**Министерство науки и высшего образования Российской Федерации**

**федеральное государственное бюджетное образовательное учреждение высшего образования «Казанский национальный исследовательский технический университет им. А.Н. Туполева-КАИ»**

**(КНИТУ-КАИ)**

**Институт Компьютерных технологий и защиты информации**

**Кафедра Динамики процессов и управления**

**ОТЧЕТ**

**по лабораторным работам «АиСД»**

Направление подготовки/специальность: **09.03.03 «Прикладная информатика»**

Выполнил:

Обучающийся гр. 4217 Старостин Э.Р

Казань, 2021 год

Оглавление

[Лабораторная работа №1 3](#_Toc72957708)

[Множества. 3](#_Toc72957709)

[Лабораторная работа №2 и №3 4](#_Toc72957710)

[Сортировки. 4](#_Toc72957711)

[Лабораторная работа №4 и №5 5](#_Toc72957712)

[Односвязные и двусвязные списки. 5](#_Toc72957713)

[Лабораторная работа №6 10](#_Toc72957714)

[Бинарные деревья 10](#_Toc72957715)

[Лабораторная работа №7 11](#_Toc72957716)

[Хеш-таблицы. 11](#_Toc72957717)

[Лабораторная работа № 8 12](#_Toc72957718)

[Графы. 12](#_Toc72957719)

[Список литературы 16](#_Toc72957720)

# Лабораторная работа №1

## Множества.

Множеством называется произвольный набор (совокупность, класс, семейство) каких-либо объектов. Объекты, входящие во множество, называются его элементами. Если объект *x* является элементом множества А, то говорят, что *x* принадлежит А.

Для выполнения первой лабораторной работы мной был создан обобщенный класс Set<T>, в котором параметр типа «T», реализует интерфейс «Icomparable».

Класс Set<T> принимает следующие значения:

* Public int size, count; (Размер множества и кол-во элементов в множестве)
* T[] data; (Массив данных)

Класс Set<T> также содержит пустой конструктор и конструктор с выше перечисленными данными.

Методы и свойства данного класса:

* Size { get {return size; } } – свойство, определяющее размер множества.
* Count { get {return count; } } – свойство, определяющее количество элементов в множестве.
* IsContains(T element) – метод для проверки наличия элемента в множестве.
* GetIndex(T element) – метод для получения индекса элемента.
* GetElementByIndex(int index) – метод для получения данных из множества по индексу.
* SetElementByIndex(int index, T newElevent) – метод для изменения данных из множества по индексу.
* Add(T element) – метод для добавления нового элемента в множество.
* Copy() – метод для копирования множеств.
* RemoveByIndex(int index) – метод для удаления элемента из множества по индексу.
* RemoveByElement(T element) – метод для удаления элемента из множества.
* Search(T el) – метод для поиска элемента в множестве.
* Union(Set<T> s2) – объединение множеств. Объединением двух множеств A и B называется множество
* Intersection(Set<T> set2) – пересечение множеств. Пересечением двух множеств А и В называется множество
* Resize(T newSize) – метод для изменения размера множества.
* Difference(Set<T> set) – дополнение (разность) множеств. Дополнение А\В состоит из всех элементов множества А, которые не принадлежат В
* GetList() – метод для получения набора множеств из последовательности элементов множества
* Perevod(int chislo, int osnov) – вспомогательный метод, выполняющий перевод из 2-ой системы в 10-ю.
* SetOfSubSets() – множество множеств или показательное множество (булеан). Булеан — это множество всех подмножеств, включая пустое множество и само множество.
* ToString();

Помимо стандартной реализации методов для добавления, объединения, пересечения множества и т.д., в C# существует возможность реализовать их при помощи «operator».

Использованная литература: 1, 4, 8.

# Лабораторная работа №2 и №3

## Сортировки.

Для выполнения лабораторной работы №2 и №3, мной был создан класс Sort.

Класс Sort содержит в себе следующие методы (сортировки):

I) Обычные:

1. BubbleSort<T>(T[] list) – пузырьковая сортировка:

Пузырьковая сортировка предполагает следующее: если массив не отсортирован, любые два смежных элемента в нем находятся в неправильном положении. Из-за этого алгоритм должен проходить по массиву несколько раз, меняя местами все неправильные пары.

2. Swap(ref T a, ref T b) – метод обмена элементов.

3. InsertionSort<T>(T[] array) – сортировка вставками:

Основная идея такого метода заключается в выборе элемента из списка ввода и его вставке в соответствующую позицию отсортированного списка вывода, который изначально пуст.

4. IndexOfMin<T>(T[] list, int n) – метод для нахождения минимального индекса. Является вспомогательным метод для сортировки выбором.

5. SelectionSort<T>(T[] list, int currentIndex = 0) – сортировка выбором:

Ее суть состоит в том, чтобы найти в списке ввода наибольший элемент и добавить его в конец отсортированного списка. Как вариант, можно искать наименьший элемент и перемещать его в начало отсортированного списка.

6. ShellSort<T>(T[] list) – сортировка Шелла:

Идея сортировки Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами.

7. ShakerSort<T>(T[] list) – сортировка «Шейкер» или по-другому усовершенствованная пузырьковая сортировка:

Идея: выполнять несколько перестановок за проход. Например, при движении вниз по массиву элемент (назовем его K) может сменить свою позицию не один раз, прежде чем займет нужное место. Мы сэкономим время, если не будем передвигать его по массиву, а сохраним во временной переменной и, сместив другие элементы вверх, найдем целевую позицию для K, вставим его туда и продолжим прохождение.

II) Рекурсивные:

8. Partition<T>(T[] list, int minIndex, int maxIndex) – метод для поиска разделяемого элемента в быстрой сортировке.

9. RandomizedPartition<T>(T[] list, int minIndex, int maxIndex) – метод для поиска разделяемого элемента, используя Random.

12. QuickSort<T>( T[] list, int minIndex, int maxIndex) – быстрая сортировка:

13. Merge<T>(T[] list, int minIndex, int middleIndex, int maxIndex) – метод для слияния массивов. Является вспомогательным методом для сортировки слиянием.

14. MergeSort<T>( T[] list, int minIndex, int middleIndex, int maxIndex) – сортировка слиянием:

Подобно быстрой сортировке, сортировка слиянием использует стратегию «разделяй и властвуй». Если в первом случае выбирается разделитель, а оставшиеся элементы делятся на две группы — большие и меньшие его, то во втором разделение происходит на две равные части, а затем алгоритм рекурсивно вызывает сам себя для их сортировки. Потом отсортированные половины сливаются в комбинированный отсортированный список (при помощи Merge<T>()).

Использованная литература: 1, 2, 5, 7, 10.

# Лабораторная работа №4 и №5

## Односвязные и двусвязные списки.

I) Односвязные списки:

Односвязный список построен из объектов, обычно называемых ячейками. Этот класс содержит все данные, которые должны храниться в списке, и ссылку на другую ячейку. Ссылка представляет собой справку или указатель на объект такого же класса. Поле типа «указатель» в ячейке часто называется Next.

Для реализации односвязного списка мной были созданы следующие два класса:

1. SNode<T> - обобщенный класс, для создания данных об узле.

2. SList<T> - обобщенный класс для реализации методов односвязного списка.

1) Класс SNode<T> принимает следующие значения:

* public int Key { get; set; }; (Ключ)
* public T Value { get; set; }; (Значение)
* public SNode<T> Next { get; set; }; (Ссылка на следующую ячейку)

Класс SNode<T> также содержит пустой конструктор и конструктор с выше перечисленными данными.

Метод: ToString();

2) Класс SList<T> принимает следующие значения:

* SNode<T> top; (Ссылка на первый узел)

Класс SList<T> также содержит пустой конструктор и конструктор с выше перечисленными данными.

Методы и свойства:

* public int count { get; set; } – свойство, определяющее количество элементов (узлов) в списке.

В односвязном списке каждая ячейка связана со следующей с помощью одинарной ссылки. Чтобы использовать такой список, нам понадобится алгоритм для передвижения по списку, поиска, добавления и удаления элементов.

* View() – метод для вывода на экран значения ячеек и ключа.

Данный алгоритм начинается с цикла While, который работает до тех пор, пока верхний указатель ячейки не станет null. Внутри цикла алгоритм сперва вызывает Console.WriteLine(), чтобы показать значение ячейки top, а затем с ее помощью указывает на следующую ячейку связного списка. Процесс продолжается до тех пор, пока top не станет указывать на null в конце списка и цикл While не остановится.

Методы для поиска узла в списке:

* FindByValue(T value) – метод поиска узла по значению.
* FindByIndex(int index) – метод поиска узла по индексу.
* FindByKey(int key) – метод поиска узла по ключу.

Поиск ячейки выглядит так: алгоритм передвигается по связному списку и останавливается, найдя нужную. Алгоритм вводит цикл While, который работает, пока top не станет равным null. Внутри этого цикла сравнивается значение ячейки top с искомым значением. Если они совпадают, алгоритм возвращает top; если нет, top превращается в указатель на следующую ячейку в списке. Если top становится равным null после того, как будет пересмотрен весь список, — искомого значения нет.

Методы добавления узла в список:

* AddTop(int k, T v) – метод добавления узла в начало списка.

Иллюстрационный пример добавления узла в начало списка представлен графически на рис. 4:

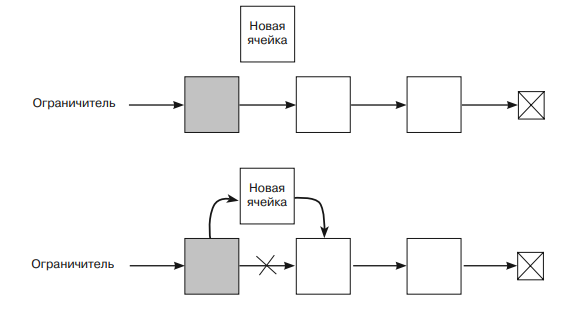


Рис. 4

* AddEnd(int k, T v) – метод добавления узла в конец списка.

Иллюстрационный пример добавления узла в конец списка представлен графически на рис. 5:

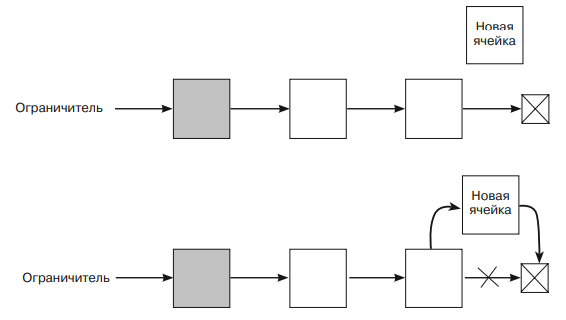


Рис. 5

* InsertByBeforeValue(T select, int key, T value) – метод вставки узла до выбранного элемента.
* InsertByAfterValue(T select, int key, T value) – метод вставки узла после выбранного элемента.

Метод вставки узла после выбранного элемента представлен графически на рис. 7:

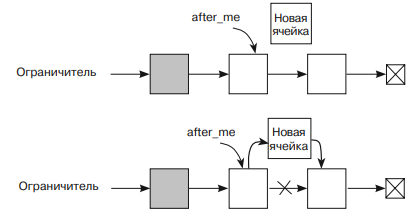


Рис. 6

Методы удаления узла из списка:

* RemoveTop() – метод удаления узла из начала списка.
* RemoveEnd() – метод удаления узла из конца списка.
* RemoveByValue(T value) – метод удаления узла по значению.
* RemoveByIndex(int index) – метод удаления узла по индексу.
* RemoveByKey(int key) – метод удаления узла по ключу.

Идейная сторона удаления узла после выбранного значения/ключа/индекса представлена графически на рис. 7:

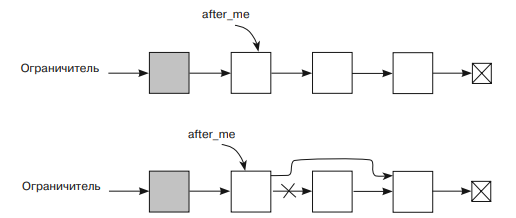


Рис. 7

* IsContainsByValue(T value) – метод проверки наличия элемента в односвязном списке.
* Clear() – очистка.

II) Двусвязные списки:

В двусвязных списках ссылки ячеек указывают на следующие и предыдущие ячейки. Вторые часто называют Prev.

Для реализации односвязного списка мной были созданы следующие два класса:

1. DNode<T> - обобщенный класс, для создания данных об узле.

2. DList<T> - обобщенный класс для реализации методов двусвязного списка.

1) Класс DNode<T> принимает следующие значения:

* public int Key { get; set; }; (Ключ)
* public T Value { get; set; }; (Значение)
* public DNode<T> Next { get; set; }; (Ссылка на следующую ячейку)
* public DNode<T> Prev { get; set; }; (Ссылка на предыдущую ячейку)

Класс DNode<T> также содержит пустой конструктор и конструктор с выше перечисленными данными.

Метод: ToString();

2) Класс DList<T> принимает следующие значения:

* DNode<T> head; (головной/первый элемент)
* DNode<T> tail; (последний/хвостовой элемент)

Класс DList<T> также содержит пустой конструктор и конструктор с выше перечисленными данными.

Пример двусвязного списка показан на рис. 8:

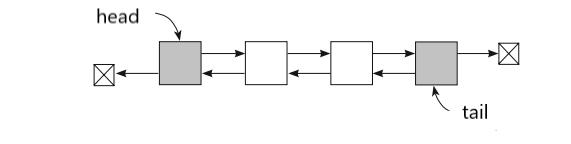


Рис. 8

Методы и свойства:

* public int count { get; set; } – свойство для определения количества элементов в списке.

DNode<T> top;

DNode<T> end;

Методы вывода на экран двусвязного списка:

* ViewForward() – метод прохода по списку от начала до конца.
* ViewBack() – метод прохода по списку от конца к началу.

Методы поиска в двусвязном списке:

* FindByValue(T value) – метод поиска узла по значению.
* FindByIndex(int index) – метод поиска узла по индексу.
* FindByKey(int key) – метод поиска узла по ключу.

Методы добавления в двусвязном списке:

* AddTop(int k, T v) – метод добавления узла в начало списка.
* AddEnd(int k, T v) – метод добавления узла в конец списка.
* InsertByBeforeValue(T select, int key, T value) – метод вставки узла до выбранного элемента.
* InsertByAfterValue(T select, int key, T value) – метод вставки узла после выбранного элемента.

Идея вставки узла после выбранного элемента представлена графически на рис. 9:

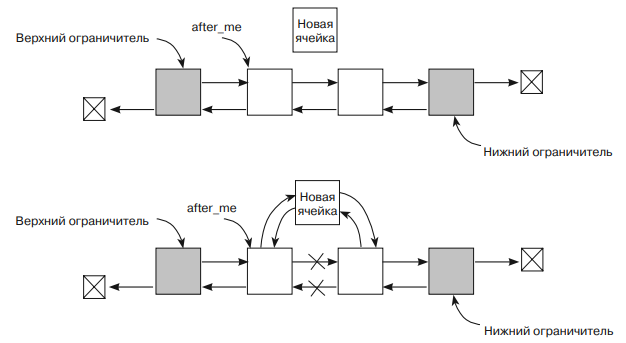


Рис. 9

Методы для удаления узла из списка:

* RemoveTop() – метод удаления узла из начала списка.
* RemoveEnd() – метод удаления узла из конца списка.
* RemoveByValue(T value) – метод удаления узла по значению.
* RemoveByIndex(int index) – метод удаления узла по индексу.
* RemoveByKey(int key) – метод удаления узла по ключу.
* IsContainsByValue(T value) – метод проверки наличия элемента в двусвязном списке.
* Clear() – очистка.

В классе Program создаются связные списки и в консольном приложении выводятся на экран, используя выше перечисленные методы.

Использованная литература: 1, 2, 4, 7.

# Лабораторная работа №6

## Бинарные деревья

Дерево состоит из вершин (узлов), которые содержат данные и соединяются ветвями. Последние изображают в виде стрелок, ведущих от родительской вершины к дочерней.

Дерево — особый вид сети, или графа, поэтому по аналогии ветви, соединяющие вершины, иногда называют ссылками или линиями.

У каждой вершины, за исключением корневой, есть одна родительская вершина. Две вершины с общим родителем иногда называют узлами-сестрами, дочерние вершины дочерних вершин — потомками, а родительские вершины родительских вершин — предками.

Бинарное дерево – это дерево, в котором значение каждого родительского узла больше значения его левого дочернего узла и меньше правого или равно ему (рис. 10).

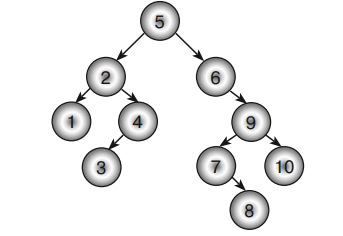


Рис. 10

Для реализации бинарных деревьев мной было создано 2 класса:

1. NodeTree<T> - обобщенный класс для создания данных об узле дерева.

2. BinaryTree<T> - обобщенный класс для реализации методов бинарного дерева.

1) Класс NodeTree<T> принимает следующие значения:

* public int Key; (Ключ)
* public T Value; (Значение)
* public NodeTree<T> Parent; (Ссылка родительскую вершину)
* public NodeTree<T> Left; (Ссылка на левую дочернюю вершину)
* public NodeTree<T> Right; (Ссылка на правую дочернюю вершину)

Класс NodeTree<T> также содержит пустой конструктор и конструктор с выше перечисленными данными.

Метод: ToString();

2) Класс BinaryTree<T> принимает следующие значения:

* NodeTree<T> root; (Свойство корня дерева)

Класс BinaryTree<T> содержит пустой конструктор.

Методы:

* AddNode(int k, T v) – добавление узла в бинарное дерево.
* Search(int k) – поиск узла в бинарном дереве.
* DeleteNode (int key) – удаление узла в бинарном дереве.
* MinNode(NodeTree<T> tmp) – нахождение минимального узла в бинарном дереве.
* MaxNode(NodeTree<T> tmp) – нахождение максимального узла в бинарном дереве .
* NextNode (NodeTree<T> tmp) – поиск следующего узла в бинарном дереве.
* PrevNode(NodeTree<T> tmp) – поиск предыдущего узла в бинарном дереве.
* LeftRotate(NodeTree<T> x) – левый поворот.
* RightRotate(NodeTree<T> x) – правый поворот.
* View1 (NodeTree<T> tmp) – симметричный обход, то есть выводит на экран бинарное дерево в порядке возрастания.
* View2 (NodeTree<T> tmp) – обход в обратном порядке, то есть выводит на экран бинарное дерево в порядке убывания.
* View3(NodeTree<T> tmp) – обход в прямом порядке.

Использованная литература: 1, 2, 3, 9.

# Лабораторная работа №7

## Хеш-таблицы.

Хеш-таблицы – это структура данных, представляющая собой специальным образом, организованный набор элементов, хранимых данных. Все данные хранятся в виде пар хеш-значения. Данная структура похожа на словарь (map c#), но имеет особенности такие как применение хеш-функции для увеличения скорости поиска. Принцип работы данной структуры схож с каталогом книг. Все книги разложены в алфавитном порядке, но не на одном стеллаже, а для каждой буквы выделен отдельный стеллаж, поэтому нам не нужно по порядку перебирать все книги, а можно подойти к нужному стеллажу и искать уже там.

Хеш-таблица позволяет очень быстро хранить и находить значения. Если она обладает разумным коэффициентом заполнения, то отыскать в ней нужный элемент несложно. Когда данных становится слишком много, скорость поиска снижается. Если же их очень мало, то ресурсы памяти расходуются неэкономно.

Для реализации хеш-таблиц мной было создано 2 класса:

1. HTLIST<T> - обобщенный класс для реализации хеш-таблиц на основе списков.

2. HashTable<T> - обобщенный класс для реализации хеш-таблиц на основе массивов.

1) Класс HTLIST<T> принимает следующие значения:

* public List<Hash<T>>[] lists;
* public int Size; (Количество списков)
* public int count; (Количество всех элементов в хеш-таблице)

Методы и свойства:

* KeySize(int key) – метод, возвращающий хеш-функцию.
* DeleteKey(int ke)-удаление по ключу
* Add(int key, T value) – метод добавления ключа и значения в хеш-таблицу.
* Resize(int newSize) – метод, меняющий размер хеш-таблицы.
* SearchByKey(int key) – метод поиска.
* View() – метод, позволяющий вывести результат на экран.

2) Класс HashTable<T> принимает следующие значения:

* public Hash<T>[] KeyValues;
* public int[] active; (Массив для активных значений)
* public int Size; (Количество массивов)
* public int count; (Количество всех элементов в хеш-таблице)

Методы и свойства:

* KeySize (int key) – метод, возвращающий хеш-функцию.
* Add(int k, T value) – метод добавления ключа и значения в хеш-таблицу.
* DeleteKey (int key) – метод удаления элемента по ключу.
* Resize(int start) – метод, меняющий размер хеш-таблицы.
* Search(int key) – метод поиска.
* SearchKe(int key) – метод поиска.
* View() – метод, позволяющий вывести результат на экран.

Использованная литература: 1, 2, 3, 6, 7,

# Лабораторная работа № 8

## Графы.

Существует два основных типа графов:

1. Ориентированные графы;

2. Неориентированные графы;

1) Ориентированный граф или орграф представляет собой пару G = (F, Е), где V — конечное множество вершин, а Е — отношение на V. Графическое изображение графа состоит из множества помеченных вершин с ориентированными ребрами (называемых дугами), соединяющими пары вершин. Совокупность всех дуг образует множество Е.

Пример ориентированного графа изображен на рис. 11:

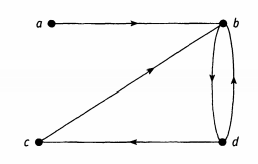


Рис. 11

2) Граф G = {V, Е) состоит из множества V, чьи элементы называют вершинами графа, и множества Е его ребер, соединяющих некоторые пары вершин.

Пример неориентированного графа изображен на рис. 12:

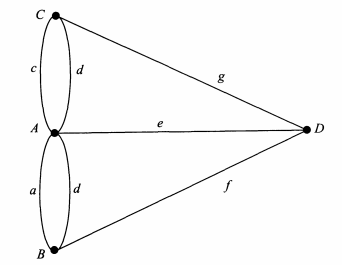


Рис. 12

Для реализации графов мной было создано три класса:

1. Vertex – класс для описания вершин.

2. Edge – класс для описания ребер.

3. Graph – класс графов.

Класс Vertex принимает следующие значения:

* public string Name { get; set; } (Имя)
* public Vertex prev; (Предыдущая вершина, от которой пришли)
* public double distance; (Суммарное расстояние)
* public bool visited ;-была ли посещена
* public List<Edge > Edges; (Набор смежных ребер т.е. ребра, которые выходят из данной вершины)
* public int discovered; (Обнаруженная вершина)
* public int finished; (Обработанная вершина)
* public int time; (Метка времени)

Класс Vertex также содержит пустой конструктор и конструктор с выше перечисленными данными.

Методы и свойства:

* ConnectWE (Edge e)-соединяет вектора
* ViewEdges()-показывает все соединения
* ToString();

2) Класс Edge принимает следующие значения:

* public Vertex First { get; set; } (От куда)
* public Vertex End { get; set; } (Куда)
* public double Length-длина

Класс Edge также содержит пустой конструктор и конструктор с выше перечисленными данными.

Метод: ToString();

3) Класс Graph принимает следующие значения:

* public List<Vertex > allvertexes = new List<Vertex >(); (Список всех вершин)
* public List<Edge > alledges = new List<Edge >(); (Список всех ребер)

Методы и свойства класса:

* AddVertex(Vertex vertex) – метод добавления вершины.
* AddEdge(Vertex from, Vertex to) – метод добавления ребра.
* RemoveVertex(Vertex ver)-удаление вершины
* Connect(Vertex f,Vertex e,double len=1.0)-соединение векторов
* ViewGraph()-выводит граф
* BFS(Vertex start) – поиск в ширину. Поиск в ширину (breadth-first search, BFS) — один из методов обхода графа. Пусть задан граф G = (V, E) и выделена исходная вершина s. Алгоритм поиска в ширину систематически обходит все ребра G для «открытия» всех вершин, достижимых из s, вычисляя при этом расстояние (минимальное количество рёбер) от s до каждой достижимой из s вершины. Алгоритм работает как для ориентированных, так и для неориентированных графов.
* DFS(Vertex startVertex) и DFS\_Visit(Vertex u) – поиск в глубину. Поиск в глубину (Depth-first search, DFS) — один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

На рис. 13 изображен несвязный граф с 3-мя компонентами связности:

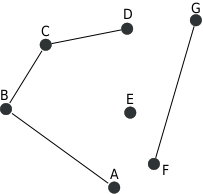


Рис. 13

* PrintPath(Vertex startVertex, Vertex vertex) и View() – методы вывода графа на экран.

Использованная литература: 1, 3, 4.

# Список литературы

1. Т. Кормен, Ч. Лейзерсон, Р. Ривест, Алгоритмы: построение и анализ / Пер. с англ. под ред. А. Шеня. – М.: МЦНМО, 2002. – 960 с.: 263 ил.
2. Стивенс, Род. Алгоритмы. Теория и практическое применение / Род Стивенс. – Москва: Издательство «Э», 2016. – 544 с.
3. Тюкачев Н. А. C#. Алгоритмы и структуры данных: учебное пособие для СПО / Н. А. Тюкачев, В. Г. Хлебостроев. — Санкт Петербург: Лань, 2021. — 232 с.: ил.
4. Кормен, Томас Х., Алгоритмы: вводный курс.: Пер. с англ. – М.: ООО «И. Д. Вильямс», 2014. – 208с.: ил. – Парал. тит. англ.
5. Н.Вирт, АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ М.: Мир, 1989, 360 стр.
6. Мясникова, Нелли Александровна, Алгоритмы и структуры данных: учебное пособие / Н. А. Мясникова. – Москва: КНОРУС, 2021. – 186 с.
7. Седжвик Роберт, Фундаментальные алгоритмы на С++, Анализ/Структуры данных/Сортировка/Поиск: Пер. с. англ./ Роберт Седжвик. – К.: Издательсво «ДиаСофт», 2001. – 688 с.
8. Трамбле Ж., Соренсон П., Введение в структуры данных: Пер. с англ./Пер. В. И. Бриккер и др.; Под ред. А. Е. Костина, В. Ф. Шаньгина. – М.: Машиностроение, 1982. – 784 с., ил.
9. https://neerc.ifmo.ru/wiki/index.php?title [Бинарное дерево поиска].
10. https://www.toptal.com/developers/sorting-algorithms [Sorting Algorithms Animations].