) - и по значению информационного поля;
- итераторы класса, возвращают значение информационного поля структуры данных;
- просмотр и вывод значений указанного элемента структуры;
- вывод всех элементов списка в допустимом для структуры данных порядке;
- дополнительные операции, обеспечивающие необходимую функциональность шаблона как контейнерного класса.

Используя шаблон класса структуры данных Таблицы 4 как контейнер, разработать программу обработки контейнера информационных объектов Таблицы 5 и Приложения 1.

Программа-клиент выполняет с контейнером информационных объектов следующие операции: создать объект, удалить объект, показать значение объекта и операции, определенные для структуры данных по варианту.

Предусмотреть графический интерфейс, позволяющий продемонстрировать все методы объектов информационных классов. Все входные данные для информационных объектов должны храниться в бинарных файлах. Предусмотреть чтение и запись данных в эти файлы.

В Таблице 4 перечислены возможные шаблоны классов абстрактных структур данных. Специфика структур данных и их программирования представлена в главе 2.

В Таблице 5 представлены классы информационных объектов.

Таблица 4

Перечень шаблонов классов

Шаблон класса	Описание
BSTree	Бинарное дерево поиска
RBTTree	Красно-черное дерево
AVLTree	АВЛ-дерево
BTreeArr	Бинарное дерево на массиве
BTree	Бинарное дерево на связном списке
BinaryHeap	Пирамида, сортирующее дерево на массиве
QuPriority	Очередь приоритетов

Таблица 5

Перечень классов информационных объектов

Класс	Описание ¹
CMyRec	Прямоугольник
CAuto	Автомобиль
CCity	Город
CBook	Книга
CMan	Человек
CComputer	Компьютер
CAccount	Банковский Счет
CCylinder	Цилиндр
CCheck	Кассовый чек
CCall	Телефонный звонок
CVector	Вектор
CMail	Почтовый адрес

Таблица 6

¹ См. подробное описание класса в Приложении 1

Спецификации вариантов 11-37

варианты	BSTree	RBTree	AVLTree	BTreeArr	BTree	BinaryHeap	QuPriority
CMyRec				11		25	
CAuto		12					26
Ccity			13			30	
CBook		14			35		24
CMan	23			37	15		
CComputer					29	16	
CAccount			21				17
CCylinder				20		28	
CCheck	27		19				36
CCall	18				31		
CVector		22		32			
CMail	34						33

2. ОПИСАНИЕ СТРУКТУР ДАННЫХ

2.1 Бинарное дерево

Дерево – это иерархическая структура данных, где все элементы расположены по уровням и происходят из одного родительского элемента – корня дерева, он расположен на нулевом уровне. Каждый элемент нижнего уровня имеет ровно одного предка.

Основные определения

Элемент дерева называется *вершиной (узлом) дерева*, вершины дерева соединены направленными дугами - *ветвями дерева*. *Листья* дерева - вершины, в которые входит одна ветвь и не выходит ни одной ветви.

Свойства дерева:

- существует вершина, в которую не входит ни одной дуги (корень);
- в каждую вершину, кроме корня, входит одна дуга.

Вершины, в которые входят ветви, называются потомками, а сама вершина – предком. Уровень потомка на единицу больше уровня его предка. Корень дерева не имеет предка, а листья дерева не имеют потомков.

Количество уровней в дереве называют *глубиной (высотой) дерева*. Пустое дерево имеет глубину ноль, глубина дерева из одного корня – единица. Уровни нумеруют, начиная с 0, рис. 1.

Степенью (арностью) вершины в дереве называют количество дуг, выходящих из нее. Арность дерева определяется по максимальной арности (степени) вершины, входящей в дерево. В зависимости от степени вершины выделяют следующие разновидности деревьев:

- бинарные деревья – степень дерева не более двух, рис.2 (б);

– N-арные, произвольные деревья – степень дерева N, рис. 2 (а).

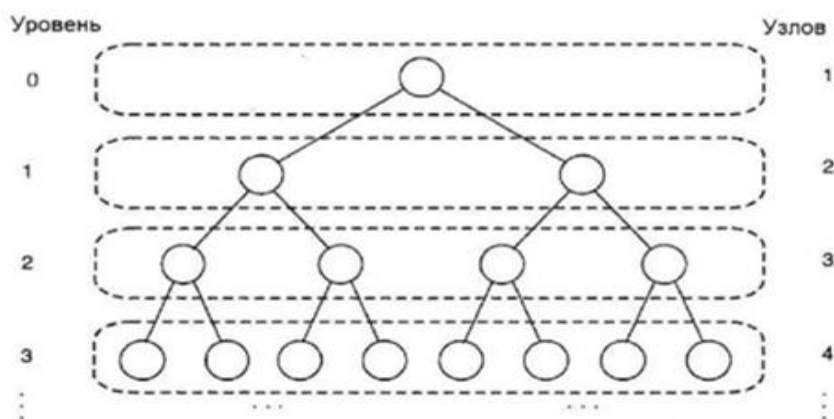


Рис.1 Бинарное дерево

Зная арность дерева (N) и номер уровня (l) можно рассчитать *максимальное число вершин (узлов)* в дереве на этом уровне по формуле:

$$T_l = N^l$$

Например, в бинарном дереве на 3-м уровне будете не более 8 вершин ($2^3 = 8$), рис. 1.

Поддерево – часть древообразной структуры данных, которая представлена в виде отдельного дерева. Следовательно, деревья - это рекурсивные структуры, где каждое поддерево является деревом. Операции с рекурсивными структурами реализуют рекурсивными алгоритмами. Если дерево определить как рекурсивную структуру, то каждый элемент является:

- 1) либо пустым деревом;
- 2) либо элементом, с которым связано конечное число поддеревьев.

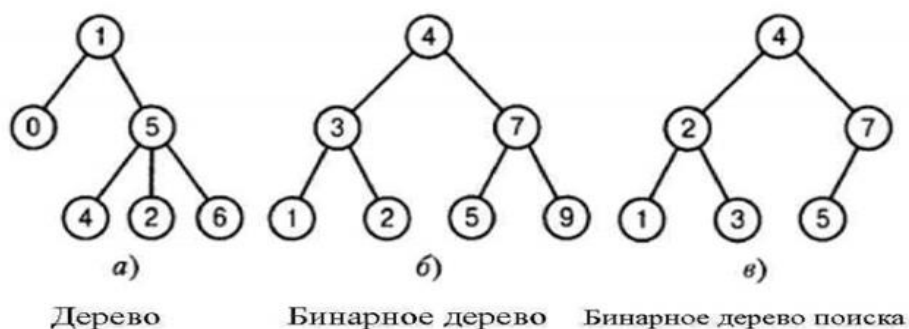


Рис.2 Разновидности деревьев

Операция обхода дерева – это упорядоченный доступ к каждой вершине дерева. Доступ к вершинам дерева осуществляется в определенном порядке. Существует три порядка обхода дерева (рис.3):

- прямой (preorder) – порядок обхода, при котором корень предшествует обоим поддеревьям;
- обратный (postorder) – порядок обхода, при котором корень следует за поддеревьями;

– симметричный (inorder) - порядок обхода, при котором сначала следует самая левая вершина, корень, правая вершина.

Все операции обхода являются рекурсивными алгоритмами.

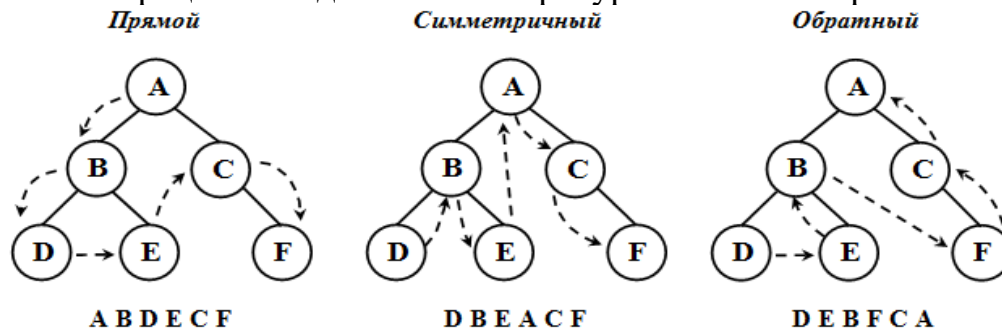


Рис. 3 Обходы деревьев

Для алгоритмизации используют простейшую разновидность дерева – бинарное. Если модель данных не соответствует бинарному дереву, то его определенными методами преобразуют из n-арного к бинарному дереву для обработки.

Бинарное (двоичное) дерево, В-дерево — иерархическая структура данных, в которой каждая вершина (узел, Node) имеет не более двух потомков (детей). Корень (root) дерева/поддерева называется родительским узлом, а потомки называются левым (left) и правым (right) наследниками (потомками), рис. 4.

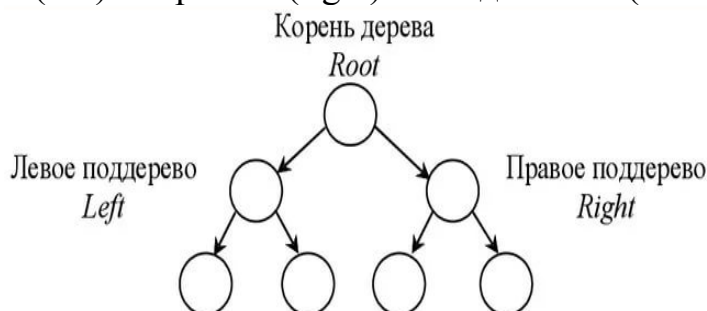


Рис. 4 Обозначения в бинарном дереве

Для обработки бинарные деревья различают, рис. 5:

- Полное и неполное бинарное дерево. Полное дерево содержит только полностью заполненные уровни, иначе оно является неполным.
- Строгое и нестрогое бинарное дерево. Строгое дерево имеет вершины степенью 0 (листья) или 2; нестрогое имеет вершины со степенью 0, 1 или 2.

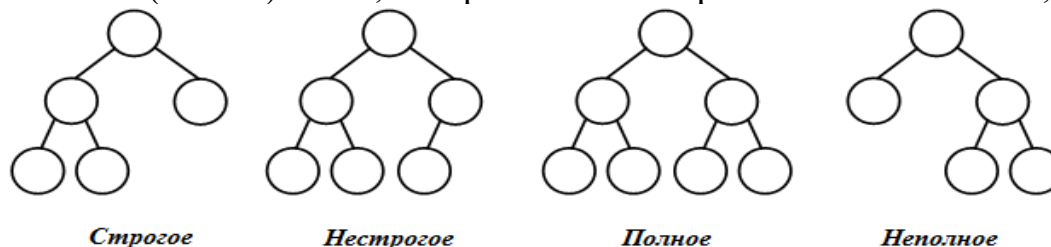


Рис.5 Разновидности бинарных деревьев

Основные операции с бинарными деревьями:

- создание бинарного дерева на базовой структуре данных – makeTree();

- обход бинарного дерева (прямой, симметричный, обратный) – preorderTree (), inorderTree (), postorderTree ();
- вставка элемента в бинарное дерево – insertTree();
- удаление элемента из бинарного дерева – removeTree();
- проверка пустоты бинарного дерева – isEmpty();
- удаление бинарного дерева – clear().

Бинарные деревья программируют на двух базовых структурах: массивах и связных списках. Связные списки для организации дерева нелинейные дву-связные или трехсвязные.

Реализация бинарного дерева на массиве

Лемма. Двоичное дерево массива длины M имеет глубину (число уровней) $\log_2 M$.

Используя эту зависимость, можно определить размер массива для хранения бинарного дерева.

Алгоритм *операции создания* makeTree() - заполнения массива значениями вершин - бинарного дерева:

1. В качестве поля структуры Tree для хранения дерева выступает одномерный массив $a[]$ фиксированной длины. Заполняем его значениями вершин бинарного дерева по правилу представленному на рис. 6.
2. Корень дерева хранится в элементе массива $a[0]$. Индекс корня $i = 0$.
3. Индекс левого потомка рассчитывается по формуле $2*i+1$, где i – индекс корня текущего поддерева.
4. Индекс правого потомка рассчитывается по формуле $2*i+2$, где i – индекс корня текущего поддерева.
5. Доступ к родительской вершине из текущей по правилу $(i-1) \div 2$, где i – номер текущей вершины.

Описываем *операции дерева в виде функций обработки одномерного массива $a[]$ или АТД Tree.* Удобно программировать доступ к левой, правой и родительской вершине через набор функций, представленный в Программе 2.15.

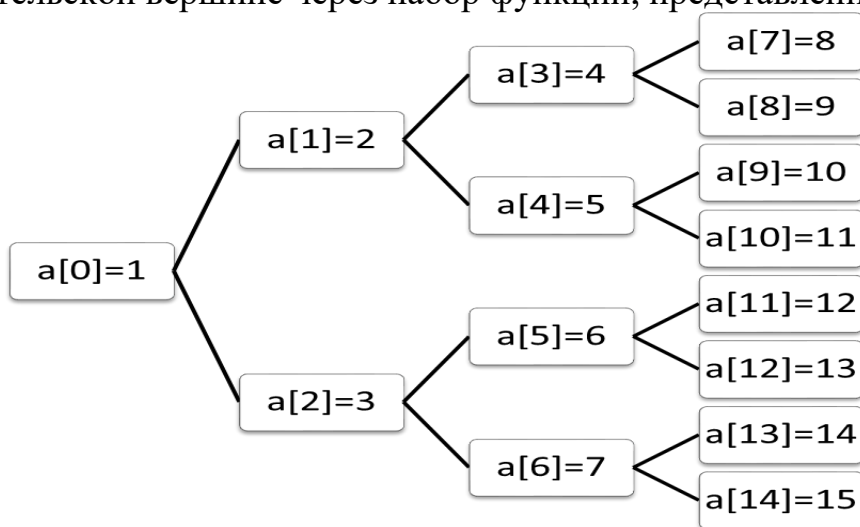


Рис. 6 Соответствие индексов массива и вершин бинарного дерева

Рассмотрим в качестве примера еще одно дерево на рис. 7. В массиве будут следующие значения `int A[] = {5, 1, 3, 9, 6, 2, 4, 7, 0, 8};`

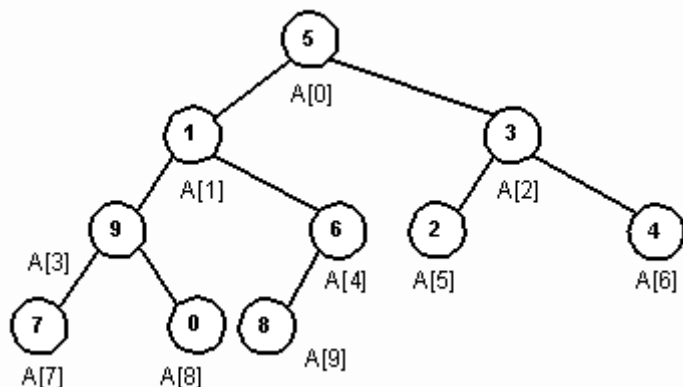


Рис. 7 Бинарное дерево на массиве

В случае неполного дерева в массиве будут пустые значения, поэтому нужно обязательно ввести обозначение признака пустого значения и по нему определять незначащие элементы массива. К недостаткам реализации дерева в массиве относятся:

- неполное дерево неэффективно использует память массива, много пустых элементов;
- размер дерева ограничен размером массива.

Реализация бинарного дерева на динамическом связном списке

Списочное представление бинарного дерева основано на описании узлов (Node) списка, соответствующих вершинам дерева, и указателя корня дерева `root` на начало списка. Узлы листовых вершин содержат в указателях левой и правой вершины `NULL`. Возможно два варианта описания узлов такого списка. Второй вариант описания структуры используют для работы с деревьями поиска.

Описание узлов и корня дерева:

1. Каждый узел имеет поле данных и два поля указателей (рис. 8): на левую вершину и правую.

```

struct Node {
    int d;
    Node* left;      //указатель на правого потомка
    Node* right;     //указатель на левого потомка
};
Node* root=NULL;   //указатель на дерево
  
```

2. Каждый узел имеет поле данных и три поля указателей (рис. 9): на левую и правую вершину, а также указатель на родительскую вершину. Корень дерева в указателе на родителя содержит `NULL`.

```

struct Node {
    int d;
    Node* father;   //указатель на родителя
    Node* left;
    Node* right;
  
```

```
};  
Node* root=NULL;
```

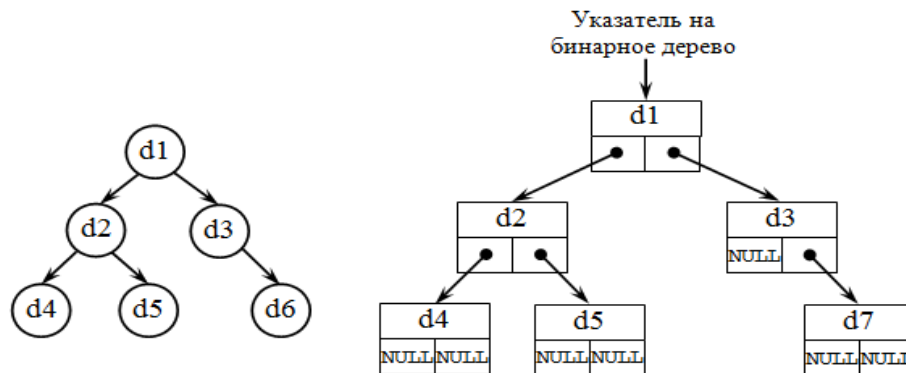


Рис. 8 Бинарное дерево на связном списке – 2 указателя

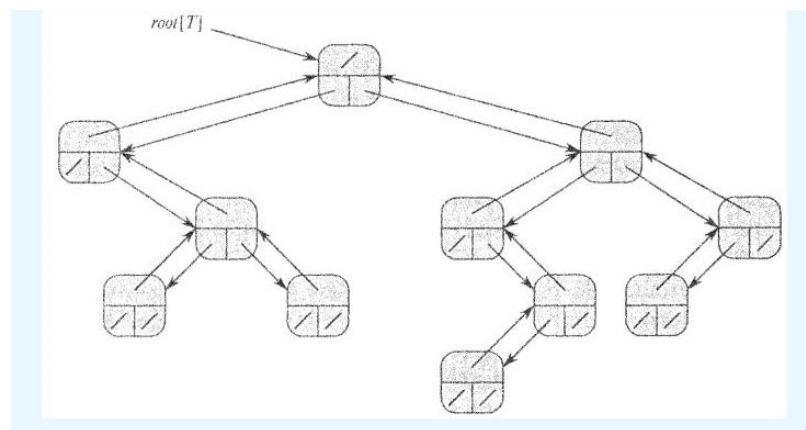


Рис. 9 Бинарное дерево поиска на связном списке – 3 указателя

Все операции дерева программируют на основе алгоритмов обработки связных списков.

Из одного и того же набора узлов можно сконструировать различные варианты двоичных деревьев. В простейшем случае дерево можно построить так, что его глубина будет равна числу элементов, и дерево выродится в односвязный список. На практике добавление и удаление элементов в дерево может происходить в произвольном порядке, что приводит к созданию различных деревьев, в том числе и вырожденных.

Операция создания бинарного дерева ставит задачу разработать полное дерево с N вершинами (узлами) и минимальной глубиной. Минимальная глубина позволит осуществить максимально быстрый доступ к каждой вершине. Это создание *сбалансированного бинарного дерева*.

Сбалансированное дерево – это дерево с максимально возможным числом вершин на каждом уровне, кроме нижнего уровня. В таком дереве для каждой вершины в левом и правом поддереве число вершин различается не более чем на 1.

Рекурсивный алгоритм построения сбалансированного бинарного дерева с N вершинами:

1. Пусть одна вершина будет корнем поддерева.
2. Создаем левое поддерево с $NL=N/2$ вершинами этим алгоритмом.
3. Создаем правое поддерево с $NR=N-NL-1$ вершинами этим алгоритмом.

Программа 2.1 Функция построения сбалансированного бинарного дерева с N вершинами на связном списке.

```
//-----  
Node* makeTree( Node* root, int N){  
    int NL, NR, val;  
    if (N==0) return NULL;  
    NL=N/2;  
    NR=N-NL-1;  
    Node* temp= new Node;  
    cout<<"Значение вершины:"<<endl; cin>>val;  
    temp->d=val;  
    temp->left=makeTree(root, NL );  
    temp->right=makeTree(root, NR);  
    return temp;  
}  
//-----  
  
...  
// первый вызов функции  
root = makeTree(root, 12);  
//-----
```

Псевдокод рекурсивной операции прямого обхода бинарного дерева, где p - указатель на корень поддерева:

```
void preorderTree (Node * p) {  
    if (p) {  
        // действия с узлом вершины  
        preorderTree (p->left);  
        preorderTree (p->right);  
    }  
}
```

Псевдокод рекурсивной операции обратного обхода бинарного дерева, где p - указатель на корень поддерева:

```
void postorderTree (Node * p) {  
    if (p) {  
        postorderTree (p->left);  
        postorderTree (p->right);  
        // действия с узлом вершины  
    }  
}
```

Псевдокод рекурсивной операции симметричного обхода бинарного дерева, где p - указатель на корень поддерева:

```

void inorderTree (Node * p) {
    if (p) {
        inorderTree (p->left);
        // действия с узлом вершины
        inorderTree (p->right);
    }
}

```

Рассмотрим функцию *операции прямого обхода* бинарного дерева для организации вывода на экран.

Программа 2.2 Функция вывода бинарного дерева на основе прямого обхода, где *p* – указатель корня поддерева, *l* – номер уровня. Вывод на экран дерева будет горизонтальный, количество отступов от начала строки – номер уровня *l*.

```

//-----
void preorderTree(Node * p, int l) {
    if (p) {
        for (int i = 0; i < l; i++) cout << ' '; //вывод отступов
        cout << p->d << endl; //вывод значения из вершины
        // рекурсивный вызов увеличивает номер уровня
        preorderTree(p->left, l + 1);
        preorderTree(p->right, l + 1);
    }
}

int main() {
    root = makeTree(root, 5);
    preorderTree(root, 0);
}
//-----

```

Для программирования произвольного дерева его преобразуют в бинарное дерево. Один из методов преобразования – использование представления дерева моделью “left child – right sibling representation” (LCRS, «левый ребенок – правый сосед»), рис. 10.

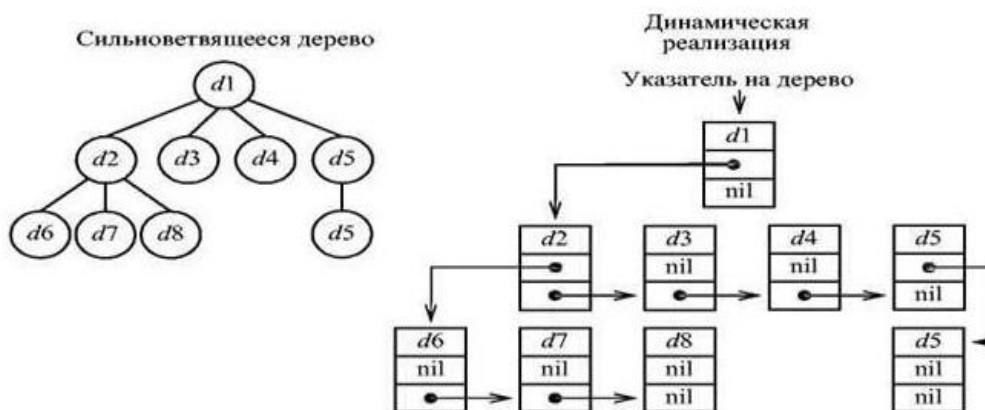


Рис. 10 Модель дерева “left child – right sibling representation”

Алгоритм преобразования дерева с произвольным ветвлением к бинарному дереву:

1. левый ребенок каждой вершины остается тем же;
2. правым ребенком текущей вершины становится её правый сосед (т.е. элемент, являющийся следующим ребенком одного и того же родителя).

На выходе будет дерево, где каждая вершина имеет не более двух потомков.

В программировании наибольшее применение имеют два вида двоичных деревьев: сбалансированное дерево поиска и сортирующее дерево (пирамида).

2.2 Бинарное дерево поиска

В основе представления дерева поиска лежат свойства упорядоченности и сбалансированности дерева. Назначение таких деревьев - эффективный поиск.

Свойство сбалансированности позволяет минимизировать время поиска и других операций в таком дереве. Время поиска по двоичному дереву зависит от глубины (высоты) дерева, и чем глубже (выше) дерево - тем больше времени требуется на поиск определенного узла в дереве и наоборот.

Упорядоченное дерево – это дерево, у которого ветви, исходящие из каждой вершины, упорядочены по определенному критерию.

Бинарное дерево поиска, BST - дерево (Binary Search Tree) – это двоичное дерево (рис.11), где:

- а. Каждая вершина/узел (Node) имеет не более двух потомков (child nodes);
- б. Каждая вершина/узел имеет ключ (key) и значение (value), где ключ соответствует критериям:
 - ключи всех узлов левого поддерева меньше значения ключа родительского узла;
 - ключи всех узлов правого поддерева больше или равны значению ключа родительского узла.

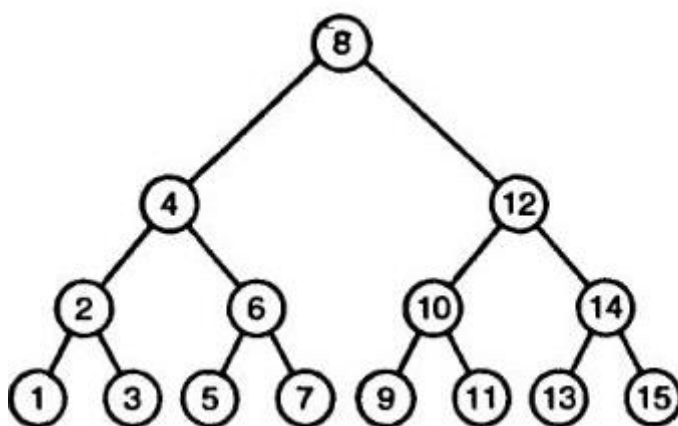


Рис. 11 Сбалансированное дерево поиска

В простейшем случае ключом будет значение вершины в узле.

Пусть x – произвольная вершина двоичного дерева поиска:

- если вершина y находится в левом поддереве вершины x , то $y < x$:

$$y \rightarrow d < x \rightarrow d$$

– если в правом, то

$$y \rightarrow d \geq x \rightarrow d.$$

Знак равенства можно ставить в любом условии.

Операции с бинарными деревьями поиска:

- добавление/вставка нового элемента - insertBST();
- обход бинарного дерева поиска (прямой, симметричный, обратный): preorderTree(), inorderTree(), postorderTree();
- поиск элемента в дереве – searchBST();
- поиск максимального, минимального элемента – minBST(), maxBST();
- поиск следующего и предыдущего по значению – succBST(), predBST();
- удаление элемента – deleteBST();
- удаление всего дерева – clear();
- проверка на пустоту – isEmpty().

Реализация этих операций может быть итеративной – циклом, и рекурсивной. На некоторых операциях будут показаны оба варианта реализации.

Описание узла дерева поиска:

```
struct Node {
    int d;
    Node* father;    //указатель на родителя
    Node* left;
    Node* right;
    Node(){father=right=left=NULL;}
};
Node* root=NULL;
```

Операция добавления/вставки элемента insertBST() находит для значения/ключа нового элемента подходящее место и вставляет его, сохраняя свойство упорядоченности.

Программа 2.3 Функция вставки нового значения в дерево поиска.

```
//-----
void insertBST(Node*&root, int val) {
    Node* z = new Node;    z->d = val;
    if (root == NULL) { root = z; return; }    //пустое дерево
    Node *y=NULL, *x = root;
    while (x) { //цикл поиска родительской вершины для вставки
        y = x;
        if (z->d < x->d) x = x->left;
        else x = x->right;
    }
    z->father = y;    // присвоили указатель на родителя
    if (y->d > z->d) // подцепляем к родителю слева или справа
        y->left = z;
    else y->right = z;
}
```

//-----

Операции обхода дерева поиска повторяют операции простого бинарного дерева. Но свойство упорядоченности позволяет обрабатывать все значения в неубывающем порядке в процессе симметричного обхода дерева. В этом случае мы обходим отсортированный список значений узлов.

Операция поиска `searchBST()` – основная целевая функция дерева поиска, рис.12 . Эту операцию рассмотрим в рекурсивной и итеративной реализации. Итеративная версия традиционно более быстрая.

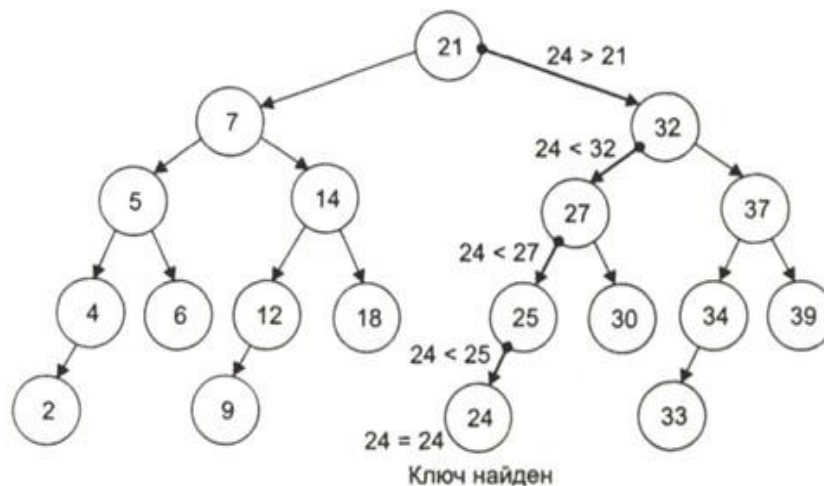


Рис. 12 Модель поиска значения 24 в BST - дереве

Программа 2.4 Функции поиска значения в дереве поиска, реализованные рекурсивно и итеративно, где `p` – корень дерева или корень поддерева, `key`- искомое значение .

//-----

```
Node * searchBST(Node * p, int key) {
    if (p == 0 || p->d == key) return p;
    if (key < p->d) return searchBST(p->left, key);
    else return searchBST(p->right, key);
}
```

//-----Итеративная версия (более эффективна – быстрая)-----

```
Node * iterative_searchBST(Node * p, int key) {
    while (p!=0 && key!=p->d)
        if (key < p->d) p =p->left;
        else p =p->right;
    return p;
}
```

//-----

Операции поиска максимального и минимального по значению элемента в дереве являются симметричными по своей реализации. Минимальное значение в дереве поиска можно найти, пройдя всю глубину дерева по указателям `left` до листьев. Максимальное значение найдем аналогично, но спускаться надо по указателям `right`.

Программа 2.5 Функция поиска минимального элемента в дереве поиска.

```
//-----
Node* minBST (Node* p){
    while (p->left != 0)
        p = p->left;
    return p;
}
```

//-----

Операция поиска следующего по значению за вершиной p требуют учесть в алгоритме два случая:

- если правое поддерево p – не пусто, то следующий за p элемент – минимальный в этом поддереве.
- если правое поддерево – пусто, тогда мы идём вверх от p , пока не найдём вершину, являющуюся правым сыном.

Операция поиска предыдущего – симметрична операции поиска следующего.

Программа 2.6 Функция поиска следующего значения относительно вершины p в дереве поиска.

```
//-----
Node * succBST (Node *p) {
    if (p->right) return minBST(p->right);
    Node *y = p->father;
    while (y != 0 && p == y->right) {
        p = y;
        y = y->father;
    }
    return y;
}
```

//-----

Алгоритм *операции удаление элемента* должен предусмотреть три случая расположения удаляемых элементов, см. рис. 13:

1. $p1$ – листовая (терминальная) вершина, ее удаляют, обнуляя указатель у ее предка;
2. $p2$ – корень неполного поддерева (один потомок), перенаправляют указатели предка, вырезая удаляемую вершину и удаляют её;
3. $p3$ – корень большого поддерева, идет поиск следующей по значению вершины, найденное значение переносят в корень поддерева, а ее удаляют.

Программа 2.7 Функция удаления элемента из дерева поиска, p – указатель на удаляемый элемент, y - фактически удаляемый элемент, x – указатель потомка удаляемого элемента.

```
//-----
void deleteBST(Node* &root, Node * p) {
Node * x = NULL, *y = NULL;
if (root && p) {
    if (p->left == 0 || p->right == 0) //фактически удаляемая
        y = p;
    else
        y = succBST(p);
}
```

```

    if (y->left) x = y->left;           //находим потомка
        else x = y->right;
    if (x) x->father = y->father;       //вырезаем вершину
    if (y->father == 0) root = x;       // если корень
    else {
        if (y == y->father->left)      // перенаправляем связи
            y->father->left = x;        // на потомка
        else y->father->right = x;
        if (y != p) p->d = y->d;        //переносим данные
        delete y;
    }
}
}
}
//-----

```

В алгоритме удаления сначала находим удаляемую вершину y – это либо явно вершина на входе, либо вершина, следующая по значению (*succBST*). Далее находим указатель на существующего потомка (x) вершины y . Вырезаем вершину y , меняя указатели. Рассматривается случай, когда y – корень дерева. Если случай $p3$, переносим данные.

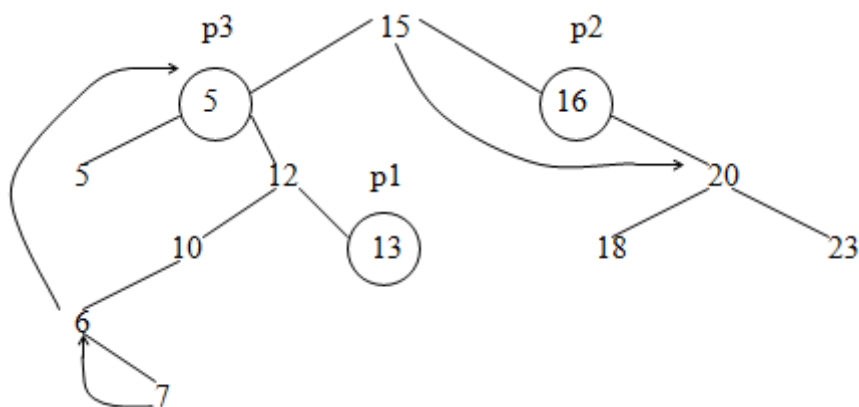


Рис. 13 Случаи удаления элементов в дереве поиска

Алгоритм *удаления всего дерева* использует обратный обход, так как при обратном обходе сначала посещаются потомки, а потом родительский узел. Это дает возможность удалить сначала нижние по уровню вершины, а затем корень. При этом для удаления очередной вершины используем операцию *delete*.

Дерево необходимо балансировать в процессе операций добавления/удаления, но требование идеальной сбалансированности делает этот процесс достаточно сложным, так как должны быть обработаны все вершины дерева.

Два основных типа сбалансированного дерева поиска, используемых в программировании: AVL-деревья² и красно-черные деревья.

AVL-дерево (АВЛ-дерево) – сбалансированное бинарное дерево поиска, где для каждого узла высота его двух поддеревьев различается не более чем на

² AVL, АВЛ – сокращение из первых букв фамилий создателей структуры – советских математиков Г.М. Адельсона-Вельского и Е.М. Ландиса.

1. Идеально сбалансированное бинарное дерево с N узлами имеет глубину (высоту) равную целому числу $\log_2(N+1)$ по основанию 2.

Каждый узел AVL-дерева определяется *показателем сбалансированности* – это разность между глубиной (высотой) правого и левого поддеревья данного узла. Пусть hR - глубина правого поддерева, hL - глубина левого поддерева. Дерево будет AVL-сбалансировано, если показатель каждого узла соответствует неравенству:

$$|hR - hL| \leq 1.$$



Рис. 57 Примеры AVL-деревьев и других

Показатель сбалансированности ($hR - hL$) в AVL-дереве может принимать одно из трех значений:

- -1, если левое поддерево на единицу выше правого;
- 0, если высоты обоих поддеревьев одинаковы;
- +1, если правое поддерево на единицу выше левого.

Если хотя бы один узел имеет другое значение показателя, то дерево не является AVL-сбалансированным. Примеры AVL-сбалансированных и AVL-несбалансированных деревьев представлены на рис. 14.

У листовых узлов показатель всегда равен 0. У остальных узлов вычисляется по формуле ($hR - hL$). Показатель сбалансированности (balance) будет полем в описание структуры узла. Указатели на левую и правую вершину представлены в виде массива указателей. Если необходимо, можно добавить в массив еще один указатель на родительский узел. Все операции добавления/удаления узлов сопровождаются пересчетом поля показателя сбалансированности.

Описание узла AVL-дерева:

```
struct NodeAVL{
    int data;
    //массив указателей на левое и правое поддерево
    NodeAVL * link[2];
    int balance;           //показатель сбалансированности
```

```
};  
NodeAVL* root=NULL;
```

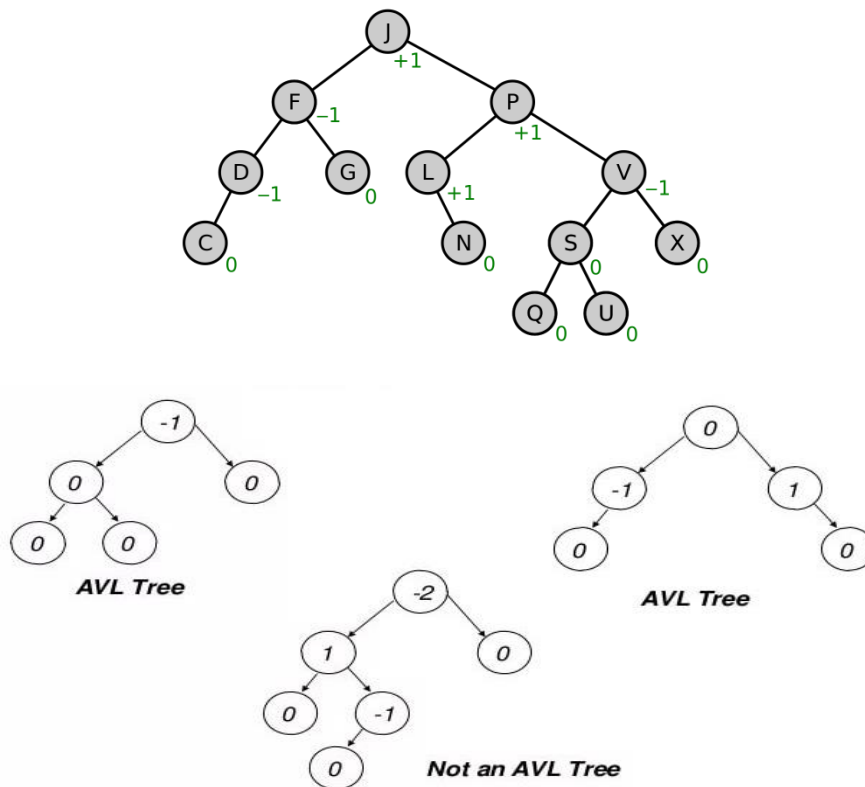


Рис. 14 Примеры AVL-сбалансированных и AVL-несбалансированных деревьев

Алгоритм добавления нового узла в AVL-дерево:

1. поиск позиции в дереве, где должен находиться новый узел;
2. добавить узел на найденную позицию;
3. пересчитать показатели сбалансированности для узлов, находящихся выше добавленного узла;
4. если дерево стало несбалансированным, то выполнить балансировку.

Балансировка выполняется алгоритмами, называемых поворотами узлов. Описание этих алгоритмов вне данного учебного пособия.

Красно-черное дерево (Red-Black-Tree, RB-Tree) – это бинарное дерево со следующими свойствами, рис.15:

- а. каждая вершина должна быть окрашена либо в черный, либо в красный цвет;
- б. корень дерева должен быть черным;
- с. листья дерева должны быть черными и объявляться как NULL-вершины (наследники узлов, которые обычно называют листьями; на них "указывают" NULL указатели);
- д. каждый красный узел должен иметь черного предка;
- е. на всех ветвях дерева, ведущих от его корня к листьям, число черных вершин одинаково.

Характеристика красно-черных деревьев:

- Красно-черные деревья являются методом балансировки деревьев, что определяется свойствами данной структуры.
- Красно-черные деревья обрабатываются теми же операциями, что и бинарные деревья.
- Операции добавления/удаления элемента требуют балансировки дерева.

Описание узла RB-дерева:

```
struct NodeRB{
    int data;
    //массив указателей на левое и правое поддереву
    NodeRB * link[2];
    char red;          //показатель сбалансированности
};
NodeRB* root=NULL;
```

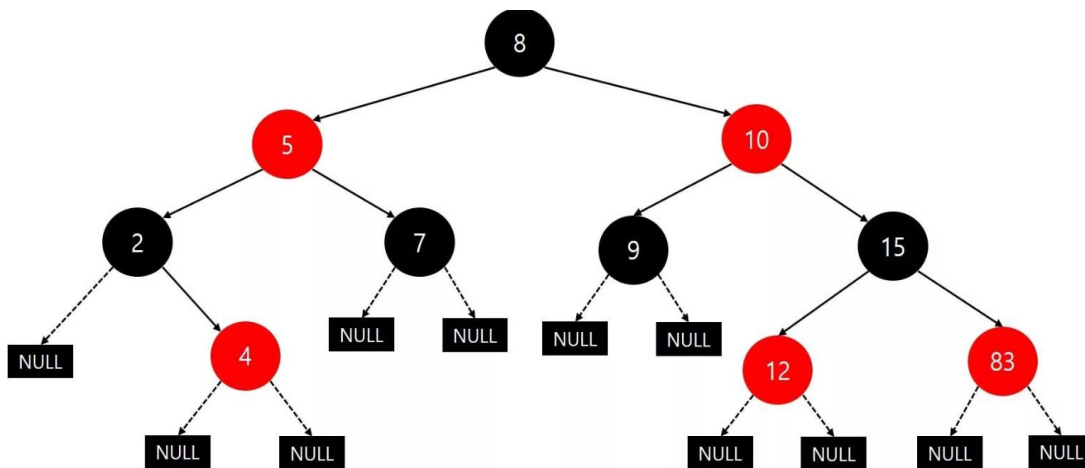


Рис. 15 Красно-черное дерево

Сравнительная характеристика методов балансировки AVL-дерева и красно-черных деревьев:

- эффективность алгоритмов вставки/удаления одинаковая, вычисляется в как двоичный логарифм от общего числа вершин в дереве;
- если число вершин дерева одинаково, максимальная глубина AVL- дерева всегда будет меньше, чем максимальная глубина красно-черного дерева, поэтому для поиска по красно-черному дереву требуется большее число операций, что сказывается на быстродействии алгоритмов;
- если число вершин дерева одинаково, хранение AVL-дерева требует больше памяти, так как каждый узел должен хранить глубину как целое число, а в узле красно-черного дерева хранится только один байт (можно ограничиться битом!), определяющий цвет узла.

2.3 Сортирующее дерево, пирамида

Пирамида, двоичная куча (binary heap) - это упорядоченное двоичное дерево, у которого минимальный/максимальный элемент расположен в корне. Для дерева обязательны три свойства:

- каждая вершина по значению не больше/не меньше, чем значения ее потомков.
- разница между глубиной листьев не более 1 уровня.
- нижний уровень формируется слева направо, стремясь к созданию полного дерева.

Пирамиду, у которой в корне расположен минимальный элемент, называют минимальной или неубывающая (min-heap), а где в корне максимальный элемент – максимальной или невозрастающая (max-heap) (рис.16).

Основная цель построения такого дерева – упорядочить, отсортировать исходную неупорядоченную последовательность данных путем создания пирамиды. Алгоритмы обработки пирамиды должны сами обновлять дерево и поддерживать упорядоченность вершин. Реализуют алгоритмы пирамиды на базе одномерного массива.

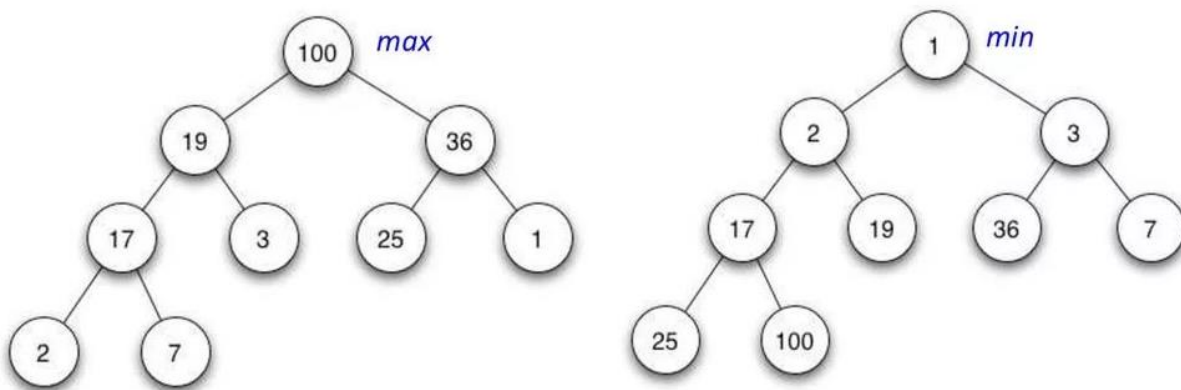


Рис. 16 Максимальная и минимальная пирамида

Основные операции с пирамидой:

- построить пирамиду – buildHeap(), преобразовать неупорядоченный массив в пирамиду;
- добавить элемент в пирамиду – pushHeap();
- удалить корневой (минимальный/максимальный) элемент – popHeap();
- упорядочивание пирамиды – heapify();
- показать корневой элемент – peekHeap();
- пирамидальная сортировка – sortHeap().

Алгоритм построения хранит пирамиду в одномерном массиве по следующим правилам:

- корень имеет индекс 0;
- потомки (left, right) i -го корня поддерева имеют индексы $2*i+1$, $2*i+2$;
- родительская вершина (father) i -го элемента $(i-1)/2$.

Пример представлен на рис. 17.

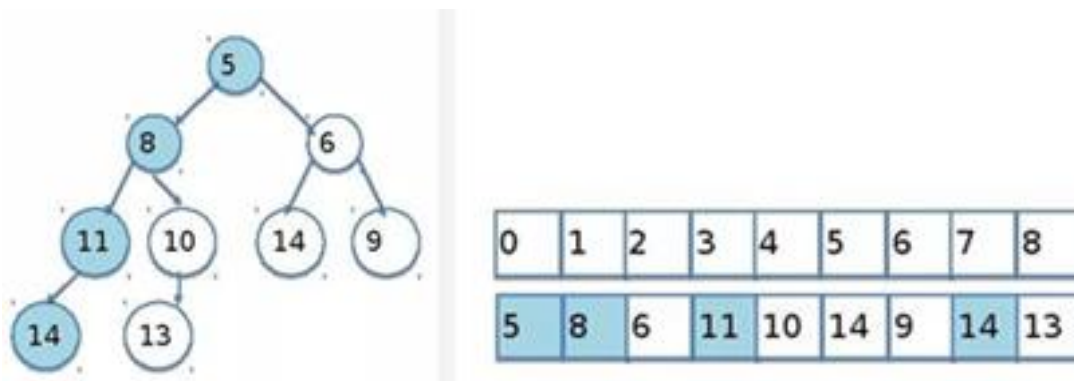


Рис. 17 Представление пирамиды в массиве

Программа 2.8 Функции расчета индексов связанных вершин пирамиды в массиве по индексу i текущей вершины.

```

//-----
int father(int i) {
    return (i-1) / 2;
}
int left(int i){
    return 2*i + 1;
}
int right(int i){
    return 2*i + 2;
}
//-----

```

Алгоритм преобразования исходного неупорядоченного массива размером N в пирамиду основан на перестановке элементов массива по *правилам алгоритма построения* минимальной/максимальной пирамиды. Последний элемент пирамиды – лист – имеет индекс $(N-1)$, следовательно, его предок расположен по индексу $(N-2)/2$. Рассмотрим данный алгоритм на примере.

Пусть имеем массив `int H[10]={9, 12, 17, 30, 50, 20, 60, 65, 4, 19}`; Пирамида будет иметь три уровня ($\log N$) – 0, 1, 2. Преобразуем этот массив в минимальную пирамиду. Рассчитаем индексы листовых вершин (5, 6, 7, 8, 9). Индексы корневых вершин соответственно (4, 3, 2, 1, 0).

1. Уровень 2: корень поддерева $H[4]=50 > H[9]=19$, поменяем их местами;
2. Уровень 2: корень поддерева $H[3]=30 > H[8]=4$, поменяем их местами (если потомка два {4, 65}, меняем местами с наименьшим из них);

Результат: `H[10]={9, 12, 17, 4, 19, 20, 60, 65, 30, 50}`;

3. Уровень 1: Корень поддерева $H[2]=17$ и он меньше чем $H[5]=20$ и $H[6]=60$, все оставим на месте;
4. Уровень 1: Корень поддерева $H[1]=12 > H[3]=4$, поменяем их местами (здесь два потомка {4,19});

Результат: `H[10]={9, 4, 17, 12, 19, 20, 60, 65, 30, 50}`;

5. Уровень 0: Корень пирамиды $H[0]=9 > H[1]=4$, поменяем их местами. Процесс закончился, так как достигли корня пирамиды.

Результат: $H[10]=\{4, 9, 17, 12, 19, 20, 60, 65, 30, 50\}$;

Алгоритм добавления элемента в минимальную/максимальную пирамиду,
рис 18:

1. Элемент добавляется в конец массива как лист пирамиды.
2. Сравнивается с родительской вершиной, если он меньше/больше, то меняется местами с ней.
3. В свою очередь измененный элемент сравнивается со своим родителем на условие пирамидальности, в случае необходимости меняется местами. Процесс завершается, если обмена (swap) не было.

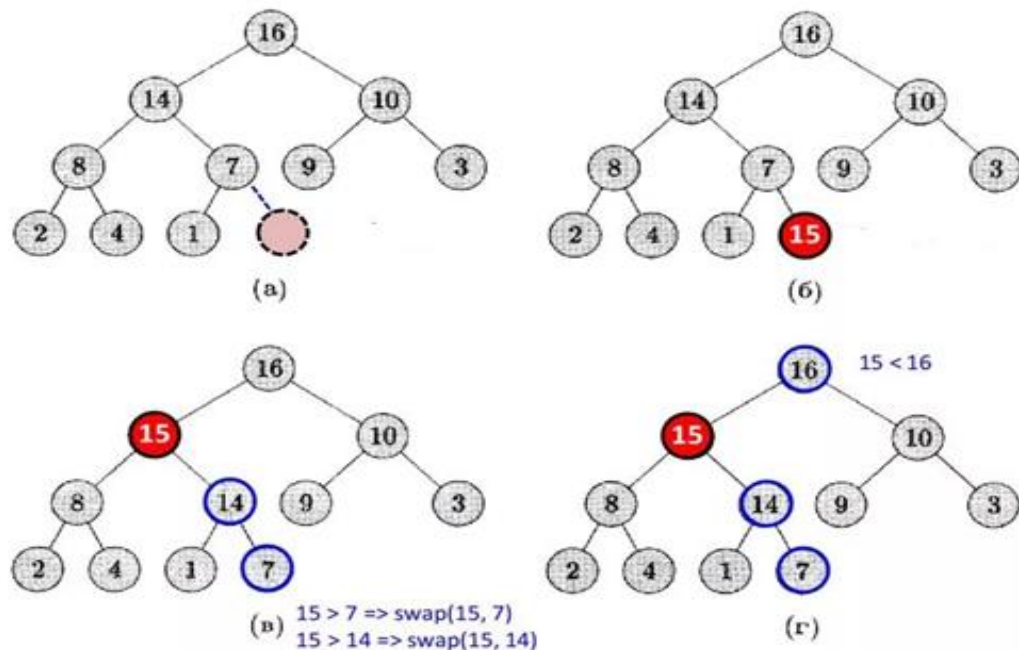


Рис. 18 Добавление элемента в пирамиду.

Алгоритм добавления можно представить как обратный проход по пирамиде снизу вверх – от листа до корня с целью поиска места нового элемента.

Алгоритм удаления элемента из корня минимальной пирамиды – это прямой проход по пирамиде от корня до листа для поиска нового корня пирамиды.

1. Удалить корневой элемент пирамиды и заменить его последним листом.
2. Если новый корень больше своих потомков, меняем местами с наименьшим потомком
3. Продолжаем сравнения с элементами потомков, до первого невыполнения условия и отсутствия обмена.

Алгоритм упорядочивания является базовым для всех алгоритмов пирамиды, так как его задача осуществить поиск места для i -го элемента в пирамиде. Имеет рекурсивную реализацию.

Программа 2.16 Функция упорядочивания пирамиды, где H – массив пирамиды, N – размер массива, i – индекс вершины, для которой идет поиск места в пирамиде.

```
//-----
void heapify(int H[], int i, int N) {
```

```

int temp;
int left = 2*i+1;
int right = 2*i+2;
if(left < N) {
    if(H[i] < H[left]) {
        temp = H[i];
        H[i] = H[left];
        H[left] = temp;
        heapify(H, left, N);
    }
}
if(right < N) {
    if(H[i] < H[right]) {
        temp = H[i];
        H[i] = H[right];
        H[right] = temp;
        heapify(H, right, N);
    }
}
}
//-----

```

Алгоритм пирамидальной сортировки – это прямой обход пирамиды с последовательным удалением элементов.

1. Преобразовать массив в пирамиду - buildHeap();
2. Последовательно удалять корневой элемент пирамиды и упорядочивать пирамиду.

Порядок сортировке задает вид пирамиды минимальный - сортировка по возрастанию или максимальный – сортировка по убыванию.

На базе пирамиды строят еще одну абстрактную структуру данных – очередь приоритетов.

Очередь приоритетов - очередь, в которой все элементы имеют приоритет (key= Priority), где две обязательные операции — добавить элемент и извлечь в соответствии с максимумом (минимумом) приоритета.

Основные операции очереди приоритетов:

- добавить элемент в очередь – Enqueue(key, value);
- удалить из очереди элемент с минимальным /максимальным ключом – DeleteMin(), DeleteMax();
- показать элемент с минимальным/максимальным ключом –Min(), Max();
- изменить значение ключа элемента и упорядочить очередь –DecreaseKey().

Используя алгоритмы пирамиды, легко реализовать операции очереди приоритетов.

3. ТРЕБОВАНИЯ К ПРОГРАММЕ

Описание и программирование системы классов реализует весь указанный в варианте задания функционал в виде методов классов. Набор дополнительных функций для эффективной работы программы необходимо определить самостоятельно. Программа-клиент предоставляет интерфейс для тестирования всех методов классов согласно варианту.

Применять для разработки языки программирования C++ или C#, по согласованию с преподавателем.

В программе использовать графический пользовательский интерфейс, предусмотреть ввод и вывод исходных данных и полученных результатов с необходимыми поясняющими текстами, иметь необходимые элементы управления, меню, подсказки. Предусмотреть контроль ввода данных.

Тестирование шаблонных классов должно быть выполнено на разных типах данных и классе информационного объекта.

В программе рекомендуется предусмотреть возможность сохранения данных в файл и их чтение из файла. Это может быть как запись в файл по команде пользователя, так и автоматическое сохранение/чтение данных при завершении/запуске программы.

Во время защиты курсовой работы выполнение программы должно быть продемонстрировано на контрольном примере и/или подготовленных тестах.

4. ТИПОВАЯ СТРУКТУРА ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ

Пояснительная записка состоит из глав и разделов:

Оглавление

Введение

1 Постановка задачи

- 1.1 Описание предметной области задачи
- 1.2 Функциональное назначение программы
- 1.3 Описание входных и выходных данных
- 1.4 Характеристики программы

2 Разработка программы

- 2.1 Диаграмма иерархии классов и ее описание
- 2.2 Описание классов
- 2.3 Описание алгоритмов и методов решения
- 2.4 Интерфейс программы

3 Описание программы

- 3.1 Общие сведения
- 3.2 Описание модулей программы
- 3.3 Текст программы
- 3.4 Спецификация состава программы
- 3.5 Тестирование программы

Источники, использованные при разработке

Пояснения к главам и разделам

Введение

Цель работы и задачи, которые потребуется решить для достижения цели, указать номера вариантов и текст задания.

Постановка задачи

Решение задачи начинается с ее постановки. Дается точное описание исходных данных, условий задачи и результатов ее решения.

Описание предметной области задачи

Текст задание и пояснения. Описываются возможные пути решения задачи с указанием их достоинств и недостатков. Выбирается и обосновывается метод решения задачи.

Функциональное назначение

Должны быть указаны требуемые функции разрабатываемой программы и сведения об ограничениях на применение.

Описание входных и выходных данных

Выходные данные, результаты. Описание выходных данных и их форматы.

Входные данные, исходные данные. Организация и предварительная подготовка входных данных, формат и смысловое описание.

Обязательно описываются ограничения, накладываемые на входные/выходные данные, необходимая разрядность и точность представления исходных данных и результатов решения. Указываются возможные пределы изменения входных/выходных параметров задачи.

В этом разделе могут быть описаны типы данных и основные приемы их программирования. Например, если в задаче используются динамические структуры, то перечисляются виды динамических структур данных и основные функции по работе с динамическими структурами. Если задача заключается в формировании массива и дальнейшей работе с массивом, то приводится описание типовых алгоритмов обработки массива. Если применяются файлы - приемы работы с файлами и т.д.

Характеристики программы

В этом разделе описывают нефункциональные и эксплуатационные требования к операционной системе, среде разработки, технологиям, методам и языкам разработки, требования к интерфейсам и др. Может быть описан концептуальный подход к методам решения.

Описание классов

Пример описания классов см. в Приложении 3.

Описание алгоритмов и методов решения

Представление алгоритмов решения в методах классов. Излагаются основные требования к алгоритму и пути их реализации. Приводится схема или псевдокод алгоритма, дается пояснение.

Интерфейс программы

Должен быть представлен проект Windows-окна (или окон), на котором расположены все визуальные компоненты разрабатываемого интерфейса. Компоненты должны быть пронумерованы. Ниже приводится расшифровка компонентов: название(тип) и имя; назначение в программе; события, на которые данный компонент откликается. Для каждого компонента должны быть указаны свойства, изменяемые при проектировании окна (Приложение 2).

Описание программы

Общие сведения. Должны быть указаны наименование программы, идентификатор исполняемого файла, программное обеспечение, необходимое для функционирования программы, язык программирования, методология и технология написания программы.

Описание модулей программы

Пример описания модулей программы см. в Приложении 5.

Текст программы

В разделе представлен исходный текст программы или ссылка на Приложение к Пояснительной записке. Перед заголовками функций, процедур, модулей должны быть комментарии, описывающие назначение подпрограммы/модуля, назначение входных и выходных параметров.

Спецификация состава программы

Пример описания спецификации состава программы см. в Приложении 4.

Тестирование программы

Кратко описывается среда программирования, перечисляются и описываются средства отладки в данной среде. Перечисляются требования, подлежащие проверке при испытании программы. Приводятся тесты или контрольный пример и ожидаемые результаты.

Прилагаются сканы решения контрольного примера и результатов тестирования.

Источники, использованные при разработке

В разделе перечисляется литература и другие информационные ресурсы, использованные при выполнении курсовой работы

5. РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ РАЗДЕЛОВ КУРСОВОЙ РАБОТЫ

Для решения поставленной задачи необходимо предварительно ознакомиться с литературой и другими источниками, посвященных теме задания. При этом следует обратить внимание на средства, используемые для решения аналогичных задач или для решения каких-либо ключевых моментов задачи. Этап работы с литературой и другими источниками, должен закончиться описанием предметной области задачи, где будут собраны полученные сведения из литературы, дан их анализ с точки зрения приложения к поставленной задаче.

Лаконичная спецификация вариантов задания на курсовую работу предоставляют обучающемуся простор для творчества. В вариантах заданий

умышленно опущены некоторые детали и необходимые требования. После ознакомления с литературой обучающейся должен оценить возможности языка программирования и вычислительной техники, на которой предлагается реализовать решение. Результатом этой работы должна быть точная формулировка задачи со всеми ограничениями и требованиями.

При решении задач необходимо придерживаться техники пошаговой детализации, использовать стандартные структуры, не забывая при этом о развитии программного окружения программиста, расширяя возможности языка за счет включения новых процедур и функций.

При разработке алгоритма необходимо предусмотреть средства проверки и тестирования программы, удобство работы пользователя, возможные модификации.

При написании программы не следует забывать о хорошем стиле программирования, о таких понятиях, как читабельность, эффективность, надежность. Необходимо искать наиболее простые и естественные приемы и методы решения.

6. ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Текст отчета должен быть напечатан на одной стороне стандартного листа белой бумаги формата 210 x 297 мм (A4) в редакторе MS Word. Шрифт – Times New Roman, размер – 14 pt, междустрочный интервал одинарный, выравнивание по ширине. Размеры полей: слева и справа – 2,5 см, сверху – 2 см, снизу – 3 см. Абзацный отступ – 1,25 см. Нумерация страниц – внизу посередине листа. На титульном листе номер не ставится.

Заголовки оформляются заглавными буквами шрифтом размера 14 pt, начертание – полужирное. Точка в конце заголовка не ставится.

Тексты программ необходимо набирать шрифтом Courier New 14 pt прямого начертания.

Изображения (рисунки) должны сопровождаться названиями. Названия размещаются под рисунком и оформляются обычным шрифтом, размер – 14 pt. Нумерация рисунков – сквозная в пределах всего документа. В конце названия точка не ставится. Например: «Рисунок 1. Главное окно программы». В тексте должны быть ссылки на все рисунки. Например: «Программа имеет графический интерфейс (Рис. 1)».

Названия таблиц размещаются над таблицами, например «Таблица 1 – Название таблицы», выравнивание вправо. Нумерация таблиц – сквозная в пределах всего документа. Шрифт – Times New Roman, размер – от 11 pt до 14 pt. В тексте должны быть ссылки на все таблицы.

Текст, изображения, таблицы не должны выходить за поля документа.

7. ПОРЯДОК ЗАЩИТЫ РАБОТЫ

Для защиты курсовой работы должны быть представлены преподавателю следующие материалы:

- 1) программа в виде исходного кода и исполняемого файла с набором тестовых случаев для проверки корректной работы;
- 2) Пояснительная записка в электронном и печатном виде. Электронный вариант должен быть отправлен в систему поддержки учебного процесса предварительно, не менее чем за 3 дня до защиты.

Защита включает в себя:

- 1) демонстрацию выполнения программы на тестах и/или контрольном примере, подготовленных заранее;
- 2) демонстрацию исходного кода;
- 3) ознакомление преподавателя с Пояснительной запиской;
- 4) ответы на вопросы преподавателя (например, «почему было реализовано именно таким образом», «имело ли смысл предусмотреть в программе такие-то функции» и т.п.);

При выставлении баллов за курсовую работу оценивается программа (до 50 баллов), Пояснительная записка (до 30 баллов), качество защиты работы и ответы на вопросы (до 20 баллов).

Библиографический список литературы

1. Ахо Альфред В. Структуры данных и алгоритмы: классическое издание. / Ахо Альфред В., Хопкрофт Джон Э., Ульман Джеффри Д. – Москва: Вильямс, 2018 – 400с. – Текст: непосредственный.
2. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт ; перевод Ф. В. Ткачев. — 3-е изд. — Саратов : Профобразование, 2024. — 272 с. — ISBN 978-5-4488-0101-3. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: <https://www.iprbookshop.ru/145901.html> (дата обращения: 29.09.2025). — Режим доступа: для авторизир. пользователей.
3. Кормен Томас Х. Алгоритмы. Вводный курс. / Кормен Томас Х. – Москва: Вильямс, 2020 – 208с. – Текст : непосредственный.
4. Алгоритмы. Построение и анализ. / Кормен Томас Х., Лейзерсон Чарльз И., Ривест Рональд Л., Штайн Клиффорд. – Москва: Вильямс, 2019 – 1328с.
5. Лэнгсам Й. Структуры данных для персональных ЭВМ / Й. Лэнгсам, М. Огенстайн, А. Тененбаум – Москва: Мир, 1989.– 568 с. – Текст: непосредственный.
6. Гуриков С.Р. Введение в программирование на языке Visual C# : учеб. пособие / С.Р.Гуриков. – Москва : ФОРУМ, 2018. – 447 с. – Текст : непосредственный.
7. Павловская Т.А. C/C++. Программирование на языке высокого уровня : учебник для студентов вузов, обучающихся по направлению «Информатика и вычислительная техника» / Т. А. Павловская. – Москва : Питер, 2011. - 461 с.: ил – Текст : непосредственный.
8. Стивенс Р. Алгоритмы. Теория и практическое применение / Р. Стивенс. – Москва : Издательство «Э», 2018. – 544 с. – Текст : непосредственный.
9. Подбельский В.В. Язык C#. Базовый курс : учеб. пособие / В.В. Подбельский. – Москва: Финансы и статистика, 2013. - 408 с. – Текст : непосредственный.
10. Троелсен Э. Язык программирования C# 5.0 и платформа .NET 4.5 : учеб. пособие / Э. Троелсен. – Москва: Вильямс, 2015. - 633 с. – Текст : непосредственный.
11. Шилдт Г. C# 4.0: полное руководство : учеб. пособие / Г. Шилдт пер. с англ. – М.: Вильямс, 2018 – 1056с. – Текст : непосредственный.
12. Зиборов В. Visual C# 2010 на примерах / В. Зиборов. - Москва: "БХВ-Петербург", 2011. - 432 с. – Текст : непосредственный.
13. Осипов Н.А. Разработка Windows приложений на C# : учеб. пособие / Н.А. Осипов. – Спб: НИУ ИТМО, 2012г. – 74с. // [сайт]. – URL: <https://books.ifmo.ru/file/pdf/905.pdf> (дата обращения: 29.09.2025). – Текст : электронный.

Приложение 1 Описания информационных классов для вариантов 11-37

Вариант 1. Класс *Прямоугольник* (CMyRect):

- a. скрытые поля *Длина*, *Ширина*, *Цвет*;
- b. методы, позволяющие получить как доступ для чтения к полям *Длина*, *Ширина* и *Цвет*, так и доступ для записи с предварительной проверкой на корректность;
- c. методы для вычисления *Площади* и *Периметра* прямоугольника;
- d. метод вывода, который возвращает строку с информацией об экземпляре класса;
- e. метод, который позволит изменить либо только длину и ширину прямоугольника, либо длину, ширину и цвет – в зависимости от количества параметров;
- f. операторы больше и меньше, которые могут сравнивать по площади как два прямоугольника, так и площадь прямоугольника с заданной числовой константой;
- g. статические поля, в котором будут храниться размеры наибольшего прямоугольника среди имеющихся экземпляров классов.

Вариант 2. Класс *Автомобиль* (CAuto):

- a. скрытые поля *Максимальная скорость*, *Расход топлива на 100 км.*, *Ёмкость бензобака* и *Высота*;
- b. методы, позволяющие получить как доступ для чтения к полям *Максимальная скорость*, *Расход топлива на 100 км.*, *Ёмкость бензобака* и *Высота*, так и доступ для записи с предварительной проверкой на корректность;
- c. методы для вычисления *Расстояния, на которое хватает полного бака* и *Минимального времени преодоления заданного расстояния*;
- d. метод вывода, который возвращает строку с информацией об экземпляре класса;
- e. метод, который проверяет, либо сможет ли автомобиль проехать заданное расстояние без дополнительных заправок, либо сможет ли он его проехать без дополнительных заправок и за указанное минимальное время – в зависимости от количества параметров;
- f. операторы больше и меньше, которые могут сравнивать по максимальной скорости как два автомобиля, так и максимальную скорость автомобиля с заданной числовой константой;
- g. статическое поле, в котором будут храниться количество имеющихся на данный момент экземпляров данного класса.

Вариант 3. Класс *Город* (CCity):

- a. скрытые поля *Население*, *Площадь*, *Название* и *Наличие аэропорта* (есть или нет);
- b. методы, позволяющие получить как доступ для чтения к полям *Население*, *Площадь*, *Название* и *Наличие аэропорта*, так и доступ для записи с предварительной проверкой на корректность;
- c. метод для вычисления *Плотности населения на 1 кв.м.* и *Площади на 1*
- d. метод вывода, который возвращает строку с информацией об экземпляре класса;
- e. метод, который позволит изменить либо только население, либо население и площадь, либо население, площадь и статус города – в зависимости от количества параметров;
- f. операторы больше и меньше, которые могут сравнивать по населению как два города, так и население города с заданной числовой константой;
- g. статическое поле, в котором будут храниться количество городов с аэропортом среди имеющихся экземпляров классов.

Вариант 4. Класс *Книга* (CBook):

- a. скрытые поля *Наименование*, *Автор*, *Количество страниц*, *Год издания*, *Количество экземпляров в библиотеке* и *Количество экземпляров на руках у читателей*;
- b. методы, позволяющие получить как доступ для чтения к полям *Наименование*, *Автор*, *Количество страниц*, *Год издания*, *Количество экземпляров в библиотеке* и *Количество*

экземпляров на руках у читателей, так и доступ для записи с предварительной проверкой на корректность;

- c. методы для вычисления *Количества доступных экземпляров на текущий момент времени*;
- d. метод вывода, который возвращает строку с описанием книги (автор, название, количество страниц, год издания);
- e. методы, которые позволяют оформить получение или возврат либо одной книги, либо заданного количества книг – в зависимости от наличия или отсутствия параметров;
- f. операторы больше и меньше, которые могут сравнивать по количеству страниц две книги;
- g. статическое поле, в котором будет храниться общее количество экземпляров различных книг в библиотеке.

Вариант 5. Класс *Человек* (CMan):

- a. скрытые поля *Имя, Пол, Возраст, Рост* и *Вес*;
- b. методы, позволяющие получить доступ для чтения к полям *Имя, Пол, Возраст, Рост* и *Вес* и доступ для записи с проверкой на корректность;
- c. метод для вычисления *Коэффициента* = $Вес / (Рост - 100)$;
- d. метод вывода, который возвращает строку с информацией о человеке;
- e. метод, которые позволят увеличить возраст либо на один год, либо на заданное количество лет – в зависимости от наличия или отсутствия параметров;
- f. операторы больше и меньше, которые могут сравнивать по возрасту двух людей;
- g. статические поля, в которых будут храниться количество мужчин и количество женщин среди имеющихся объектов классов.

Вариант 6. Класс *Компьютер* (CComputer):

- a. скрытые поля *Частота процессора, Количество ядер, Объем оперативной памяти, Объем жесткого диска* и *Объем занятого пространства на жестком диске*;
- b. методы, позволяющие получить как доступ для чтения к полям *Частота процессора, Количество ядер, Объем оперативной памяти, Объем жесткого диска* и *Объем занятого пространства на жестком диске*, так и доступ для записи с предварительной проверкой на корректность;
- c. методы для вычисления *Объема свободного пространства на жестком диске* и проверки, *Можно ли скопировать на жесткий диск файл указанного размера*;
- d. метод вывода, который возвращает строку с информацией о компьютере;
- e. метод, который позволит изменить либо только объем занятого места на жестком диске, либо объем занятого места на жестком диске и размер самого жесткого диска – в зависимости от количества параметров;
- f. операторы больше и меньше, которые могут сравнивать по количеству ядер два компьютера;
- g. статическое поле, в котором будет храниться общий объем всех жестких дисков среди имеющихся экземпляров класса *Компьютер*.

Вариант 7. Класс *Банковский Счет* (CAccount):

- a. скрытые поля *Имя владельца, Номер счета, Сумма, Годовой процент*;
- b. методы, позволяющие получить как доступ для чтения к полям *Имя владельца, Номер счета, Сумма*, так и доступ для записи с предварительной проверкой на корректность;
- c. метод для вычисления *ежемесячной Суммы* счета согласно *Годовому проценту*: $Сумма = Сумма + Сумма * Годовой\ процент / 12$;
- d. метод вывода, который возвращает строку с элементами счета;
- e. метод, который позволит либо изменить только сумму на счету (+/-), либо еще и годовой процент – в зависимости от количества параметров;
- f. операторы сравнения на равенство и неравенство (>) два счета по полю *сумма*;

- g. статическое поле, в котором будет храниться общая сумма на всех счетах на текущий момент.

Вариант 8. Класс Цилиндр (CCylinder):

- a. скрытые поля *Радиус*, *Высота*;
- b. методы, позволяющие получить как доступ для чтения к полям класса *Радиус*, *Высота*, так и доступ для записи с предварительной проверкой на корректность;
- c. методы для вычисления площади основания и объема цилиндра $Объем = Площадь\ основания * Высота$;
- d. метод вывода, который возвращает строку с параметрами цилиндра;
- e. метод, который позволит увеличить/уменьшить радиус цилиндра на заданную величину, либо изменить еще и высоту – в зависимости от числа параметров;
- f. операторы сравнения на равенство и неравенство (>) двух цилиндров, по полю радиус;
- g. статическое поле, в котором будет храниться количество имеющихся на данный момент экземпляров данного класса.

Вариант 9. Класс Кассовый чек (CCheck):

- a. скрытые поля *Номер*, *Сумма чека со скидкой*, *Форма оплаты (наличные/платежная карта)*, *ФИО кассира*;
- b. методы, позволяющие получить как доступ для чтения к полям класса *Номер*, *Сумма чека со скидкой*, *Форма оплаты (наличные/платежная карта)*, *ФИО кассира*, так и доступ для записи с предварительной проверкой на корректность;
- c. метод расчета суммы скидки по чеку, процент скидки – параметр метода;
- d. метод вывода, который возвращает строку с информацией о чеке;
- e. метод, который изменит форму оплаты чека, либо изменить еще и ФИО кассира – в зависимости от числа параметров;
- f. операторы сравнения на равенство и неравенство (>) двух чеков по сумме;
- g. два статических поля, в которых будут храниться соответственно суммы по чекам оплаченных наличными и платежными картами.

Вариант 10. Класс Телефонный звонок (CCall):

- a. скрытые поля *Абонент*, *Номер*, *Важность Звонка (bool)*, *Тема*, *Время начала звонка*, *Время окончания звонка*;
- b. методы, позволяющие получить как доступ для чтения к полям класса, так и доступ для записи с предварительной проверкой на корректность;
- c. метод для вычисления длительности звонка;
- d. метод вывода, который возвращает строку с информацией о звонке;
- e. метод, который позволит изменить либо только тему, либо еще и важность телефонного звонка – в зависимости от количества параметров;
- f. операторы сравнения на равенство и неравенство (>) звонков по длительности;
- g. статическое поле, в котором будет храниться количество важных звонков.

Вариант 11. Класс Вектор (CVector):

- a. представляет собой вектор (массив) произвольной длины, состоящий из целых чисел, размер которого может задаваться только при создании (через конструктор);
- b. скрытые поля *Количество элементов* и *Элементы*;
- c. общее свойство только для чтения *Количество элементов*;
- d. общее свойство, позволяющие получить как доступ для чтения к полю *Элементы*, так и установить значения в него (индексатор);
- e. общие методы для вычисления *Среднего*, *Максимального*, *Минимального значений*, *Суммы* и *Произведения всех элементов*;
- f. общий метод *Info*, который возвращает строку с информацией о векторе;

- g. общий метод, который позволит либо найти количество положительных чисел, либо количество чисел больше заданного A , либо количество чисел в диапазоне от A до B – в зависимости от количества параметров (нет параметров, один или два);
- h. операторы $+$ и $-$, которые могут складывать два вектора. Если в одном из векторов элементов меньше, чем во втором, то недостающие принимаются за «0»;
- i. операторы больше и меньше, которые могут сравнивать сумму элементов двух векторов;
- j. статическое поле, в котором будет храниться общее количество элементов в имеющихся экземплярах класса *Вектор*.

Вариант 12. Класс **Почтовый адрес** (*CMail*):

- a. скрытые поля *Страна, Индекс, Область, Район, Населенный пункт, Улица, Дом, Квартира*;
- b. общие свойства всех полей, позволяющие получить как доступ для чтения, так и установить их с предварительной проверкой на корректность;
- c. общий метод для вычисления признака проживания адресата – *город, сельская местность*;
- d. общий метод для оповещения об отсутствии необходимой информации в адресе;
- e. общий метод *Info*, который возвращает строку с информацией об экземпляре класса;
- f. общий метод, который позволит изменить либо только часть адреса, либо весь адрес – в зависимости от количества параметров;
- g. операторы равно и не равно, которые могут сравнивать весь адрес с другим адресом, так и отдельные части адреса с такими же полями другого адреса;
- h. статическое поле, в котором будет храниться количество имеющихся на данный момент экземпляров данного класса.

Приложение 2 Образец оформления титульного листа
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«ТЮМЕНСКИЙ ИНДУСТРИАЛЬНЫЙ УНИВЕРСИТЕТ»

<Институт>

<Кафедра>

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
по дисциплине **«Технологии разработки программного обеспечения»**
на тему Разработка программного обеспечения на языке C++/C#
Вариант < >

Выполнил:
студент группы <номер группы>
<Фамилия><Имя>

Проверил:
Преподаватель <ФИО преподавателя>

Тюмень
ТИУ
<год>

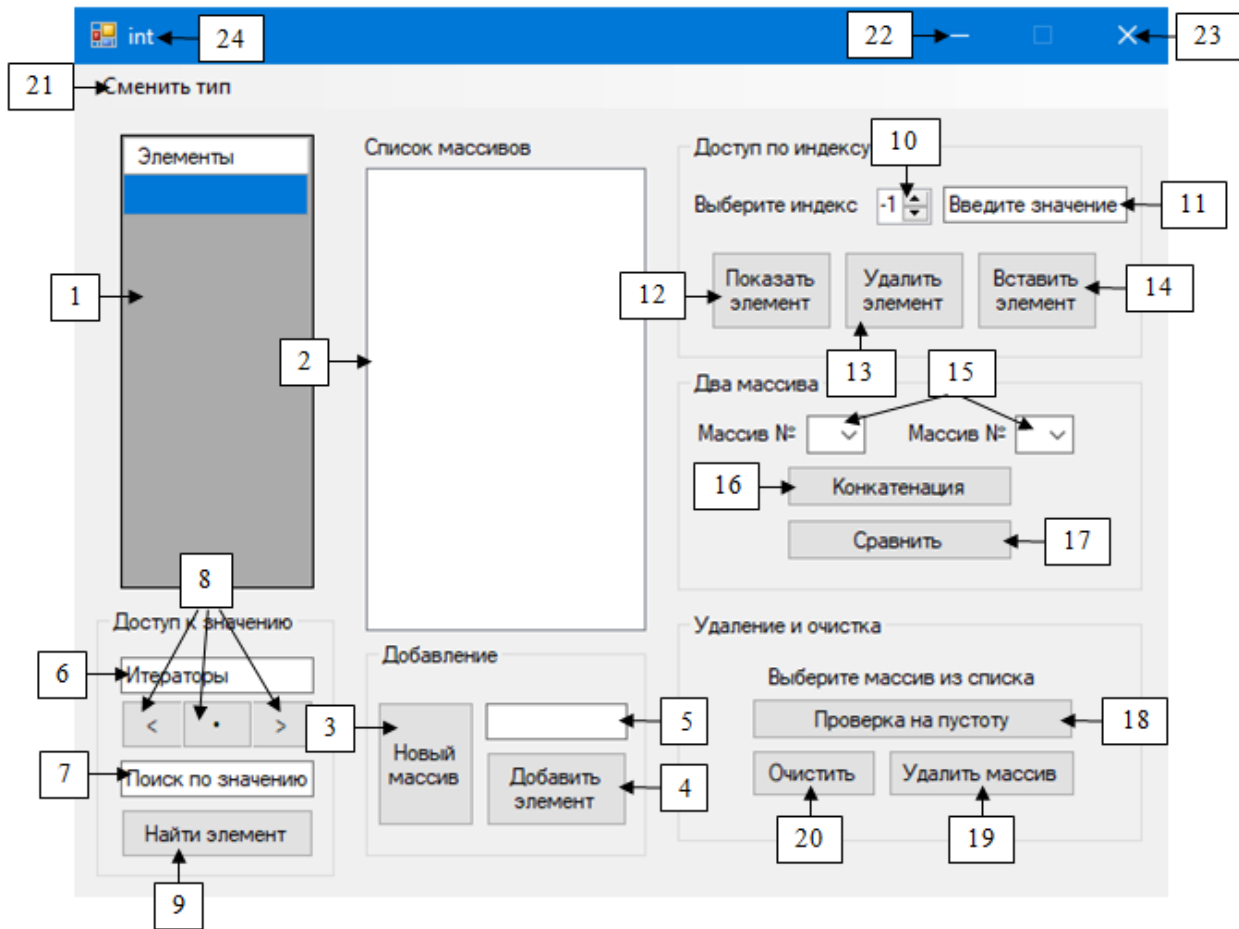


Рисунок 1. Главное окно программы

1.array1_dataGridView: DataGridView – таблица, в которой отображаются элементы выбранного массива.

2.array_listBox: ListBox – список, в котором отображаются массивы.

3.new_array_button: Button – срабатывает при событии Click и вызывает модальную форму, в которой пользователь должен ввести размер массива.

4.add_node_button: Button – срабатывает при событии Click и добавляет в выбранный массив элемент, введенный в add_node_textbox.

5.add_node_textbox: Textbox– поле ввода элементов массива.

6.iterator_textBox: Textbox – поле, в котором отображаются элементы массива при использовании итераторов.

7.find_by_value_textBox: Textbox поле ввода элементов для поиска по значению.

8.prev_button, curr_button, next_button: Button – срабатывает при событии Click и отображает элементы в iterator_textBox.

И т.д. Продолжение описания компонент интерфейса.

Приложение 4 Пример диаграммы классов

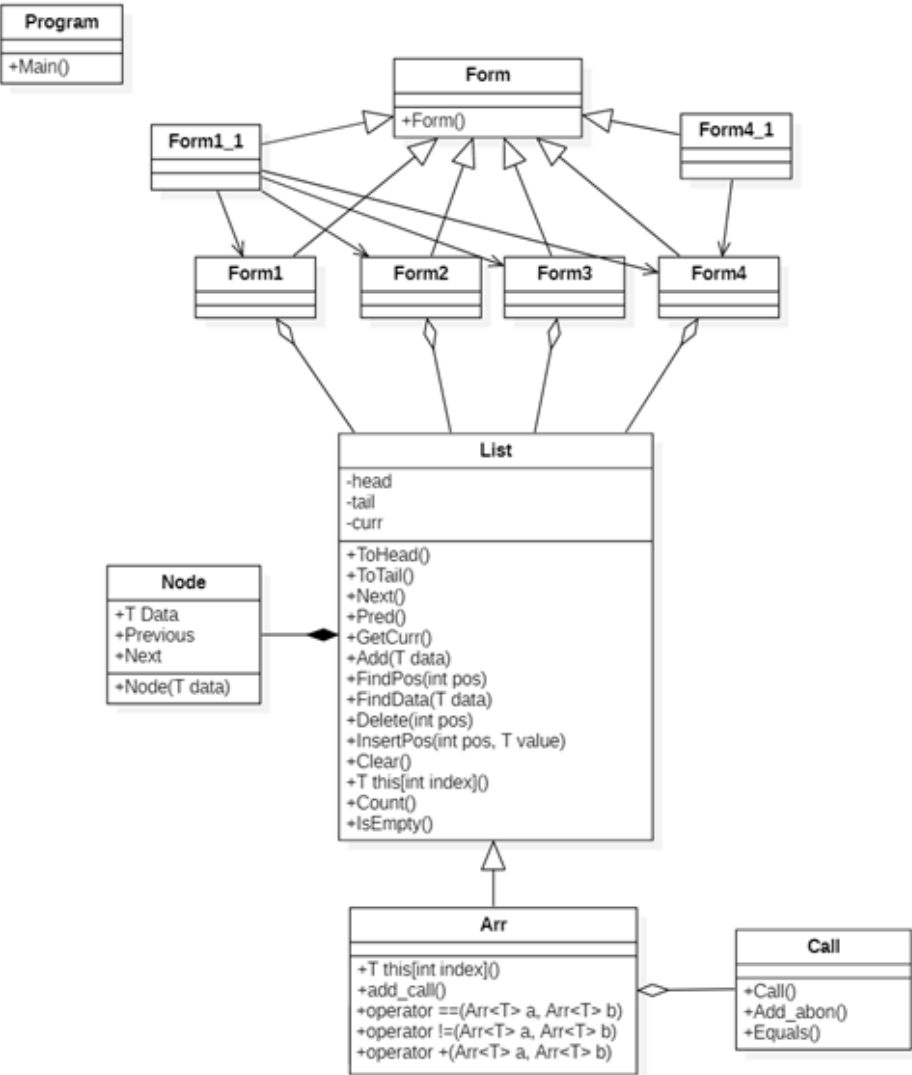


Рисунок 1. Диаграмма классов

Таблица 1 – Класс Node

Параметр	Значение
Комментарий	Класс, представляющий собой узел списка
Поля	public T Data – значение узла public Node<T> Previous – предыдущий узел public Node<T> Next – следующий узел
Методы	public Node(T data) конструктор с параметрами

Таблица 2 – Класс ListP

Параметр	Значение
Комментарий	Класс, представляющий собой двусвязный список
Поля	private Node<T> head – голова списка private Node<T> tail – хвост списка private Node<T> curr – текущий элемент списка int count – счётчик количества элементов списка
Методы	void ToHead() – итератор – установка на голову void ToTail() – итератор – установка на хвост void Next() – итератор – установка на следующий элемент списка void Pred() – итератор – установка на предыдущий элемент списка T GetCurr() – метод для получения значения текущего элемента списка void Add(T data) – добавление элемента в список Node<T> FindPos(int pos) – поиск элемента по позиции int FindData(T data) – поиск элемента по значению void Delete(int pos) – удаление элемента из списка void InsertPos(int pos, T value) – вставка элемента на выбранную позицию void Clear() – очистка списка int Count – метод, возвращающий счётчик количества элементов bool IsEmpty – проверка списка на пустоту abstract T this[int index] { get; set; } – абстрактный метод, доступ к элементу по индексу Все операции имеют модификатор доступа – public.

Приложение 6 Пример спецификации состава программы

Таблица 1 – Спецификация состава программы

Обозначение	Наименование	Примечание
course_work.sln	Решение Visual Studio	Собирает все файлы проекта в единое решение
course_work.csproj	Файл проекта Visual C#	Исполнительный файл
Form1.cs	Файл исходного кода Visual C#	Файл главной формы
Form1.Designer.cs	Файл исходного кода Visual C#	Файл графических объектов главной формы
Form1.resx	Управляемый файл ресурсов Microsoft.NET	Двоичный файл, содержащий данные ресурсов .NET
Form2.cs	Файл исходного кода Visual C#	Файл модальной формы
Form2.Designer.cs	Файл исходного кода Visual C#	Файл графических объектов модальной формы
Form2.resx	Управляемый файл ресурсов Microsoft.NET	Двоичный файл, содержащий данные ресурсов .NET
Form3.cs	Файл исходного кода Visual C#	Файл модальной формы
Form3.Designer.cs	Файл исходного кода Visual C#	Файл графических объектов модальной формы
Form3.resx	Управляемый файл ресурсов Microsoft.NET	Двоичный файл, содержащий данные ресурсов .NET
Program.cs	Файл исходного кода Visual C#	Содержит в себе исходный код программного обеспечения
Node.cs	Файл исходного кода Visual C#	Файл класса узел
List.cs	Файл исходного кода Visual C#	Файл класса связного списка
ArrayOfSize.cs	Файл исходного кода Visual C#	Файл класса массива на базе списка
Call.cs	Файл исходного кода Visual C#	Файл класса пользовательского типа данных

Приложение 7 Пример описания модулей программы

Таблица 1 – Описание модулей программы

Имя файла	Выполняемые функции
Form1	Экранная форма, реализует интерфейс, предоставляющий пользователю возможность добавлять массивы, а также заполнять их элементами и производить дальнейшие действия с ними. Реализует следующие функции: функция добавления, удаления, опрос размера массива, очистка массива, получение доступа к элементам массива по номеру позиции и по значению, проверка массива на пустоту, сравнение массивов, конкатенация массивов, переход на начало массива, на предыдущий и следующий элементы с помощью итератора. Работает с типом данных int
Form2	Экранная форма, идентичная по функционалу и интерфейсу форме Form1.cs, но работающая с типом данных double
Form3	Экранная форма, идентичная по функционалу и интерфейсу форме Form1.cs, но работающая с типом данных string
Form4	Экранная форма, идентичная по функционалу и интерфейсу форме Form1.cs, но работающая с пользовательским типом данных Call
Form1_1	Экранная форма, предназначенная для ввода размера массива
Form4_1	Экранная форма, реализует интерфейс, предоставляющий пользователю возможность ввести значения для элемента массива типа Call
Program	Класс, предназначенный для запуска программы
Node	Класс, представляющий собой узел. Содержит в себе указатель на голову, хвост и текущий элемент списка
List	Класс, представляющий собой связный список. Содержит в себе все базовые операции обработки связных списков: установка на начало списка, установка в конец списка, установка на следующий элемент, установка на предыдущий элемент, добавление элемента, поиск элемента по позиции, поиск элемента по значению, удаление элемента с заданной позиции, вставка элемента в указанную позицию, очистка списка
ArrOfSize	Класс, представляющий собой массив на базе списка. Содержит в себе индексатор, позволяющий получить доступ к элементу по индексу, а также операции конкатенации и сравнения двух массивов
Call	Пользовательский тип данных. Содержит в себе следующие поля: номер абонента, имя абонента, важность звонка, тема звонка, время начала разговора, время конца разговора