

**В.Д. Колдаев**

**Лабораторный практикум по курсу  
«Алгоритмы и структуры данных»**

Часть 1

Утверждено редакционно-издательским советом университета

Москва 2019

УДК 004.42(075.8)

Рецензент докт. физ.-мат. наук, проф. *В.В. Уздовский*

**Колдаев В.Д.**

Лабораторный практикум по курсу «Алгоритмы и структуры данных». Часть 1. - М.: МИЭТ, 2019. - с.: ил.

Рассмотрен широкий круг алгоритмов обработки линейных и нелинейных структур данных, без знания которых невозможно современное компьютерное моделирование. Приведены основные понятия и определения, технология работы и фрагменты программ. Содержит семь лабораторных работ, выполнение которых предусматривает использование компьютерных обучающе-контролирующих программ, разработанных в институте системной и программной инженерии и информационных технологий (СПИНТех).

Для студентов всех специальностей НИУ МИЭТ.

© МИЭТ, 2019

## Предисловие

В последние годы программирование для персональных компьютеров вылилось в дисциплину, владение которой стало ключевым моментом, определяющим успех многих инженерных проектов, а сама она превратилась в объект научного исследования. Специалисты в области программирования показали, что программы поддаются точному анализу, основанному на строгих математических рассуждениях. Убедительно продемонстрировано, что можно избежать многих ошибок, традиционных для программистов, если они будут осмысленно пользоваться методами и приемами, которые раньше применялись интуитивно. При этом основное внимание программисты обычно уделяют построению и анализу программы, а выбор представления данных, как правило, удостоивается меньшего, явно второстепенного внимания.

Современная методология программирования предполагает, что оба аспекта программирования - запись алгоритма на языке программирования и выбор структур представления данных - заслуживают абсолютно одинакового внимания, т.е. решение о том, как представлять данные, невозможно принять без понимания того, какие алгоритмы будут к ним применяться, и, наоборот, выбор алгоритма часто очень сильно зависит от строения данных, к которым он применяется.

Ни одна информационная система не обходится без программного обеспечения, более того, она просто не может существовать без этой компоненты. Поэтому задача настоящего лабораторного практикума состоит в следующем:

- познакомить со всем разнообразием имеющихся структур данных, показать, как эти структуры реализованы в языках программирования;
- познакомить с основными операциями, которые выполняются над структурами данных;
- показать особенности структурного подхода к разработке алгоритмов, продемонстрировать порядок разработки алгоритмов.

Ценность настоящего лабораторного практикума состоит в том, что он предназначен не столько для обучения технике программирова-

ния, сколько для обучения, если это возможно, «искусству» программирования.

Часть 1 практикума содержит семь лабораторных работ. В конце каждой лабораторной работы содержится большое число задач для самостоятельного решения и контрольных вопросов. В приложении содержится решение шести логических задач на алгоритмическом языке C++.

Теоретическая часть лабораторных работ изложена кратко и носит справочный характер, ее цель - дать основу для решения практических задач. В каждом разделе содержатся задачи с подробными решениями. Приведенные задания для самостоятельной работы во многом покрывают контрольные и зачетные задачи. Предполагается, что каждая тема осваивается за несколько этапов.

*Отчет по лабораторной работе* должен содержать:

- 1) конспект лабораторной работы;
- 2) демонстрационные примеры, анализирующие наиболее типичные приемы алгоритмического решения поставленной задачи;
- 2) результаты вычислений для каждого шага алгоритма;
- 3) программы на языке C++;
- 4) результаты выполнения работы (таблицы, графики, диаграммы);
- 5) выводы по работе.

Контроль усвоения дисциплины проводится с применением различных форм текущего контроля: опросы, тестирование, контрольные и самостоятельные работы.

# Лабораторная работа № 1

## Методы сортировки

**Цель работы:** ознакомление с алгоритмами сортировки линейных и нелинейных структур и методикой оценки эффективности алгоритмов.

**Продолжительность работы:** 2 часа.

### Теоретические сведения

Упорядочение элементов множества в возрастающем или убывающем порядке называется сортировкой. С упорядоченными элементами проще работать, чем с произвольно расположенными: легче найти необходимые элементы, исключить, вставить новые. Сортировка применяется при трансляции программ, при организации наборов данных на внешних носителях, при создании библиотек, каталогов, баз данных и т.д. Алгоритмы сортировки можно разбить на несколько групп (рис.1).

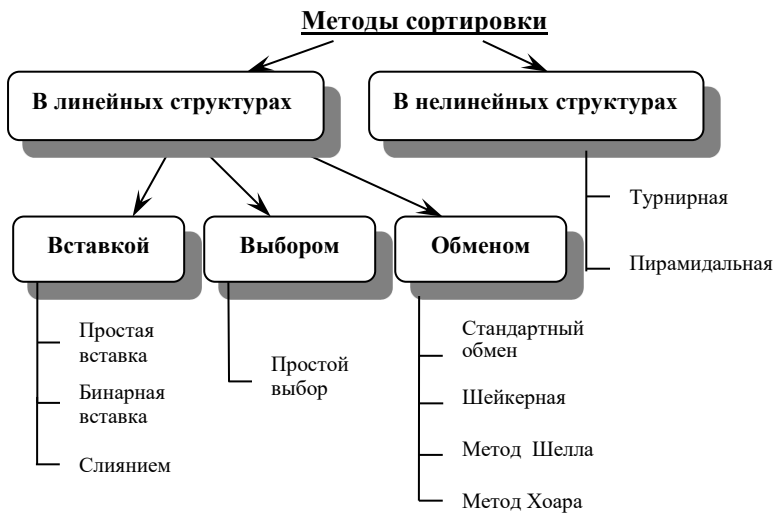


Рис.1. Классификация методов сортировки

Обычно сортируемые элементы множества называют записями и обозначают  $k_1, k_2, \dots, k_n$ .

### ***Сортировка в линейных структурах***

Предложены основные принципы и приемы построения алгоритмических схем обработки массивов для различного рода задач, а также механизм реализации алгоритмов на языке высокого уровня.

#### ***Сортировка выбором***

Сортировка выбором состоит в том, что сначала в неупорядоченном списке выбирается и отделяется от остальных наименьший элемент. После этого исходный список оказывается измененным. Измененный список принимается за исходный и процесс продолжается до тех пор, пока все элементы не будут выбраны. Очевидно, что выбранные элементы образуют упорядоченный список.

Например, требуется найти минимальный элемент списка:

{5, 11, 6, 4, 9, 2, 15, 7}.

Процесс выбора показан на рис.2, где в каждой строке выписаны сравниваемые пары. Выбираемые элементы с меньшим весом обозначены (min). Нетрудно видеть, что число сравнений соответствует числу строк, а число перемещений - количеству изменений выбранного элемента (см. рис.2).

Шаг 1	5 (min)		После сравнения ( $\vee$ ) элемент с большим весом помещается в стек
Шаг 2	$5 \vee 11$	5 (min)	
	<b>Стек:</b>	11	
Шаг 3	$5 \vee 6$	5 (min)	
	<b>Стек:</b>	11, 6	
Шаг 4	$5 \vee 4$	4 (min)	
	<b>Стек:</b>	11, 6, 5	
Шаг 5	$4 \vee 9$	4 (min)	
	<b>Стек:</b>	11, 6, 5, 9	
Шаг 6	$4 \vee 2$	2 (min)	
	<b>Стек:</b>	11, 6, 5, 9, 4	
Шаг 7	$2 \vee 15$	2 (min)	
	<b>Стек:</b>	11, 6, 5, 9, 4, 15	
Шаг 8	$2 \vee 7$	2 (min)	
	<b>Стек:</b>	11, 6, 5, 9, 4, 15, 7	

Рис.2. Сортировка выбором

Выбранный в исходном списке минимальный элемент размещается на предназначенном ему месте несколькими способами:

- минимальный элемент после  $i$ -го просмотра перемещается на  $i$ -е место результирующего списка ( $i = 1, 2, \dots, n$ ), а в исходном списке на место выбранного элемента записывается какое-то очень большое число, превосходящее по величине любой элемент списка, при этом длина заданного списка остается постоянной; измененный таким образом список принимается за исходный;
- минимальный элемент записывается на  $i$ -е место исходного списка ( $i = 1, 2, \dots, n$ ), а элемент с  $i$ -го места - на место выбранного; при этом очевидно, что уже упорядоченные элементы (а они будут расположены начиная с первого места) исключаются из дальнейшей сортировки, поэтому длина каждого последующего списка (списка, участвующего в каждом последующем просмотре) должна быть на один элемент меньше предыдущего.

Сложность метода сортировки выбором порядка  $O(n^2)$ .

### *Сортировка вставкой*

Метод сортировки вставкой предусматривает поочередный выбор из неупорядоченной последовательности элементов каждого элемента, сравнение его с предыдущим, уже упорядоченным, и перемещение на соответствующее место.

Сортировку вставкой рассмотрим на примере заданной неупорядоченной последовательности элементов:

{40, 11, 83, 57, 32, 21, 75, 64}.

Процедура сортировки отражена на рис.3, где на каждом этапе выделен анализируемый элемент, стрелкой сверху отмечено место перемещения анализируемого элемента, в рамку заключены упорядоченные части последовательности.

На шаге 1 сравниваются два начальных элемента. Поскольку второй элемент меньше первого, он перемещается на место первого элемента, который сдвигается вправо на одну позицию. Остальная часть последовательности остается без изменения.

На шаге 2 из неупорядоченной последовательности выбирается элемент и сравнивается с двумя упорядоченными ранее элементами. Так как он больше предыдущих, то остается на месте. Затем анализируются четвертый, пятый и последующие элементы - до тех пор, пока весь список не будет упорядоченным, что имеет место на последнем 7-м шаге.

Сложность метода сортировки вставкой порядка  $O(n^2)$ .

Сортировка вставкой			
Шаг 1	40	11	Анализируемый элемент помещается на соответствующую позицию множества
	11, 40		
Шаг 2	11, 40	83	
	11, 40, 83		
Шаг 3	11, 40, 83	57	
	11, 40, 57, 83		
Шаг 4	11, 40, 57, 83	32	
	11, 32, 40, 57, 83		
Шаг 5	11, 32, 40, 57, 83	21	
	11, 21, 32, 40, 57, 83		
Шаг 6	11, 21, 32, 40, 57, 83	75	
	11, 21, 32, 40, 57, 75, 83		
Шаг 7	11, 21, 32, 40, 57, 75, 83	64	
	11, 21, 32, 40, 57, 64, 75, 83		

Рис.3. Последовательность шагов сортировки вставкой

### Сортировка бинарной вставкой

При использовании метода сортировки вставками ключ  $i$ -го элемента сравнивается приблизительно с  $i/2$  ключами уже отсортированных элементов. Для того чтобы ускорить этот процесс, Джон Мочлив предложил использовать метод бинарных вставок, т.е. сравнивать ключ  $i$ -го элемента ( $K_i$ ) с ключом середины уже отсортированной последовательности ( $K_{i/2}$ ). Затем в зависимости от того, ключ  $K_i$  меньше или больше  $K_{i/2}$ , будем рассматривать следующие элементы:

$[K_i; K_{i/2}-1]$  или  $[K_{i/2} + 1; K_i]$ .

Для решения проблемы нечетного числа элементов множества номер центрального элемента вычисляется по формуле:

$$\text{mid} = (\text{low} + \text{high}) \text{ div } 2.$$

Поскольку процесс сравнения занимает относительно малый промежуток времени, а перемещать придется для  $i$ -го элемента в среднем  $i/2$  элементов, время, необходимое на реализацию алгоритма, все равно пропорционально  $1/4 \cdot n^2$ . Сложность метода сортировки бинарной вставкой такая же, как и у метода простой вставки.



Разновидностью сортировки вставкой является метод фон Неймана.

Алгоритм решения этой задачи, известный как **сортировка фон Неймана**, или сортировка слиянием, состоит в следующем: сначала анализируются первые элементы обоих массивов. Меньший элемент переписывается в новый массив. Оставшийся элемент последовательно сравнивается с элементами из другого массива. В новый массив после каждого сравнения попадает меньший элемент. Процесс продолжается до исчерпания элементов одного из массивов. Затем остаток другого массива дописывается в новый массив. Полученный новый массив упорядочен таким же образом, как исходный.

Пусть имеются два отсортированных в порядке возрастания массива  $p[1], p[2], \dots, p[n]$  и  $q[1], q[2], \dots, q[n]$  и имеется пустой массив  $r[1], r[2], \dots, r[2 \times n]$ , который нужно заполнить значениями массивов  $p$  и  $q$  в порядке возрастания. Для слияния выполняются следующие действия: сравниваются элементы  $p[1]$  и  $q[1]$ , и меньшее из значений записывается в элемент  $r[1]$ . Предположим, что это значение  $p[1]$ . Тогда  $p[2]$  сравнивается с  $q[1]$ , и меньшее из значений заносится в  $r[2]$ . Предположим, что это значение  $q[1]$ . Тогда на следующем шаге сравниваются значения  $p[2]$  и  $q[2]$  и т.д., пока не будут достигнуты границы одного из массивов. Затем остаток другого массива просто дописывается в «хвост» массива  $r$ .

Пусть даны два исходных множества, содержащие по четыре элемента

$$P = \{3, 5, 7, 44\}, Q = \{6, 8, 33, 555\}.$$

Необходимо получить новый упорядоченный массив, включающий все элементы первых двух множеств. Пример слияния двух массивов показан на рис.4.

### *Сортировка обменом*

В сортировке обменом элементы списка последовательно сравниваются между собой и меняются местами в том случае, если предыдущий элемент больше последующего. Требуется, например, провести сортировку списка методом стандартного обмена, или методом «пузырька»:

$$\{40, 11, 83, 57, 32, 21, 75, 64\}.$$

Обозначим квадратными скобками со стрелками  $\updownarrow$  обмениваемые элементы, а скобкой  $\sqcup$  – сравниваемые элементы. Первый этап сортировки показан на рис.5.

Нетрудно видеть, что после каждого просмотра списка все элементы, начиная с последнего, занимают свои окончательные позиции, поэтому их не требуется проверять при следующих просмотрах.

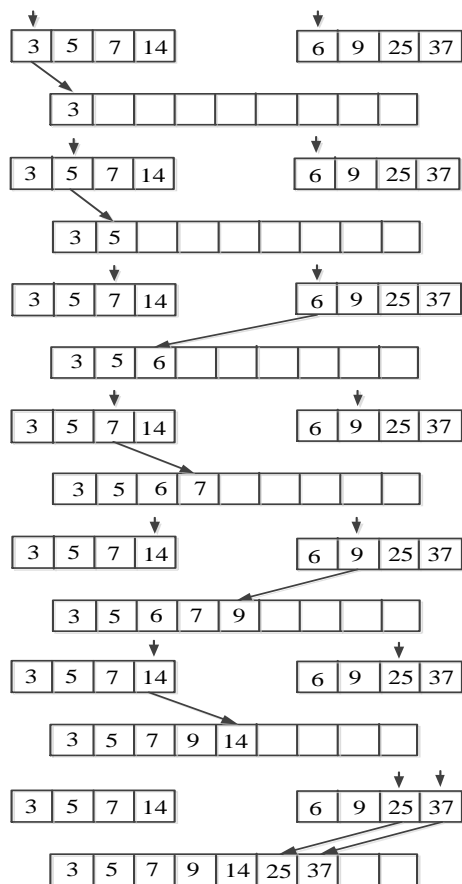


Рис.4. Сортировка слиянием (фон Неймана)

Каждый последующий этап исключает очередную позицию с найденным максимальным элементом, тем самым укорачивая список. После первого этапа в последней позиции оказался больший элемент, рав-

ный 83 (исключается из дальнейшего рассмотрения). Второй этап выявляет максимальный элемент, равный 75 (рис.6).

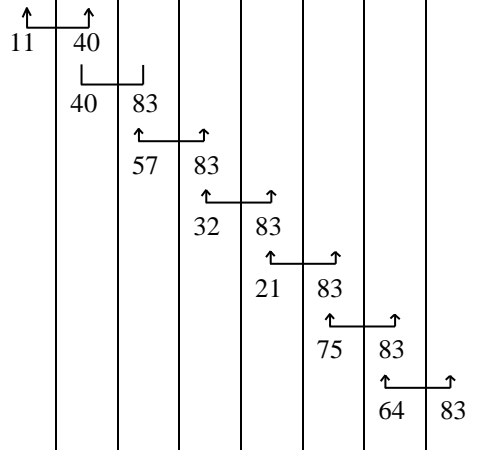
Исходное множество	40	11	83	57	32	21	75	64
Шаг $d=1$								
Полученное множество	11	40	57	32	21	75	64	83

Рис.5. Сортировка обменом (первый этап)

Процесс сортировки продолжается до тех пор, пока не будут сформированы все элементы конечного списка, либо не выполнится условие Айверсона.

**Условие Айверсона:** если в ходе сортировки при сравнении элементов не было сделано ни одной перестановки, то множество считается упорядоченным (условие Айверсона выполняется только при шаге  $d=1$ ).

Модификацией сортировки стандартным обменом является шейкерная, или челночная сортировка. Здесь, как и в методе «пузырька», проводится попарное сравнение элементов.

При этом первый этап осуществляется слева направо, второй - справа налево и т.д. Иными словами, меняется направление просмотра элементов списка.

Сложность метода стандартного обмена  $O(n^2)$ .

<i>Исходное множество</i>	11	40	57	32	21	75	64
Шаг $d=1$							
<i>Полученное множество</i>	11	40	32	21	21	64	75

Рис.6. Сортировка обменом (второй этап)

### *Шейкерная сортировка*

Очевидный прием улучшения алгоритма стандартного обмена - запоминать, были или не были перестановки в процессе некоторого прохода. Если в последнем проходе перестановок не было, то алгоритм можно заканчивать. Это улучшение, однако, можно опять же улучшить, если запоминать не только сам факт, что обмен имел место, но и положение (индекс) последнего обмена. Ясно, что все пары соседних элементов выше этого индекса  $k$  уже упорядочены. Поэтому просмотры можно заканчивать на этом индексе, а не идти до заранее определенного нижнего предела для  $i$ .

Например, массив  $\{12, 18, 42, 44, 55, 67, 94, 06\}$  с помощью усовершенствованной «пузырьковой» сортировки можно упорядочить за один просмотр, а для сортировки массива  $\{94, 06, 12, 18, 42, 44, 55, 67\}$  требуется семь просмотров. Это приводит к мысли: чередовать направление последовательных просмотров. Модификацией сортировки стандартным обменом является **шейкерная**, или челночная сортировка. Далее приведена схема шейкерной сортировки восьми ключей (рис.7).

```

L = 2  3  3  4  4
R = 8  8  7  7  4
dir  ↑  ↓  ↑  ↓  ↑
    44 06 06 06 06
    55 44 44 12 12
    12 55 12 44 18
    42 12 42 18 42
    94 42 55 42 44
    18 94 18 55 55
    06 18 67 67 67
    67 67 94 94 94

```

Рис.7. Схема шейкерной сортировки

### Сортировка методом Шелла

В методе Шелла сравниваются не соседние элементы, а элементы, расположенные на расстоянии  $d$  (где  $d$  - шаг между сравниваемыми элементами):  $d = \lfloor n/2 \rfloor$ . После каждого просмотра шаг  $d$  уменьшается вдвое. На последнем просмотре он сокращается до  $d = 1$ .

**Пример.** Дан исходный список, в котором число элементов четно:  
 $\{40, 11, 83, 57, 32, 21, 75, 64\}$ .

Первоначальный шаг вычисляется по формуле, т.е.  $d = \lfloor n/2 \rfloor = 4$ .

Исходный массив	40	11	83	57	32	21	75	64
Шаг $d = 4$	↑				↑			
	32				40			
		11				21		
			75				83	
				57				64
Полученный массив	32	11	75	57	40	21	83	64

Рис.8. Метод Шелла (шаг  $d = 4$ )

При первом просмотре сравниваются элементы, отстоящие друг от друга на  $d = 4$  (рис.8), т.е.  $k_1$  и  $k_5$ ,  $k_2$  и  $k_6$  и т.д. Если  $k_i > k_{i+d}$ , то происходит обмен между позициями  $i$  и  $(i + d)$ . Перед вторым просмотром вы-

бирается шаг  $d = \lfloor d/2 \rfloor = 2$ . Затем выбирается шаг  $d = \lfloor d/2 \rfloor = 1$  (рис.9), т.е. налицо аналогия с методом стандартного обмена.

<i>Исходный массив</i>	32	11	75	57	40	21	83	64
Шаг $d = 2$	↑ 32		↑ 75					
		└─┘ 11	↑ 40	└─┘ 57	↑ 75			
			└─┘ 21	└─┘ 75	↑ 57	└─┘ 83		
					└─┘ 57	└─┘ 83	└─┘ 64	
<i>Полученный массив</i>	32	11	40	21	75	57	83	64

Рис.9. Метод Шелла (шаг  $d = 2$ )

Сложность метода Шелла  $O(0,3n(\log_2 n)^2)$ .

### *Быстрая сортировка (сортировка Хоара)*

В методе быстрой сортировки фиксируется базовый ключ, относительно которого все элементы с большим весом перемещаются вправо, а с меньшим - влево. В качестве базового элемента обычно выбирается любой крайний элемент исходного множества, который постоянно сравнивается с противоположно стоящими элементами. При этом весь список элементов делится относительно базового ключа на две части. Для каждой части процесс повторяется.

Например, пусть дано множество  $\{40, 11, 83, 57, 32, 21, 75, 64\}$ .

На рис.10 представлен первый этап быстрой сортировки. В первой строке указана исходная последовательность элементов. Примем первый элемент (самый левый) последовательности за базовый ключ, выделим его квадратом и обозначим  $k_0 = 40$ . Установим два указателя:  $i$  и  $j$ , из которых  $i$  начинает отсчет слева ( $i = 1$ ), а  $j$  - справа ( $j = n$ ).

Сравниваем базовый ключ  $k_0$  и текущий ключ  $k_j$ . Если  $k_0 \leq k_j$ , то устанавливаем  $j = j - 1$  и еще раз сравниваем  $k_0$  и  $k_j$ . Продолжаем уменьшать  $j$  до тех пор, пока не достигнем условия  $k_0 > k_j$ . После этого меняем местами ключи  $k_0$  и  $k_j$  (см. шаг 3 на рис.10). Теперь начинаем изменять индекс  $i = i + 1$  и сравнивать элементы  $k_i$  и  $k_0$ .

Номер шага	$i \longrightarrow$				$\longleftarrow j$			
	40	11	83	57	32	21	75	64
1	40							64
2	40						75	
3	40						21	
	21						40	
4		11					40	
5			83				40	
			40				83	
6			40				32	
			32				40	
7				57	40			
				40	57			
	21	11	32	40	57	83	75	64

Рис.10. Метод Хоара

Продолжаем увеличение  $i$  до тех пор, пока не получим условие  $k_i > k_0$ , после чего следует обмен  $k_i$  и  $k_0$  (см. шаг 5 на рис.10). Снова возвращаемся к индексу  $j$ , уменьшаем его. Чередую уменьшение  $j$  и увеличение  $i$ , продолжаем этот процесс с обоих концов к середине до тех пор, пока не получим  $i = j$  (см. шаг 7 на рис.10).

В отличие от предыдущих рассмотренных сортировок уже на первом этапе имеют место два факта: во-первых, базовый ключ  $k_0 = 40$  занял свое постоянное место в сортируемой последовательности, во-вторых, все элементы слева от  $k_0$  будут меньше него, а справа - больше него. Таким образом, по окончании первого этапа исходное множество разбивается на два подмножества:

21, 11, 32	<u>40</u>	57, 83, 75, 64
Левое подмножество		Правое подмножество

Указанная процедура сортировки применяется независимо к левой и правой частям.

Сложность метода Хоара  $O(n \log_2 n)$ .

## Сортировка в нелинейных структурах

Сортировка в нелинейных структурах осуществляется только на бинарных деревьях, т.е. деревьях, из каждой вершины которого выходит по два ребра.

### Турнирная сортировка

Метод турнирной сортировки основан на повторяющихся поисках наименьшего ключа среди  $n$  элементов, среди оставшихся  $(n - 1)$  элементов и т.д. Например, сделав  $n/2$  сравнений, можно определить в каждой паре ключей меньший. С помощью  $n/4$  сравнений – меньший из пары уже выбранных меньших и т.д. Прделав  $(n - 1)$  сравнений, можно построить дерево выбора и идентифицировать его корень как наименьший ключ.

Затем осуществляется спуск вдоль пути, отмеченного наименьшим элементом, и исключение его из дерева путем замены либо на пустой элемент (дырку) в самом низу, либо на элемент из соседней ветви в промежуточных вершинах. Элемент, передвинувшийся в корень дерева, вновь будет наименьшим (теперь уже вторым) ключом, и его можно исключить. После  $n$  таких шагов дерево станет пустым (т.е. в нем останутся одни дырки) и процесс сортировки заканчивается.

**Пример.** Осуществить турнирную сортировку множества  
 $\{ a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8 \}$ .

Производится попарное сравнение вершин дерева (рис.11).

$a_2 = \min(a_1, a_2)$   
 $a_3 = \min(a_3, a_4)$   
 $a_5 = \min(a_5, a_6)$   
 $a_8 = \min(a_7, a_8)$

$a_3 = \min(a_2, a_3)$   
 $a_5 = \min(a_5, a_8)$

$a_5 = \min(a_3, a_5)$

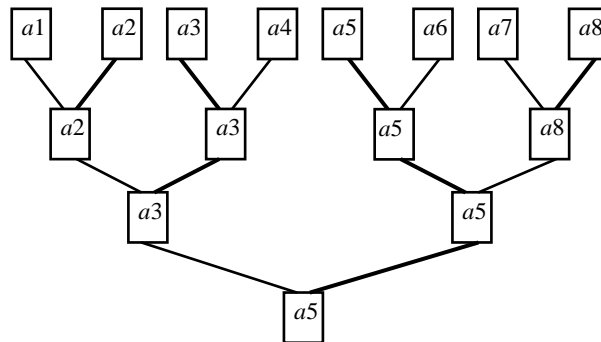


Рис.11. Первый этап турнирной сортировки



Найденный минимальный элемент заменяется на специальный символ  $M$  и помещается в результирующее множество (рис.12).

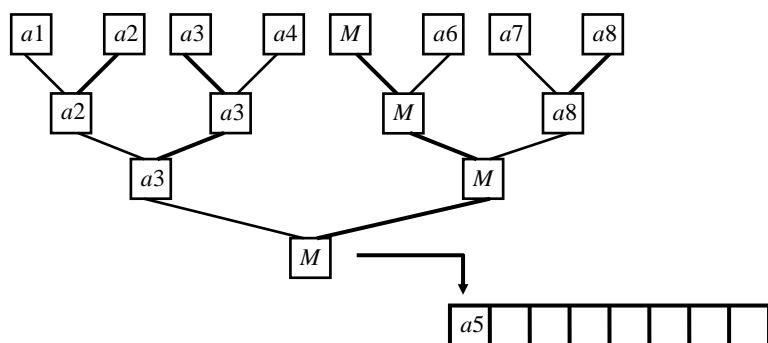


Рис.12. Удаление минимального элемента

На последующих этапах найденные минимальные элементы помещаются в результирующее множество (рис.13).

Свое название эта сортировка получила потому, что она используется при проведении соревнований, турниров и олимпиад. Элементы исходного множества представляются листьями дерева. Их попарное сравнение позволяет определить максимальный элемент.

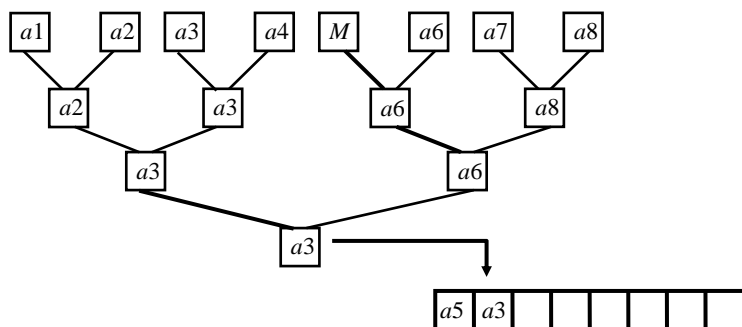


Рис.13. Этапы турнирной сортировки

### Пирамидальная сортировка

Данный тип сортировки заключается в построении пирамидального дерева. Пирамидальное дерево - это бинарное дерево, обладающее тремя свойствами:

1. В вершине каждой триады располагается элемент с бóльшим весом.
2. Листья бинарного дерева находятся либо в одном уровне, либо в двух соседних (рис.14).
3. Листья нижнего уровня располагаются левее листьев более высокого уровня, т.е. заполнение каждого уровня осуществляется слева направо.

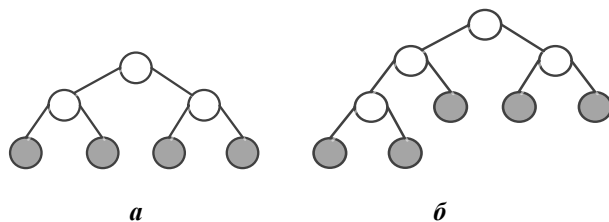


Рис.14. Бинарное дерево: *а* - листья на одном уровне; *б* - листья на соседних уровнях

В ходе преобразования элементы триад сравниваются дважды, при этом элемент с бóльшим весом перемещается вверх, а с меньшим - вниз (рис.15).

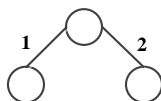


Рис.15. Порядок сравнения элементов триад

**Пример.** Дано исходное множество  $\{2, 4, 6, 3, 5, 7\}$ , которое представляется в виде бинарного дерева. Последовательно анализируются триады дерева и определяется максимальный элемент множества (рис.16), который перемещается в корень дерева.

После очередного этапа найденный максимальный элемент меняется местами с последним элементом множества. В результате сортировки получено упорядоченное множество  $\{2, 3, 4, 5, 6, 7\}$ .

### *Топологическая сортировка*

Топологическая сортировка - это одна из известных задач в программировании, в которой широко применяются списки. Под топологической сортировкой понимается сортировка элементов, для которых

определен частичный порядок, т.е. упорядочение задано не на всех, а только на некоторых парах элементов.

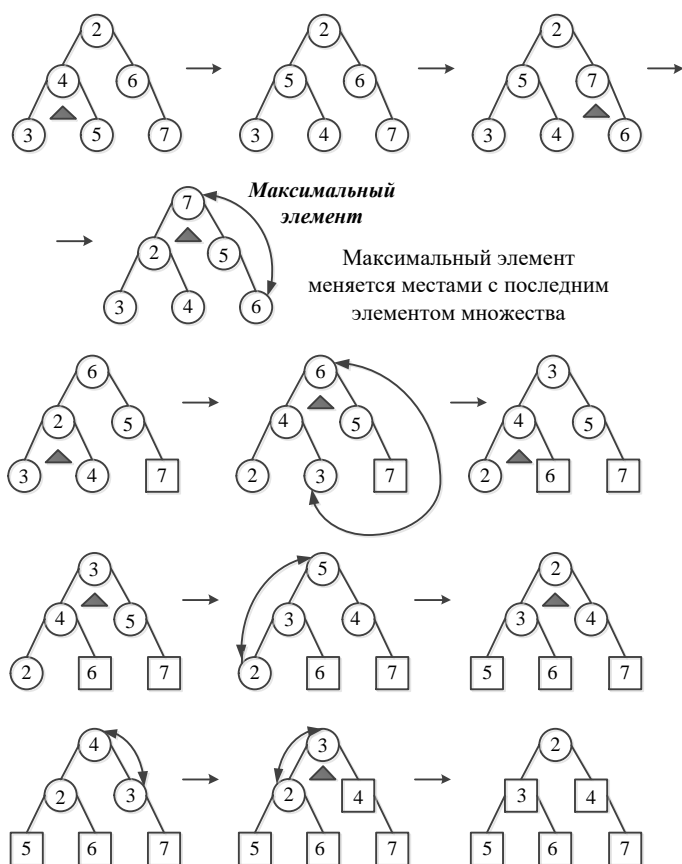


Рис.16. Пирамидальная сортировка

**Пример.** В вузовской программе изложение одних курсов включает информацию из других курсов (рис.17).

Так для того чтобы понять курс по структурам и алгоритмам обработки данных, необходимо прежде прослушать курс по конструированию программ и языкам программирования. С другой стороны, на порядок изложения перечисленных курсов никак не влияет курс по физике.

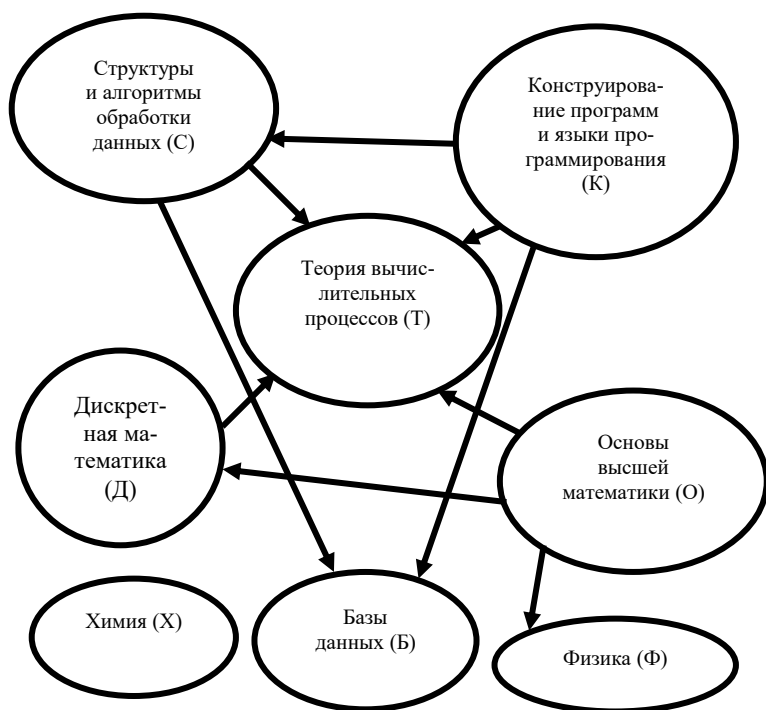


Рис.17. Межпредметные связи дисциплин

Топологическая сортировка для набора курсов означает определение такого порядка чтения курсов, при котором ни один курс не читается раньше того курса, на материале которого он основан. Частичный порядок элементов удобно представлять в графическом виде. При этом каждому курсу соответствует некоторый узел графа, а зависимость (частичный порядок) задана направленными дугами. На рис.17 приводится пример задания частичной зависимости на множестве восьми учебных дисциплин: О, Ф, К, Х, Д, С, Б, Т.

Понятно, что этот вариант не является единственным, так как реальная задача может оказаться гораздо более громоздкой. Возможны варианты, когда топологическая сортировка может вовсе не иметь решения.

Проверка найденного решения проста: в полученной последовательности достаточно провести дуги, связывающие частично упорядоченные элементы. Если все дуги будут направлены слева направо, то найденный порядок допустим. Если хотя бы одна дуга направлена в обратную сторону, то решение неверно.

**Алгоритм топологической сортировки.** Для упорядоченной пары элементов  $A \rightarrow B$  элемент  $A$  будем называть предшественником, а элемент  $B$  - преемником. Суть алгоритма заключается в следующем.

а) Для каждого элемента определяем его преемников и подсчитываем количество его предшественников. В дальнейшем работаем не с исходным графом, а с множеством элементов, каждому из которых приписан счетчик предшественников и список преемников.

б) Выбираем из построенного множества и помещаем в выходной список элементы, не имеющие предшественников (с нулевым счетчиком).

в) Каждое извлечение элемента (см. п. б) сопровождаем пересчетом количества оставшихся предшественников. Причем эта операция затрагивает только те элементы, которые являются преемниками удаляемого.

Выполнение пунктов б) и в) повторяем до тех пор, пока это возможно. Если в очередной раз п. б) не выполним, а множество непусто, то топологическая сортировка невозможна. По алгоритму для каждого элемента необходимо хранить список преемников и счетчик числа предшественников, поэтому представим множество массивом по количеству элементов. Если же множество исчерпано, то топологическая сортировка выполнена удачно и в выходном списке перечислены элементы в допустимом порядке.

У каждого ациклического ориентированного графа есть хотя бы один исток и хотя бы один сток (рис.18).

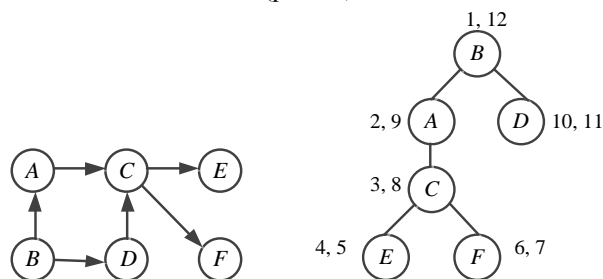


Рис. 18. Топологическая сортировка графа

Вершина называется **стоком**, если из нее не выходит ни одной дуги. Вершина графа называется **исток**, если в нее не входит ни одно ребро.

**Пример.** Выпишем связи между дисциплинами из предыдущего примера:  $K \rightarrow C$ ,  $K \rightarrow T$ ,  $K \rightarrow B$ ,  $C \rightarrow T$ ,  $C \rightarrow B$ ,  $D \rightarrow T$ ,  $O \rightarrow \Phi$ ,  $O \rightarrow D$ ,  $O \rightarrow T$ . Описанный выше массив  $A$  графически будет выглядеть так, как показано в табл.1.

**Таблица 1**

Связи между предметами

A[1]:	C	1	→	5	→	6	→	nil		
A[2]:	K	0	→	1	→	5	→	6	→	nil
A[3]:	Φ	1	→	nil						
A[4]:	O	0	→	3	→	8	→	5	→	nil
A[5]:	T	4	→	nil						
A[6]:	B	2	→	nil						
A[7]:	X	0	→	nil						
A[8]:	D	1	→	5	→	nil				

Первый числовой столбец (1, 0, 1, 0, 4, 2, 0, 1) – счетчики числа предшественников. Все последующие числовые значения задают номера элементов, являющихся преемниками данного.

В начальном состоянии имеются три элемента: 2, 4, 7 - с нулевыми счетчиками. Свяжем их в отдельный список. Через  $Q$  будем обозначать указатель на начальный элемент списка. Через  $R$  обозначим последний элемент. Таким образом, информация об исходном частично упорядоченном множестве примет вид:  $Q = 2, R = 7$  (табл.2).

**Таблица 2**

Алгоритм топологической сортировки

A[1]:	C	1	→	5	→	6	→	nil		
A[2]:	K	4	→	1	→	5	→	6	→	nil
A[3]:	Φ	1	→	nil						
A[4]:	O	7	→	3	→	8	→	5	→	nil
A[5]:	T	4	→	nil						
A[6]:	B	2	→	nil						
A[7]:	X	0	→	nil						
A[8]:	D	1	→	5	→	nil				

Для связывания элементов списка  $Q$  будем использовать поле  $N$ . Элемент  $A[Q].N$  – это уже не счетчик, а указатель на следующий элемент списка  $Q$ . Соответственно,  $A[R].N$  принимает нулевое значение, которое можно трактовать как пустую ссылку. Случай, когда список  $Q$  пуст задается значениями  $Q = 0, R = 0$ .

Укрупненный алгоритм топологической сортировки выглядит следующим образом:

- п. 1. Инициализация массива  $A$ .
- п. 2. Ввод исходных данных вида  $M \rightarrow K$ .
- п. 3. Подсчет количества предшественников и формирование списков преемников.
- п. 4. Организация списка  $Q$  и задание его хвоста  $R$ .
- п. 5.  $L := 0$ . { Количество элементов, выведенных на печать }
- п. 6. Если  $Q = 0$ , то переход к п. 13.
- п. 7. Извлечение и печать первого элемента списка  $Q$ .
- п. 8.  $L := L + 1$ .
- п. 9. Если  $L = \max$ , то переход к п. 14.
- п. 10. Просмотр преемников элемента с номером  $Q$ , пересчет числа предшественников этих элементов. Если у какого-либо элемента  $P$  число предшественников стало равным нулю, то включить данный элемент в список  $Q$  с конца:  $A[R].data.N := P; R := P$ .
- п. 11. Перестроение начала списка:  $Q := A[Q].data.N$ .
- п. 12. Переход к п. 6.
- п. 13. Вывод: «Полное решение не существует».
- п. 14. Конец работы.

## Лабораторное задание

Для каждого из перечисленных методов сортировки провести анализ временных затрат для списков различной размерности.

Алгоритм проведения лабораторной работы следующий (папка SORT).

1. Вызвать программу **Sort.exe**, включающую в себя сортировку неупорядоченных списков в линейных и нелинейных структурах.

Путь к файлу: D:\ИПОВС\АиСД\SORT\Sort\_new.exe.

Система работает в диалоговом режиме с использованием меню. Вся необходимая информация во время работы системы отображается на экране дисплея и не требует специальных пояснений.

**Основные функции системы:**

Esc - аварийное прерывание выполнения функции.

*Методы сортировки:*

- 1 - простого обмена;
- 2 - бинарной вставки;
- 3 - простой вставки;
- 4 - челночной;
- 5 - простого выбора;
- 6 - слиянием;
- 7 - Шелла;
- 8 - Хоара;
- 9 - пирамиды.

Для перечисленных в настоящей лабораторной работе методов сортировки провести следующие исследования: выполнить сортировку массивов из  $N$  чисел. Номер варианта соответствует номеру компьютера (варианты заданий). Результаты занести в форму табл.3.

**Форма таблицы 3**

**Результаты сортировки**

Метод сортировки			
Количество элементов исходного массива $N$			
Время сортировки $t$ , с			

2. Провести исследование методов сортировки упорядоченных списков с использованием программы **Winsort.exe**.

Путь к файлу: D:\ИПОВС\АиСД\SORT\Winsort.exe.

Результаты занести в форму табл.3.

3. Составить программу одного из методов сортировки в соответствии с вариантом (номером компьютера).

4. Оценить сложность рассмотренных методов сортировки:

а) провести анализ отклонения полученной в результате эксперимента сложности алгоритма от теоретической;

б) построить графические зависимости времени сортировки от количества элементов сортируемого массива.

5. Выполнить топологическую сортировку на небольшом примере (8-10 элементов).



6. Составить программу топологической сортировки. Предусмотреть возможность задания висячих вершин и кратных связей. Формат ввода выбрать самостоятельно.

7. Предусмотреть в программе учет времени сортировки для указанных значений  $n_1, n_2, n_3$ .

8. Для сортировок Шелла, Хоара, пирамидальной значения  $n_1, n_2, n_3$  удвоить. Составить программу, реализующую один из методов сортировки.

9. Рассчитать функцию сложности разработанного алгоритма по полученным временным затратам.

### Варианты заданий

Вариант	$n_1$	$n_2$	$n_3$	Составить программу
1; 15	1000	4000	6000	Простая вставка
2; 16	800	3000	7000	Сортировка слиянием
3; 17	2000	5000	6800	Метод Шелла
4; 18	1500	3500	6000	Простой выбор
5; 19	1000	2800	8500	Метод Хоара
6; 20	500	2500	6500	Бинарная вставка
7; 21	900	3000	8500	Шейкерная сортировка
8; 22	1200	2000	7500	Простая вставка
9; 23	2500	3500	6500	Шейкерная сортировка
10; 24	1600	2800	8800	Метод Шелла
11; 25	2200	3400	7200	Простой выбор
12; 26	1900	2700	8000	Метод Хоара
13; 27	1300	3500	6000	Бинарная вставка
14; 28	1700	2800	7500	Сортировка слиянием

### Пояснения к выполнению работы

Современные процессоры выполняют сортировку небольших массивов за время, значительно меньшее, чем тысячная доля секунды ( $t_{1.ti\_hund}$ ), поэтому для получения точного результата нужно выполнить сортировку достаточно большое число раз. При этом необходимо каждый раз сортировать один и тот же неупорядоченный массив. Лучше использовать второй массив, каждый раз перед сортировкой копируя в него данные из исходного неупорядоченного массива. Копирование за-

нимает определенное время, которое необходимо учесть. После завершения всего процесса следует проверить правильность сортировки, вывести содержимое второго массива на экран монитора.

Итоговая программа в обобщенном виде может выглядеть следующим образом:

```
#include <dos.h>
#include <iostream.h>
const int N = 5; // Размер массива.
const unsigned long NN = 1000000; // Число сортировок.
void main()
{
    char arr1[N]={ /* Элементы неупорядоченного массива */ };
    char arr[N];
    time t1,t2;
    double t_copy,t_sort;

    gettimeofday(&t1);
    // Копирование из массива arr1 в массив arr.
    gettimeofday(&t2);
    t_copy = (t2.ti_hour*360000.+t2.ti_min*6000.+
        t2.ti_sec*100.+t2.ti_hund-(t1.ti_hour*360000.+
        t1.ti_min*6000.+t1.ti_sec*100.+t1.ti_hund))/100.;
    gettimeofday(&t1);
    // Цикл сортировки из большого числа проходов.
    gettimeofday(&t2);
    t_sort = (t2.ti_hour*360000.+t2.ti_min*6000.+
        t2.ti_sec*100.+t2.ti_hund-(t1.ti_hour*360000.+
        t1.ti_min*6000.+t1.ti_sec*100.+t1.ti_hund))/100.;
    t_sort=t_copy;
    // Контроль содержимого массива arr после сортировки.
    // Вывод на экран времени сортировки t_sort.
}
```

## Контрольные вопросы

1. Что понимается под сортировкой?
2. Каковы особенности сортировки: вставкой, выбором, обменом, Шелла, Хоара, турнирной, пирамидой?
3. Что включает в себя понятие сложности алгоритма?

4. В чем состоит методика анализа сложности алгоритмов сортировки?

5. Дайте определение понятия «линейная динамическая структура данных». Приведите примеры таких структур. Назовите их характерные особенности.

6. Сформулируйте принцип топологической сортировки, приведите примеры. Поясните необходимость использования тех или иных структур данных и выбранный метод реализации.

## Лабораторная работа № 2

### Методы поиска

**Цель работы:** ознакомление с алгоритмами поиска в линейных и нелинейных структурах и оценкой эффективности алгоритмов.

**Продолжительность работы:** 2 часа.

### Теоретические сведения

Предметы (объекты), составляющие множество, называются его элементами. Элемент множества называется ключом и обозначается латинской буквой  $k$  с индексом, указывающим номер элемента. Алгоритмы поиска можно разбить на группы (рис.1).



Рис.1. Классификация методов поиска

Задача поиска заключается в следующем. Пусть дано множество ключей  $\{k_1, k_2, k_3, \dots, k_n\}$ . Необходимо отыскать во множестве ключ  $k_i$ . Поиск может быть завершён в двух случаях:

- 1) ключ во множестве отсутствует;
- 2) ключ найден во множестве.

## **Методы поиска, основанные на сравнении ключей**

### **Последовательный поиск**

В последовательном поиске исходное множество не упорядоченное, т.е. имеется произвольный набор ключей  $\{k_1, k_2, k_3, \dots, k_n\}$ . Метод заключается в том, что отыскиваемый ключ  $K$  последовательно сравнивается со всеми элементами множества. При этом поиск заканчивается досрочно, если ключ найден.

### **Бинарный поиск**

В бинарном поиске исходное множество должно быть упорядоченным по возрастанию. Иными словами, каждый последующий ключ больше предыдущего:

$$\{k_1 \leq k_2 \leq k_3 \leq k_4, \dots, k_{n-1} \leq k_n\}.$$

Отыскиваемый ключ сравнивается с центральным элементом множества. Если он меньше центрального, то поиск продолжается в левом подмножестве, в противном случае - в правом.

Центральный элемент находится по формуле:

$$N_{\text{эл}} = [n/2] + 1,$$

где квадратные скобки обозначают, что от деления берется только целая часть (дробная часть не учитывается). В методе бинарного поиска анализируются только центральные элементы.

**Пример.** Дано исходное множество ключей:

$\{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198\}$ .

Найти во множестве ключ  $K = 61$ .

**Шаг 1.**

$$N_{\text{эл}} = [n/2] + 1 = [22/2] + 1 = 12.$$

$$K \sim k_{12}.$$

Символ « $\sim$ » обозначает сравнение элементов (чисел, значений).

$61 > 60$ . Дальнейший поиск в правом подмножестве

$$\{61, 69, 75, 79, 80, 81, 95, 101, 123, 198\}.$$

**Шаг 2.**

$$N_{\text{эл}} = [n/2] + 1 = [12/2] + 1 = 7.$$

$$K \sim k_{12+7}.$$

$$K \sim k_{19}.$$

$61 < 95$ . Дальнейший поиск в левом подмножестве

$$\{61, 69, 75, 79, 80, 81\}$$

(относительно предыдущего подмножества).

**Шаг 3.**

$$N_{\text{эл}} = \lfloor n/2 \rfloor + 1 = \lfloor 6/2 \rfloor + 1 = 4.$$

$$K \sim k_{16}.$$

$61 < 79$ . Дальнейший поиск в левом подмножестве  $\{61, 69, 75, 79\}$ .

**Шаг 4.**

$$N_{\text{эл}} = \lfloor n/2 \rfloor + 1 = \lfloor 4/2 \rfloor + 1 = 3.$$

$$K \sim k_{15}.$$

$61 < 75$ . Дальнейший поиск в левом подмножестве  $\{61, 69\}$ .

**Шаг 5.**

$$N_{\text{эл}} = \lfloor n/2 \rfloor + 1 = \lfloor 2/2 \rfloor + 1 = 2.$$

$$K \sim k_{14}.$$

$61 < 69$ . Дальнейший поиск в левом подмножестве  $\{61\}$ .

**Шаг 6.**

$$K \sim k_{13}.$$

$$61 = 61.$$

Вывод: искомый ключ найден под номером 13.

### *Фибоначчиев поиск*

В этом поиске анализируются элементы, находящиеся в позициях, равных числам Фибоначчи. Числа Фибоначчи получаются по следующему правилу: каждое последующее число равно сумме двух предыдущих чисел. Например:

$$\{1, 2, 3, 5, 8, 13, 21, 34, 55, \dots\}.$$

Поиск продолжается до тех пор, пока не будет найден интервал между двумя ключами, где может располагаться отыскиваемый ключ.

**Пример.** Дано исходное множество ключей:

$$\{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52\}.$$

Пусть отыскиваемый ключ равен 42 ( $K = 42$ ).

Последовательное сравнение отыскиваемого ключа будет проводиться в позициях, равных числам Фибоначчи:  $\{1, 2, 3, 5, 8, 13, 21, \dots\}$ .

**Шаг 1.**  $K \sim k_1$ .  $42 > 3 \Rightarrow$  отыскиваемый ключ сравнивается с ключом, стоящим в позиции, равной числу Фибоначчи.

**Шаг 2.**  $K \sim k_2$ .  $42 > 5 \Rightarrow$  сравнение продолжается.

**Шаг 3.**  $K \sim k_3$ .  $42 > 8 \Rightarrow$  сравнение продолжается.

**Шаг 4.**  $K \sim k_5$ .  $42 > 11 \Rightarrow$  сравнение продолжается.

**Шаг 5.**  $K \sim k_8$ .  $42 > 19 \Rightarrow$  сравнение продолжается.

**Шаг 6.**  $K \sim k_{13}$ .  $42 > 35 \Rightarrow$  сравнение продолжается.

**Шаг 7.**  $K \sim k_{18}$ .  $42 < 52 \Rightarrow$  найден интервал, в котором находится отыскиваемый ключ: от 13-й до 18-й позиции, т.е.

$\{35, 37, 42, 45, 48, 52\}$ .

В найденном интервале поиск вновь ведется в позициях, равных числам Фибоначчи.

### Интерполяционный поиск

Исходное множество должно быть упорядоченно по возрастанию весов. Первоначальное сравнение осуществляется на расстоянии шага  $d$ , который определяется по формуле:

$$d = \left\lfloor \frac{(j-i)(K - K_i)}{K_j - K_i} \right\rfloor,$$

где  $j$  - номер последнего рассматриваемого элемента;  $i$  - номер первого рассматриваемого элемента;  $K$  - отыскиваемый ключ;  $K_i, K_j$  - значения ключей в  $i$ -й и  $j$ -й позициях;  $\lfloor \rfloor$  - целая часть числа.

Идея метода заключается в следующем: шаг  $d$  меняется после каждого этапа по формуле, приведенной выше. Алгоритм заканчивает работу при  $d = 0$ . В этом случае осуществляется анализ соседних элементов, после чего делается окончательный вывод о результатах поиска (рис.2).

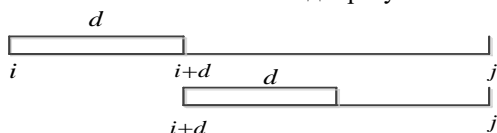


Рис.2. Интерполяционный поиск

Этот метод прекрасно работает, если исходное множество представляет собой арифметическую прогрессию или множество, приближенное к ней.

**Пример.** Дано исходное множество ключей:

$\{2, 9, 10, 12, 20, 24, 28, 30, 37, 40, 45, 50, 51, 60, 65, 70, 74, 76\}$ .

Найти во множестве ключ, равный 70 ( $K = 70$ ).

**Решение.** Определяется шаг  $d$  для исходного множества ключей:

$$d = \lfloor (18 - 1)(70 - 2) / (76 - 2) \rfloor = 15.$$

Сравнивается ключ, стоящий под шестнадцатым порядковым номером в данном множестве, с искомым ключом:

$K_{1+15} \sim K, \quad k_{16} \sim K, \quad 70 = 70.$  Ключ найден.

### Поиск по бинарному дереву

Использование структуры бинарного дерева позволяет быстро вставлять и удалять записи и проводить эффективный поиск по таблице. Такая гибкость достигается добавлением в каждую запись двух полей для хранения ссылок.

Пусть дано бинарное дерево (рис.3).

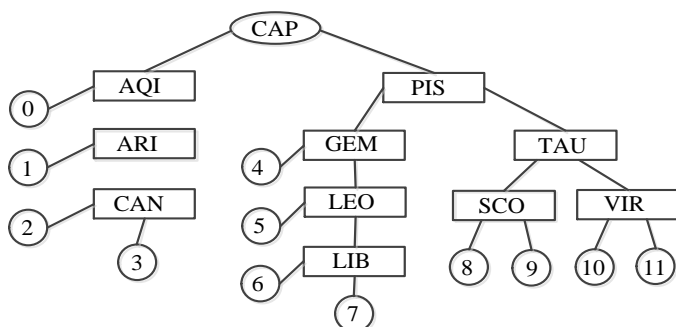


Рис.3. Бинарное дерево

Требуется по бинарному дереву отыскать ключ SAG. При просмотре от корня дерева видно, что по первой букве латинского алфавита название SAG больше чем CAP. Следовательно, дальнейший поиск будем осуществлять в правой ветви. Это слово больше, чем PIS - снова идем вправо; оно меньше, чем TAU - идем влево; оно меньше, чем SCO, и попадаем в узел 8. Таким образом, название SAG должно находиться в узле 8.

Бинарное дерево является сбалансированным, если высота левого поддерева каждого узла отличается от высоты правого не более чем на  $\pm 1$  (рис.4).



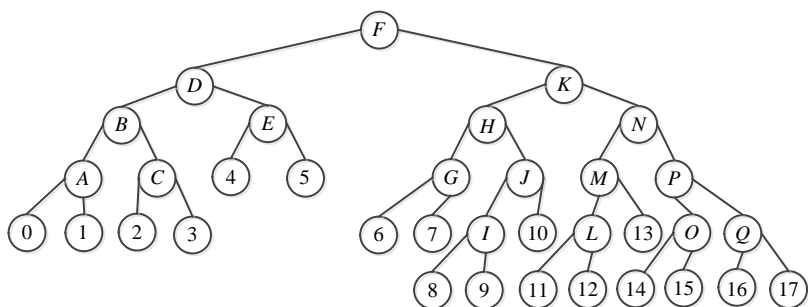


Рис.4. Сбалансированное бинарное дерево

Сбалансированные бинарные деревья занимают промежуточное положение между оптимальными бинарными деревьями (все внешние узлы которых расположены на двух смежных уровнях) и произвольными бинарными деревьями.

Рассмотрим структуру узлов сбалансированного бинарного дерева (рис.5), где В - показатель сбалансированности узла, т.е. разность высот правого и левого поддеревьев ( $B = +1; 0; -1$ ).

Ключ	Указатель на левое поддерево	Указатель на правое поддерево	Показатель сбалан- сированности узла
KEY	LLINK	RLINK	B

Рис.5. Структура узлов сбалансированного дерева

При восстановлении баланса дерева по высоте учитывается показатель В (рис.6).

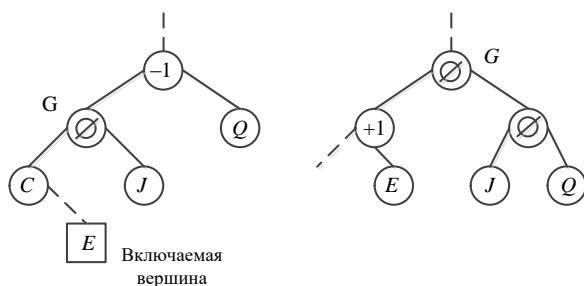


Рис.6. Учет показателей сбалансированности

Символы  $+1$ ,  $\emptyset$ ,  $-1$  указывают, что левое поддереву выше правого, поддеревья равны по высоте, правое поддереву выше левого.

**Основные операции** при работе с деревьями:

а) поиск элемента в дереве. Операция заключается в прохождении узлов дерева в одном из рассмотренных выше порядке. При прохождении дерева поиска достаточно пройти только то поддереву, которое возможно содержит искомый элемент;

б) включение элемента в дерево. Операция заключается в добавлении листа (исключая сбалансированное дерево - включение элемента в сбалансированное дерево приводит, как правило, к его перестройке) в какое-то поддереву, если такого элемента нет в исходном дереве. При включении нового элемента в дерево поиска лист добавляется в то поддерево, в котором не нарушается отношение порядка;

в) удаление элемента из дерева. Операция заключается в изменении связей между дочерними и родительскими, по отношению к удаляемому, элементами (исключая сбалансированное дерево - удаление элемента из сбалансированного дерева приводит, как правило, к его перестройке); здесь необходимо рассматривать три случая: удаление листа (рис.7,а), удаление элемента с одним потомком (рис.7,б), удаление элемента с двумя потомками (рис.7,в).

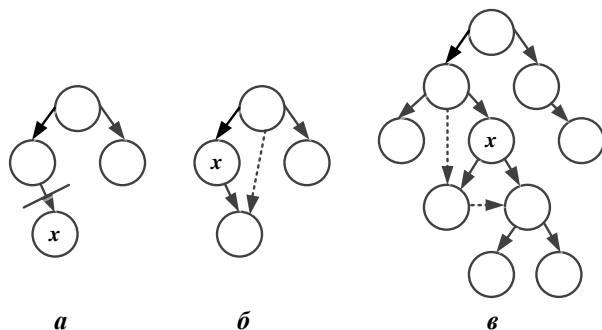


Рис.7. Удаление элемента из дерева

При удалении элемента степени 2 из дерева поиска изменять связи необходимо так, чтобы не нарушалось установленное в дереве отношение порядка. В этом случае лучшим вариантом будет перемещение на место удаляемого элемента либо самого правого листа из левого поддерева удаляемого элемента, либо самого левого листа из правого поддерева удаляемого элемента;

г) сравнение деревьев (проверка деревьев на равенство). Операция заключается в прохождении обоих деревьев в одном порядке до тех пор, пока либо не будут пройдены оба дерева, либо одно из них, либо соответствующие элементы окажутся не равными, либо в одном из деревьев не будет обнаружено отсутствие соответствующего элемента. Только в случае равенства деревьев оба дерева будут пройдены одновременно;

д) копирование дерева. Операция заключается в прохождении исходного дерева и построении нового дерева с элементами, информационные поля которых равны информационным полям соответствующих элементов исходного дерева.

**Ввод дерева.** Для того чтобы ввести дерево, надо выбрать способ перечисления его узлов. Одним из возможных способов является так называемая списочная запись (представление). Список - это конечная последовательность атомов либо списков, число которых может быть и равно нулю (атом - это понятие, определяющее элемент из любого множества предметов). Используя скобки, список можно задать перечислением через запятую его атомов (порядок перечисления атомов определяет их упорядочение). Таким образом, дерево можно записать в виде списка.

Ввод дерева, представленного списком, можно описать рекурсивной функцией (тип дерева `btree` описан выше, здесь `type` - тип информационного поля элемента - `char`):

```
btree * build_tree ( )
{ char sym;
  btree *d;
  scanf ("%c", &sym );
  switch ( sym )
  { case '(': { d = new btree;
    scanf ("%c", &sym ); d->elem = sym;
    d->left = build_tree ( );
    d->right = build_tree ( ); scanf( "%c", &sym );
    return d ; }
    case '0' : return NULL ;
    case ',' : d = build_tree ( ); break ;
  }
}
```

## Методы поиска, основанные на числовых свойствах ключей

### Поиск по бору

Особую группу методов поиска образует представление ключей в виде последовательности цифр и букв. Рассмотрим, например, имеющиеся во многих словарях буквенные высечки (гнезда). Тогда по первой букве данного слова можно отыскать страницы, содержащие все слова, начинающиеся с этой буквы. Развивая идею побуквенных высечек, получим схему поиска, основанную на индексации в структуре бора (термин использует часть слова «выборка»).

Бор имеет вид  $m$ -арного дерева. Каждый узел уровня  $h$  представляет множество всех ключей, начинающихся с определенной последовательности из  $h$  литер. Узел определяет  $m$ -путевое разветвление в зависимости от  $(h + 1)$ -й литеры.

Бор обычно представляют формой таблицы следующего вида (табл.1).

Таблица 1

Форма таблицы бора

Символы	Узлы				
	1	2	3	...	$N$
Пробел ( )					

Пробел ( ) - обязательный символ таблицы.

В первом узле записывается первая буква (цифра) ключа. Во втором узле к ней добавляется еще один символ и т.д. Если слово, начинающееся с определенной буквы (цифры), единственное, то оно сразу записывается в первом узле.

**Пример.** Дано исходное множество

{A, AA, AB, ABC, ABCD, ABCA, ABCC, BOR, C, CC, CCC, CCCD, CCCB, CCCA}.

От исходного множества перейдем к построению бора (табл.2).

Исходный алфавит = {A, B, C, D}. BOR - единственное слово на букву B, и оно посимвольно не разбивается.

Узлы бора представляют собой векторы, каждая компонента которых представляет собой либо ключ, либо ссылку (возможно пустую). Узел 1 - корень, и первую букву следует искать здесь. Если первой ока-

залась, например, буква В, то из табл.2 видно, что ей соответствует слово BOR.

**Таблица 2**

**Поиск по бору**

Символы	Узлы						
	1	2	3	4	5	6	7
_		A_	AB_	ABC_	C_	CC_	CCC_
A	2	AA		ABCA			CCCA
B	BOR	3					CCCB
C	5		4	ABCC	6	7	
D				ABCD			CCCD

Если же первая буква А, то первый узел передает управление к узлу 2, где аналогичным образом отыскивается вторая буква. Узел 2 указывает, что вторыми буквами будут \_, А, В и т.д.

**Поиск хешированием**

В основе поиска лежит переход от исходного множества к множеству хеш-функций  $h(k)$ . Хеш-функция, основанная на делении, имеет следующий вид:

$$h(k) = k \bmod (m),$$

где  $k$  - ключ;  $\bmod$  - целочисленный остаток от деления;  $m$  - целое число.

**Пример.** Дано исходное множество {9, 1, 4, 10, 8, 5}.

Определим для него хеш-функцию  $h(k) = k \bmod (m)$ :

- пусть  $m = 1$ , тогда  $h(k) = \{0, 0, 0, 0, 0, 0\}$ ;
- пусть  $m = 20$ , тогда  $h(k) = \{9, 1, 4, 10, 8, 5\}$ ;
- пусть  $m$  равно половине максимального значения ключа, тогда  $m = \lfloor 10/2 \rfloor = 5$ ;  $h(k) = \{4, 1, 4, 0, 3, 0\}$ .

Хеш-функция указывает адрес, по которому следует отыскивать ключ. Для разных ключей хеш-функция может принимать одинаковые значения, такая ситуация называется коллизией. Таким образом, поиск хешированием заключается в устранении (разрешении) коллизий (рис.8).

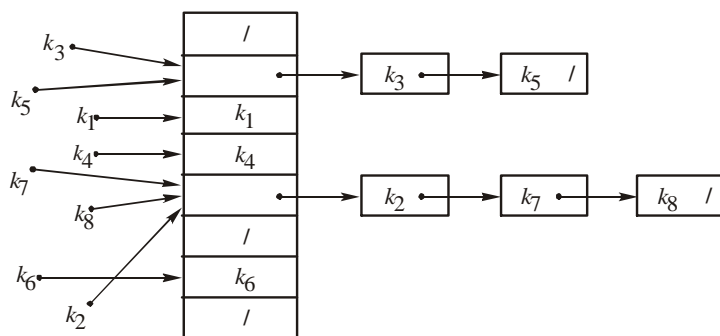


Рис.8. Разрешение коллизий

**Пример.** Дано исходное множество  $\{7, 13, 6, 3, 9, 4, 8, 5\}$ .

Найти ключ  $K = 27$ , используя поиск хешированием. Хеш-функция равна  $h(k) = K \bmod (m)$ ;  $m = \lceil 13/2 \rceil = 6$  (так как 13 - максимальный ключ. Тогда  $h(k) = \{1, 1, 0, 3, 3, 4, 2, 5\}$ .

Таблица 3

Разрешение коллизий  
табличным методом

$h(k)$	Цепочки ключей
0	6
1	7, 12
2	8
3	3, 9
4	4
5	5

Для устранения коллизий построим табл.3.

Попарным сравнениям множества хеш-функций и множества исходных ключей заполняем таблицу. Следует напомнить, что хеш-функция указывает адрес, по которому следует отыскивать ключ.

Например, если отыскивается ключ  $K = 27$ , то

$$h(k) = 27 \bmod 6 = 3.$$

Это значит, что ключ  $K = 27$  может быть только в третьей строке. Так как его там нет, то данный ключ отсутствует в исходном множестве.

### Алгоритмы поиска словесной информации

В настоящее время наличие сверхпроизводительных микропроцессоров и дешевизна электронных компонентов позволяют делать значительные успехи в алгоритмическом моделировании. Рассмотрим несколько алгоритмов обработки слов.

### Алгоритм Кнута - Морриса - Пратта (КМП)

Данный алгоритм получает на вход слово

$$X = x[1]x[2] \dots x[n]$$

и просматривает его слева направо буква за буквой, заполняя при этом массив натуральных чисел  $l[1] \dots l[n]$ , где  $l[i]$  - длина слова  $l(x[1] \dots x[i])$ .

Таким образом,  $l[i]$  есть длина наибольшего начала слова  $x[1] \dots x[i]$ , одновременно являющегося его концом.

**Задание.** Используя алгоритм КМП, определить, является ли слово  $A$  подсловом слова  $B$ ?

**Решение.** Применим алгоритм КМП к слову  $A\#B$ , где  $\#$  - специальный символ, не встречающийся ни в слове  $A$ , ни в слове  $B$ . Слово  $A$  является подсловом слова  $B$  тогда и только тогда, когда среди чисел в массиве  $l$  будет число, равное длине слова  $A$ .

Предположим, что первые  $i$  значений  $l[1] \dots l[i]$  уже найдены. Читается очередная буква слова, т.е.  $x[i+1]$  и вычисляется  $l[i+1]$ .

Другими словами, необходимо определить начало  $Z$  слова  $x[1] \dots x[i+1]$ , одновременно являющиеся его концами, - из них следует выбрать самое длинное (рис.9).

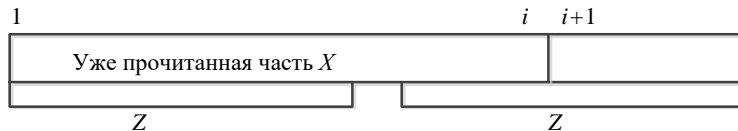


Рис.9. Анализ слова

Получаем следующий способ отыскания слова  $Z$ .

Рассмотрим все начала слова  $x[1] \dots x[i]$ , являющиеся одновременно его концами. Выберем из них подходящие - те, за которыми следует буква  $x[i+1]$ . Из подходящих выберем самое длинное. Приписав в его конец  $x[i+1]$ , получим искомое слово  $Z$ . Теперь воспользуемся сделанными приготовлениями и вспомним, что все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции  $l$ .

Получим следующий фрагмент программы.

```
i:=1; l[1]:=0;  
  {таблица l[1]..l[i] заполнена правильно}
```

```

while i <> n do begin
  len:= l[i]
  {len - длина начала слова x[1]..x[i], которое является
его концом; все более длинные начала оказались
неподходящими}
  while (x[len+1]<>x[i+1]) and (len>0) do begin
    {начало не подходит, применяем к нему функцию l}
    len:=l[len];
  end;
  {нашли подходящее слово или убедились в его отсутствии}
  if x[len+1]=x[i+1] do begin
    {x[1]..x[len] - самое длинное подходящее начало}
    l[i+1]:=len+1;
  end else begin
    {подходящих нет}
    l[i+1]:= 0;
  end;
  i:=i+1;
end;

```

Запишем алгоритм, проверяющий, является ли слово  $X = x[1] \dots x[n]$  подсловом слова  $Y = y[1] \dots y[m]$ .

*Решение.* Вычисляем таблицу  $l[1] \dots l[n]$ .

```

j:=0; len:=0;
{len - длина максимального начала слова X, одновременно
являющегося концом слова y[1]..j[j]}
while (len<>n) and (j<>m) do begin
  while (x[len+1]<>y[j+1]) and (len>0) do begin
    {начало не подходит, применяем к нему функцию l}
    len:= l[len];
  end;
  {нашли подходящее слово или убедились в его отсут-
ствии}
  if x[len+1]=y[j+1] do begin
    {x[1]..x[len] - самое длинное подходящее начало}
    len:=len+1;
  end else begin
    {подходящих нет}
    len:=0;
  end;
  j:=j+1;
end;
{если len=n, слово X встретилось; иначе мы дошли до конца

```



слова  $Y$ , так и не встретив  $X$ }

### Алгоритм Бойера - Мура (БМ)

Этот алгоритм читает лишь небольшую часть всех букв слова, в котором ищется заданный образец.

Пусть, например, отыскивается образец  $abcd$ . Посмотрим на четвертую букву слова: если это буква  $e$ , то нет необходимости читать первые три буквы. (В самом деле, в образце буквы  $e$  нет, поэтому он может начаться не раньше пятой буквы.)

Рассмотрим простой вариант этого алгоритма, который не гарантирует быстрой работы во всех случаях. Пусть  $x[1] \dots x[n]$  - образец, который надо отыскать. Для каждого символа  $s$  найдем самое правое его вхождение в слово  $X$ , т.е. наибольшее  $k$ , при котором  $x[k] = s$ . Эти сведения будем хранить в массиве  $pos[s]$ ; если символ  $s$  вовсе не встречается, то можно предположить, что  $pos[s] = 0$ .

*Решение.* Принять все  $pos[s]$  равными 0.

```
for i:=1 to n do begin
  pos[x[i]]:=i;
end;
```

В процессе поиска будем хранить в переменной  $last$  номер буквы в слове, против которой стоит последняя буква образца. Вначале  $last = n$  (длина образца), затем переменная  $last$  постепенно увеличивается.

```
last:=n;
{все предыдущие положения образца уже проверены}
while last<= m do begin {слово не кончилось}
  if x[m]<>y[last] then begin {последние буквы разные}
    last:=last+(n-pos[y[last]]);
    {n - pos[y[last]] - минимальный сдвиг образца,
     при котором напротив y[last] встанет такая же
     буква в образце. Если такой буквы нет вообще,
     то сдвиг на всю длину образца}
  end else begin
    {если нынешнее положение удовлетворяет, т.е. если}
    x[i]..x[n]=y[last-n+1]..y[last],
    {то сообщить о совпадении}
    last:=last+1;
  end;
end;
```

### *Алгоритм Рабина*

Пусть в слове длиной  $m$  ищется образец длиной  $n$ . Вырежем окно размером  $n$  и будем двигать его по входному слову, проверяя, не совпадает ли слово в окошке с заданным образцом. Сравнивать по буквам долго. Вместо этого фиксируем некоторую функцию, определенную на словах длиной  $n$ . Если значения этой функции на слове в окне и на образце различны, то совпадения нет. Только если значения одинаковы, нужно проверять совпадение по буквам.

Выигрыш при таком подходе состоит в следующем. Чтобы вычислить значение функции на слове в окне, нужно прочесть все буквы этого слова. Так уж лучше сразу сравнить их с образцом. Тем не менее выигрыш возможен, так как при сдвиге окна слово не меняется полностью, а лишь добавляется буква в конце и убирается в начале. Заменим все буквы в слове и образце их номерами, представляющими собой целые числа. Тогда удобной функцией является сумма цифр. (При сдвиге окошка нужно добавить новое число и вычесть пропавшее.) Выберем некоторое число  $p$  (желательно простое) и некоторый вычет  $x$  по модулю  $p$ . Каждое слово длиной  $n$  представим как последовательность целых чисел (заменяя буквы кодами). Числа будем рассматривать как коэффициенты многочлена степени  $(n - 1)$  и вычислим значение этого многочлена по модулю  $p$  в точке  $x$ . Это будет одна из функций семейства (для каждой пары  $p$  и  $x$  получается, таким образом, своя функция). Сдвиг окна на единицу соответствует вычитанию старшего члена ( $x^{n-1}$  следует вычислить заранее), умножению на  $x$  и добавлению свободного члена. Следующее соображение говорит в пользу того, что совпадения не слишком вероятны.

Пусть число  $p$  фиксировано и к тому же простое, а  $X$  и  $Y$  - два различных слова длиной  $n$ . Им соответствуют различные многочлены (предполагаем, что коды всех букв различны - это возможно, если число  $p$  больше числа букв алфавита). Совпадение значений функции означает, что в точке  $x$  эти два различных многочлена совпадают, т.е. их разность обращается в нуль. Разность есть многочлен степени  $(n - 1)$  и имеет не более  $(n - 1)$  корней. Таким образом, если  $n$  много меньше  $p$ , то у случайного  $x$  мало шансов попасть в неудачную точку.

### **Лабораторное задание**

Для каждого из перечисленных методов поиска проанализировать временные затраты для списков различной размерности (папка POISK).

Алгоритм проведения лабораторной работы следующий.

1. Выполнить тестовые примеры поиска с использованием файлов \*.exe в папках Poisk, Poisk1, Poisk2, Poisk3.

Путь к файлам: D:\ИПОВС\АиСД\POISK

а) выполнить поиск в массиве из  $N$  чисел (номер варианта соответствует номеру компьютера). Результаты занести в форму табл.4.

**Форма таблицы 4**

**Форма заполняемой таблицы**

Метод поиска			
Количество элементов массива $N$			
Время поиска $t$ , с			

б) оценить сложность перечисленных методов поиска;

в) провести анализ отклонения полученной в результате эксперимента сложности алгоритма от теоретической;

г) построить графические зависимости времени поиска от количества элементов массива.

3. Провести исследование методов получения случайных чисел.

Путь к файлам: D:\ИПОВС\АиСД\POISK\RND

4. Составить программу с использованием алгоритмов Кнута - Морриса - Пратта, Бойера - Мура, Рабина для поиска текстовой информации (по заданию преподавателя).

5. Провести анализ методов поиска для списков различной размерности и построить зависимости времени от числа элементов исходного множества (табл.5).

**Таблица 5**

**Анализ методов поиска**

Вариант	$n_1$	$n_2$	$n_3$
1; 14	1000	4000	6000
2; 15	800	3000	7000
3; 16	2000	5000	6800
4; 17	1500	3500	6000
5; 18	1000	2800	8500
6; 19	500	2500	6500

7; 20	900	3000	8500
8; 21	1200	2000	7500
9; 22	2500	3500	6500
10; 23	1600	2800	8800
11; 24	700	3400	7200
12; 25	1900	2700	8000
13; 26	1300	3500	6000

6. Составить программу для реализации метода поиска (варианты заданий).

### Варианты заданий

Вариант	Составить программу
1; 14	Бинарный поиск
2; 15	Интерполяционный поиск
3; 16	Поиск хешированием
4; 17	Фибоначчиев поиск
5; 18	Поиск по бинарному дереву
6; 19	Поиск по бору
7; 20	Фибоначчиев поиск
8; 21	Поиск по бору
9; 22	Интерполяционный поиск
10; 23	Поиск по бору
11; 24	Поиск по бинарному дереву
12; 25	Поиск хешированием
13; 26	Бинарный поиск

### Контрольные вопросы

1. Что понимается под поиском?
2. В чем состоит методика анализа сложности алгоритмов поиска?
3. Каковы особенности поиска последовательного и бинарного?
4. Каковы особенности поиска интерполяционного и Фибоначчиевого?
5. Каковы особенности поиска по бинарному дереву?
6. Каковы особенности поиска по бору и хешированием?
7. В чем состоит методика анализа сложности алгоритмов поиска?

8. В чем особенность алгоритмов поиска словесной информации ?
9. Что такое коллизия?
10. Какой метод используется для разрешения коллизий?
11. Что определяет показатель сбалансированности узла дерева?
12. Перечислите понятие дерева, двоичного дерева, упорядоченного дерева, дерева поиска.
13. Укажите способы задания дерева.

# 1. Лабораторная работа № 3

## Итеративные и рекурсивные алгоритмы

**Цель работы:** изучить рекурсивные алгоритмы и рекурсивные структуры данных; проводить анализ итеративных и рекурсивных процедур; исследовать эффективность итеративных и рекурсивных процедур при реализации на ЭВМ.

**Продолжительность работы:** 2 часа.

### Теоретические сведения

Эффективным средством программирования для некоторого класса задач является рекурсия. С ее помощью можно решать сложные задачи численного анализа, комбинаторики, алгоритмов трансляции, операций над списковыми структурами и т.д. Программы в этом случае имеют небольшие объемы по сравнению с итерацией и требуют меньше времени на отладку.

*Под рекурсией* понимают способ задания функции через саму себя, например, способ задания факториала в виде

$$N! = (n - 1)! * n.$$

В программировании под рекурсивной процедурой (функцией) понимают способ обращения процедуры (функции) к самой себе.

*Под итерацией* понимают результат многократно повторяемой какой-либо операции, например, представление факториала в виде

$$n! = 1 * 2 * 3 \dots * n.$$

Среди широкого класса задач удобно представлять с использованием рекурсивных процедур (функций) те задачи, которые сводятся на подзадачи того же типа, но меньшей размерности.

Общая методика анализа рекурсии содержит три этапа:

1) параметризация задачи, заключающаяся в выделении различных элементов, от которых зависит решение, в частности размерности решаемой задачи. После каждого рекурсивного вызова размерность должна убывать;

2) поиск тривиального случая и его решение. Как правило, это ключевой этап в рекурсии, размерность задачи при этом часто равна нулю или единице;

3) декомпозиция общего случая, имеющая целью привести задачу к одной или нескольким задачам того же типа, но меньшей размерности.

Рассмотрим понятие итеративного и рекурсивного алгоритмов на примере вычисления факториала.

**Итеративный алгоритм.** Наиболее простой и естественной формой представления итеративного алгоритма при реализации на ЭВМ является описание его с использованием цикла.

//Листинг 1. Итеративная функция вычисления факториала

```
int fact(int n) {
    int t, answer;
    answer = 1;
    for(t=1; t<=n; t++)
        answer=answer*(t);
    return(answer);
}
```

//Листинг 2. Итеративная функция вычисления факториала

```
long Fact(int k)
{ long f; int i; for(f = 1, i = 1; i<k; i++) f*= i;
  return f;
}
```

**Рекурсивный алгоритм.** Рекурсивное представление факториала имеет вид  $n! = (n - 1)! * n$ .

Проведем анализ этого выражения согласно методике:

1) параметризация. В данном случае имеется всего один параметр  $N$  - целое число;

2) поиск тривиального случая. При  $n = 0$  или  $n = 1$  значение факториала равно 1, что соответствует выходу из рекурсии;

3) декомпозиция общего случая. Вычисление  $n!$  через меньшую размерность этой же задачи  $(n - 1)!$

Рекурсивная функция вычисления факториала представлена ниже.

//Листинг 3. Рекурсивная функция вычисления факториала

```
int factr(int n) {
    int answer;
    if(n==1) return(1);
    answer = factr(n-1)*n; // рекурсивный вызов
    return(answer);
}
```

```

}

//Листинг 4. Рекурсивная функция вычисления факториала
long Fact(int k)
{ if (k==0) return 1;
  return (k*Fact(k-1));    // рекурсивный вызов
}

```

Процедура-функция имеет следующие особенности:

- при вычислении факториала происходит обращение функции к самой себе, но с меньшим значением аргумента ( $n - 1$ ) по сравнению с первым вызовом  $n$ ;
- при вычислении факториала не используется цикл, что является существенной особенностью рекурсивного алгоритма.

В программу рекурсивного вычисления факториала можно добавить стандартную функцию определения текущего времени

GETTIME(Var Hour, Minute, Second, Sec100: WORD).

### ***Рекурсивные структуры данных***

К рекурсивным процедурам относятся структуры строчные (стек, очередь, дек) и списковые.

Список - набор элементов, расположенных в определенном порядке.

Список очередности - список, в котором последний поступающий элемент добавляется к нижней части списка. Список с использованием указателей - список, в котором каждый элемент содержит указатель на следующий элемент списка.

Однонаправленный и двунаправленный списки - линейный список, в котором все исключения и добавления происходят в любом месте списка.

Однонаправленный список отличается от двунаправленного списка только связью. Иными словами, в однонаправленном списке можно перемещаться только в одном направлении (из начала в конец), а двунаправленным - в любом (рис.1).

В однонаправленном списке структура добавления и удаления такая же, только связь между элементами односторонняя.



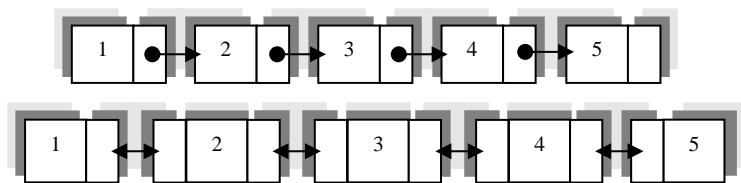


Рис.1. Однонаправленный и двунаправленный списки

**Очередь** - тип данных, при котором новые данные располагаются следом за существующими в порядке поступления; поступившие первыми данные при этом обрабатываются первыми.

Очередью называют циклическую память или список типа FIFO («first-in-first-out»: «первым включается - первым исключается»). Другими словами, у очереди есть голова и хвост (рис.2). В очереди новый элемент добавляется только с одного конца. Удаление элемента происходит на другом конце. По сути очередь - однонаправленный список, только добавление и исключение элементов происходит на концах списка (рис.2).

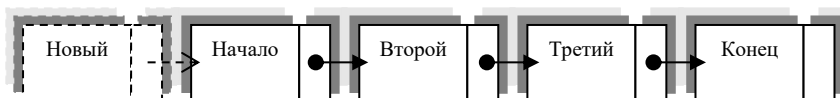


Рис.2. Пример очереди

**Стек** - линейный список, в котором все включения и исключения осуществляются в одном конце списка (рис.3). Стек называют списком, реверсивной памятью, гнездовой памятью, магазином, списком типа LIFO («last-in-first-out»: «последним включается - первым исключается»).

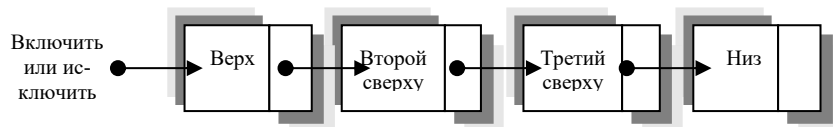


Рис.3. Стек

Стек - часть памяти ЭВМ, предназначенная для временного хранения байтов, используемых микропроцессором. Действия со стеком производятся при помощи регистра указателя стека. Любое повреждение этой части памяти приводит к сбою.

**Дек (стек с двумя концами)** - линейный список, в котором все включения и исключения делаются на обоих концах списка. Еще один термин, «архив», применяется к декам с ограниченным выходом, а деки с ограниченным входом называют перечнями или реестрами (рис.4).

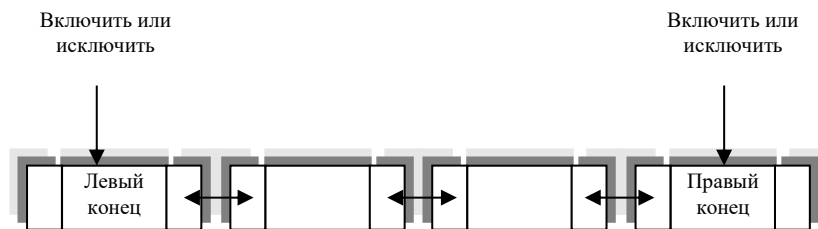


Рис.4. Дек

Рекурсивно можно представить не только алгоритм решения задачи, но и обрабатываемую информацию. Рассмотрим применение рекурсивных процедур при обработке рекурсивных структур данных.

Рассмотрим понятия списка, указателя и динамической переменной языка Турбо Паскаль.

Информационную часть можно описать как INTEGER (целый тип), REAL (действительный), CHAR (символьный) и т.д. Для отображения указателя (горизонтальной стрелки) в языке Турбо Паскаль введен особый тип данных, который называется указателем. Для его описания нет ключевого слова, вместо него используется символ  $\wedge$ . Структуру данных, рассмотренную в виде списка, можно представить на языке Турбо Паскаль следующим образом:

```
TYPE LINK =  $\wedge$ ELEMENT;
ELEMENT = RECORD;
INFORM: CHAR;
NEXT: LINK;
END;
```

Здесь LINK - указатель, указывает на ELEMENT. Структура элемента ELEMENT отражена в виде записи, состоящей из двух частей: INFORM и NEXT. Следует обратить внимание на характер структуры данных. В начале описания тип данных LINK указывает на ELEMENT, у которого, в свою очередь, одна из составляющих NEXT является типом указателя LINK, а LINK указывает на ELEMENT. Получается замкнутый круг. Такая структура данных называется рекурсивной.

В разделе определения типов допускается менять местами описание указателя и описание элемента:

```
TYPE ELEMENT = RECORD
  INFORM: CHAR;
  NEXT: LINK;
END;
LINK = ^ELEMENT;
```

Для того чтобы иметь возможность использовать в программе переменные типа ELEMENT (т.е. переменные, имеющие такую же структуру), необходимо описать их как переменные типа указателя.

Например: VAR A, B, C: LINK;

где A, B, C называются переменными-указателями, которые обозначаются в программе со стрелкой ^: A ^, B ^, C ^.

Доступ к элементу записи осуществляется с указанием составного (уточненного) имени, содержащего внутри себя символ точки.

Например:

```
A^.INFORM: = 'Z';
C^.INFORM: = B^.INFORM;
```

Каждый тип указателей среди своих возможных значений содержит значение NIL (зарезервированное слово), которое не указывает ни на один элемент. Чаще всего NIL используется при формировании начала или конца списка. Например: A^.LINK: = NIL.

В определении типа данных переменные-указатели A, B, C описаны как указатели (VAR A, B, C: LINK), а не как переменные типа ELEMENT. В этом случае при трансляции память выделяется только для указателей, а для переменных, имеющих структуру записи ELEMENT, не выделяется. В языке Турбо Паскаль имеется понятие динамической переменной, которая начинает существовать в результате вызова стандартной процедуры NEW, например NEW (A). Это означает выделение области памяти и формирование ее адреса для создания новой переменной, имеющей структуру записи. Таким образом, если в программе объявлена переменная типа указателя, то в результате вызова NEW (A) формируется переменная типа ELEMENT (состоящая из двух частей - INFORM и NEXT). Только после этого возможен доступ к элементу записи.

Например: A^.INFORM: = 'Z'; или A ^.LINK: = NIL;

Если динамическая переменная становится ненужной, то выделенная область памяти освобождается с помощью стандартной процедуры DISPOSE, например DISPOSE (A).

### ***Виды обхода бинарных деревьев***

В процедуре CREATE функция NEW(A) резервирует в памяти ЭВМ область для размещения записи типа TREE. В операторе вводится идентификатор текущего узла дерева (один символ) и заносится в поле IDENT записи, адрес которой в данный момент хранится в переменной A. Далее на экран выдается запрос, есть ли левое поддерево у данного узла. Если в ответ введен символ Y (да), то рекурсивно вызывается процедура CREATE для формирования левого поддерева. После ее завершения в переменной B возвращается адрес узла - корня левого поддерева, и он запоминается в поле LEFT текущей записи. Аналогично формируется правое поддерево. Выход из рекурсии происходит при обработке концевых вершин дерева. В записи, представляющей эти узлы, в поля LEFT и RIGHT заносится константа NIL - неопределенный адрес. Описанный алгоритм реализуется в приведенном примере рекурсивной процедурой. Условие  $A = NIL$  позволяет обнаружить концевые вершины дерева и обеспечивает выход из рекурсии.

В основной программе выполняется последовательное обращение к описанным выше подпрограммам создания и обхода бинарного дерева. Заметим, что начальное значение переменной указателя ROOT определяется в процедуре CREATE. Это значение используется для указания адреса корневого узла сформированного бинарного дерева при обращении к процедуре обхода его узлов в порядке сверху вниз.

Наиболее распространены три способа обхода узлов дерева, выше получили следующие названия (рис.5):

- 1) обход в направлении слева направо (обратный порядок, инфиксная запись);
- 2) обход сверху вниз (прямой порядок, префиксная запись);
- 3) обход снизу вверх (концевой порядок, постфиксная запись).

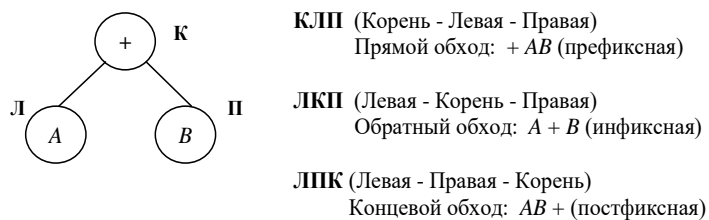


Рис.5. Три различных обхода бинарного дерева

В результате обхода дерева (рис.6), каждым из трех способов порождаются следующие последовательности прохождения узлов:

- слева направо:  $A + B * C - D$  (обратный порядок);
- сверху вниз:  $* + AB - CD$  (прямой порядок);
- снизу вверх:  $AB + CD - *$  (концевой порядок).

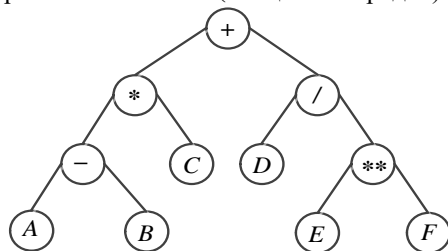


Рис.6. Бинарное дерево

Если проанализировать последовательность прохождения узлов в порядке сверху вниз, то можно установить следующее. Для любого узла, например \* (см. рис.6), сначала фиксируется факт прохождения через данный узел, затем просматриваются все узлы, входящие в его левое поддерево, и в последнюю очередь просматриваются узлы, составляющие его правое поддерево. Тогда алгоритм обхода бинарного дерева в порядке сверху вниз имеет следующий вид.

**Шаг 1.** Посетить корень дерева (напечатать его идентификатор).

**Шаг 2.** Пройти сверху вниз левое поддерево корневого узла.

**Шаг 3.** Пройти сверху вниз правое поддерево.

Рассмотрим алгоритмы поиска по дереву с включением и исключением на языке программирования C++.

Пусть имеется бинарное упорядоченное дерево, в полях элементов которого ключи – вещественные числа, а поля записей отсутствуют.

Функция поиска со вставкой в созданном дереве будет иметь следующий вид.

```

void Vstavka(double key) //функция, вставляющая элемент в дерево
{ Q=NULL;
  P=tree;
  while(P)
  { Q=P;
    if(key==P->k)
    { cout<<"\n В дереве уже есть такой элемент, он найден, вставлять не надо\n";
      return;
    }
  }
}
  
```

```

/*если такой элемент в дереве уже есть, то функция завершает свою работу*/
}
if(key<P->k) P=P->left;
else P=P->right;
};
V=new element;
V->k=key;
V->left=V->right=NULL;
if(key<Q->k) Q->left=V;
else Q->right=V;
}

```

Функция поиска с удалением в созданном дереве будет иметь следующий вид.

```

void Poisk_Udalenie (double key) //функция поиска с удалением
{Q=NULL;
P=tree;
{ while(P&&P->k!=key)
{ Q=P;
if(key<P->k) P=P->left;
else P=P->right;
};
if(!P) { cout<<"Ключ не найден!"; return;};
if(P->left==NULL) V=P->right;
else
{if(P->right==NULL) V=P->left;
else
{ T=P;
V=P->right;
S=V->left;
while(S)
{ T=V;
V=S;
S=V->left;
};
if(T!=P) { T->left=V->right;
V->right=P->right;
}
V->left=P->left;
}
}
}
if(Q==NULL) tree=V;
else

```

```

        if(P==Q->left) Q->left=V;
        else Q->right=V;
delete P;
}
}

```

Для создания бинарного сбалансированного дерева и его обхода используется дерево, поле записи которого содержит только ключи, являющиеся вещественными числами. Тогда элемент дерева описывается следующим образом.

```

struct element
{ double k;
  element* left;
  element* right;
};

```

Затем необходимо ввести указатель вершины дерева и рабочие указатели для реализации функции создания дерева:

```

element *tree=NULL, *P,*Q.

```

Пока дерево не создано, указатель корня нулевой (\*tree=NULL).

Ниже представлена функция создания бинарного дерева.

```

void maketree(double a) //создание дерева
{ if(!tree)
  { tree=new element;
    tree->k=a;
    tree->right=tree->left=NULL;
    P=tree;
    Q=NULL;
    return;
  };
P=Q=tree;
while (P!=NULL)
{ Q=P;
  if(a<P->k) P=P->left; else P=P->right;
};
if(a<Q->k) { Q->left=new element;
            Q->left->k=a;
            Q->left->left=Q->left->right=NULL;
          }
else { Q->right=new element;
      Q->right->k=a;
      Q->right->left=Q->right->right=NULL;
    }
}

```

```
};
}
```

Алгоритм обхода бинарного дерева слева направо будет иметь следующий вид.

```
void printtree(element* tree) //просмотр и печать дерева
{ if(tree)
{ printtree(tree->left);
  cout<<tree->k<<" ";
  printtree(tree->right);
};
}
```

## ***Параллельные методы умножения блочно-представленных матриц***

### ***Алгоритм Фокса***

Для организации параллельных вычислений при блочном представлении матриц предположим, что процессоры образуют логическую прямоугольную решетку размером  $k \times k$  (обозначим через  $p_{ij}$  процессор, располагаемый на пересечении  $i$  строки и  $j$  столбца решетки).

Основные положения параллельного метода, известного как алгоритм Фокса, состоят в следующем:

- каждый из процессоров решетки отвечает за вычисление одного блока матрицы  $C$ ;
- в ходе вычислений на каждом из процессоров  $p_{ij}$  располагаются четыре матричных блока: блок  $C_{ij}$  матрицы  $C$ , вычисляемый процессором; блок  $A_{ij}$  матрицы  $A$ , размещенный в процессоре перед началом вычислений; блоки  $A'_{ij}$  и  $B'_{ij}$  - матриц  $A$  и  $B$ , получаемые процессором в ходе выполнения вычислений.

Выполнение параллельного метода включает:

- 1) этап инициализации, на котором на каждый процессор  $p_{ij}$  передаются блоки  $A_{ij}$ ,  $B_{ij}$  и обнуляются блоки  $C_{ij}$  на всех процессорах;
- 2) этап вычислений, на каждой итерации  $l$  ( $1 \leq l \leq k$ ) которого выполняется:
  - для каждой строки  $l$  ( $1 \leq l \leq k$ ) процессорной решетки блок  $A_{ij}$  процессора  $p_{ij}$  пересылается на все процессоры той же строки  $i$ ; индекс  $j$ , определяющий положение процессора  $p_{ij}$  в строке, вычисляется по со-



отношению  $j = (i + l - 1) \bmod (k + 1)$ , ( $\bmod$  - операция получения остатка от целочисленного деления);

- полученные в результате пересылок блоки  $A'_{ij}$  и  $B'_{ij}$  каждого процессора  $p_{ij}$  перемножаются и прибавляются к блоку  $C_{ij}$ :  $C_{ij} = C_{ij} + A'_{ij} \times B'_{ij}$ ;

- блоки  $B'_{ij}$  каждого процессора  $p_{ij}$  пересылаются процессорам  $p_{ij}$ , являющимся соседями сверху в столбцах процессорной решетки (блоки процессоров из первой строки решетки пересылаются процессорам последней строки решетки).

Управляющий процессор (процессор 0) выполняет:

- создание топологии «решетка» (функция *CreateGrid*);
- создание производного типа «блок матрицы» (функция *DefineType*) и пересылку блоков на соответствующие процессоры в топологии «решетка» (функция *BcastMatrixFox*);
- итерации параллельного перемножения матриц (функция *MultiplyMatrixFox*);
- сбор результатов (перемноженных блоков) от всех процессоров (функция *GatherMatrixFox*).

Функциональные процессоры выполняют:

- создание топологии «решетка» (функция *CreateGrid*);
- создание производного типа «блок матрицы» (функция *DefineType*);
- прием от управляющего процессора соответствующих блоков матриц (функция *BcastMatrixFox*);
- итерации параллельного перемножения матриц (функция *MultiplyMatrixFox*);
- передачу результата (перемноженного блока) на управляющий процессор.

Формат вызова параллельного алгоритма Фокса имеет вид:

```
void ParallelMultiplyFox (int *pMatrixA, int *pMatrixB, int **pMatrixC,
int DataSize)
```

где:

- int \*pMatrixA и int \*pMatrixB - перемножаемые матрицы;
- int \*\*pMatrixC - матрица результата;
- int DataSize - порядок матриц.

Пример использования алгоритма:

```
#include "paralib.h"
#include "mpi.h"
int main(int argc, char *argv[])
{
    MPI_Init ( &argc, &argv );
    int DataSize = 100;
    int *pMatrixA = new int[DataSize*DataSize];
    int *pMatrixB = new int[DataSize*DataSize];
    int *pMatrixC;
    for(int i=0; i<DataSize*DataSize; i++)
    {
        pMatrixA[i] = (int) rand()/1000;
        pMatrixB[i] = (int) rand()/1000;
    }
    // запуск параллельного алгоритма Фокса
    ParallelMultiplyFox (pMatrixA, pMatrixB, &pMatrixC, DataSize);
    MPI_Finalize();
    return 0;
}
```

### *Алгоритм Кэннона*

Отличие алгоритма Кэннона от алгоритма Фокса состоит в изменении схемы начального распределения блоков перемножаемых матриц между процессорами вычислительной системы. Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы располагаемые блоки на процессорах могли быть перемножены без каких-либо дополнительных передач данных между процессорами. При этом подобное распределение блоков может быть организовано таким образом, что перемещение блоков между процессорами в ходе вычислений может осуществляться с использованием более простых коммуникационных операций.

С учетом высказанных замечаний этап инициализации алгоритма Кэннона включает выполнение следующих операций передач данных:

- на каждый процессор  $p_{ij}$  передаются блоки  $A_{ij}$ ,  $B_{ij}$ ;
- для каждой строки  $i$  процессорной «решетки» блоки матрицы  $A$  сдвигаются на  $(i - 1)$  позиций влево;
- для каждого столбца  $j$  процессорной «решетки» блоки матрицы  $B$  сдвигаются на  $(j - 1)$  позиций вверх.

В ходе вычислений на каждой итерации алгоритма Кэннона каждый блок матрицы  $A$  сдвигается на один процессор влево по решетке, а каждый блок матрицы  $B$  - на один процессор вверх.

Управляющий процессор (процессор 0) выполняет:

- создание топологии «решетка»;
- создание производного типа «блок матрицы» и пересылку блоков на соответствующие процессоры в топологии «решетка» (функция *BcastMatrixCannon*);
- выполнение итерации параллельного перемножения матриц (эта часть действий является общей для всех процессоров и детально описана в алгоритме работы функциональных процессоров);
- сбор результатов (перемноженных блоков) от всех процессоров (функция *GatherMatrixCannon*).

Функциональные процессоры выполняют:

- создание топологии «решетка»;
- создание производного типа «блок матрицы»;
- прием от процессора 0 соответствующих блоков матриц (функция *BcastMatrixCannon*);
- итерации параллельного перемножения матриц (функция *MultiplyMatrixCannon*): локальное перемножение блоков матриц на каждом процессоре (функция *MultiplyLocalMatrixCannon*); циклический сдвиг блоков  $A_{ij}$  влево на один шаг; циклический сдвиг блоков  $B_{ij}$  вверх на один шаг;
- передачу результата (перемноженного блока) на управляющий процессор.

Формат вызова параллельного алгоритма Кэннона имеет вид:

```
void ParallelMultiplyCannon (int *pMatrixA, int *pMatrixB, int **pMatrixC, int DataSize )
```

где:

int \*pMatrixA и int \*pMatrixB - перемножаемые матрицы;

int \*\*pMatrixC - матрица результата;

int DataSize - порядок матриц.

Пример использования алгоритма:

```
#include "paralib.h"
#include "mpi.h"
int main(int argc, char *argv[])
{
    MPI_Init ( &argc, &argv );
```

```

int DataSize = 100;
int *pMatrixA = new int[DataSize*DataSize];
int *pMatrixB = new int[DataSize*DataSize];
int *pMatrixC;
// генерация матриц случайным образом
for( int i=0; i<DataSize*DataSize; i++)
{
pMatrixA[i] = (int) rand()/1000;
pMatrixB[i] = (int) rand()/1000;
}
//запуск параллельного алгоритма Кэннона
ParallelMultiplyCannon (pMatrixA, pMatrixB, &pMatrixC, DataSize);
MPI_Finalize();
return 0;
}

```

### *Ленточный алгоритм*

При ленточной схеме разделения данных исходные матрицы разбиваются на горизонтальные (для матрицы *A*) и вертикальные (для матрицы *B*) полосы (рис.7).



Рис.7. Разбиение данных при выполнении ленточного алгоритма

Получаемые полосы распределяются по процессорам, при этом на каждом из имеющегося набора процессоров располагается только по одной полосе матриц *A* и *B*. Перемножение полос (эту операцию процессоры могут выполнить параллельно) приводит к получению части блоков результирующей матрицы *C*. Для вычисления оставшихся блоков матрицы *C* сочетания полос матриц *A* и *B* на процессорах должны быть изменены. В наиболее простом виде это может быть обеспечено, например, при кольцевой топологии вычислительной сети (при числе процессоров, равном количеству полос), т.е. необходимое для матричного умножения изменение положения данных может быть обеспечено циклическим сдвигом полос матрицы *B* по кольцу. После многократного выполнения описанных действий (количество необходимых повторений является равным числу процессоров) на каждом процессоре получается набор блоков, образующий горизонтальную полосу матрицы *C*.

Рассмотренная схема вычислений позволяет определить параллельный алгоритм матричного умножения при ленточной схеме разделения данных как итерационную процедуру, на каждом шаге которой происходит параллельное выполнение операции перемножения полос и последующего циклического сдвига полос одной из матриц по кольцу.

*Управляющий процессор* (процессор 0) выполняет:

- создание производного типа - «полоса матрицы» (функция *DefineTypeColumn*) и рассылку полос матриц на соответствующие процессоры (функция *BcastMatrixStrip*);
- итерации параллельного перемножения матриц (действия общие для всех процессоров);
- сбор результатов (перемноженных блоков) от всех процессоров (функция *GatherMatrixStrip*).

*Функциональные процессоры* выполняют:

- создание производного типа - «полоса матрицы» (функция *DefineType*);
- прием от управляющего процессора соответствующих полос матриц (функция *BcastMatrixStrip*);
- итерации параллельного перемножения матриц (функция *MultiplyMatrixStrip*): локальное перемножение элементов на каждом процессоре (функция *MultiplyLocalMatrixStrip*); циклический сдвиг полос матрицы *B* вдоль матрицы *A* (функция *MultiplyMatrixStrip*);
- передачу результата (перемноженного блока) на управляющий процессор.

Формат вызова параллельного алгоритма ленточного умножения имеет вид:

```
void ParallelMultiplyStrip ( int *pMatrixA, int *pMatrixB, int **pMatrixC, int  
DataSize )
```

где:

int \*pMatrixA и int \*pMatrixB - перемножаемые матрицы;

int \*\*pMatrixC - матрица результата;

int DataSize - порядок матриц.

Пример использования алгоритма:

```
#include "paralib.h"  
#include "mpi.h"  
int main(int argc, char *argv[])  
{  
    MPI_Init ( &argc, &argv );  
    int DataSize = 100;
```

```

int *pMatrixA = new int[DataSize*DataSize];
int *pMatrixB = new int[DataSize*DataSize];
int *pMatrixC;
// генерация матриц случайным образом
for(int i=0; i<DataSize*DataSize; i++)
{
pMatrixA[i] = (int) rand()/1000;
pMatrixB[i] = (int) rand()/1000;
}
//запуск параллельного ленточного алгоритма
ParallelMultiplyStrip (pMatrixA, pMatrixB, &pMatrixC, DataSize);
MPI_Finalize();
return 0;
}

```

## Лабораторное задание

Алгоритм проведения лабораторной работы следующий (папки DataStruct и Obhod).

1. Исследовать работу стека, очереди и дека, определив алгоритмы работы перечисленных структур данных. Результаты записать в тетрадь.

Используя программу **DataStruct**, задать исходное множество элементов, содержащее 12 - 15 элементов. Вызов программы:

Путь к файлу: D:\ИПОВС\АиСД\ DataStruct.

Система работает в диалоговом режиме с использованием меню. Вся необходимая информация во время работы системы отображается на экране дисплея и не требует специальных пояснений.

2. Используя программы **Obhod** либо **Obhod1**, сформировать бинарные деревья, содержащие 8, 10, 12 элементов и выполнить для каждого из них три вида обхода дерева.

Путь к программе: D:\ИПОВС\АиСД\OBHOD\Obhod1.

3. Разработать программу анализа арифметического выражения (рис.8).

Программа должна:

а) напечатать приглашение для ввода строки; прочесть с клавиатуры строку, введенную пользователем;

б) с помощью стека проанализировать правильность расстановки круглых скобок: если в строке встретилась открывающая скобка, то записать ее в стек; если встретилась закрывающая скобка, то извлечь один символ из стека; операции записи и извлечения символа из стека реали-

зовать в виде двух функций: PUSH и POP. Функция PUSH принимает символ и возвращает код возврата (0 - норма, 1 - переполнение стека), функция POP возвращает символ с вершины стека и код возврата (0 - норма, 1 - стек пуст);

в) напечатать сообщение о правильности или ошибочности введенной строки;

г) повторять действия, описанные в пп. а) - в), до тех пор, пока пользователь не введет пробел.

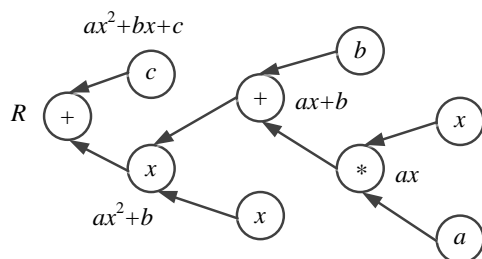


Рис.8. Запись арифметических выражений

4. Написать программу для вычисления выражения  $a_n$  по формуле варианта, соответствующего номеру ЭВМ (варианты заданий). Вычисления организовать в виде рекурсивной функции. Программу выполнить по шагам, записать в конспекте последовательное изменение стека.

5. Используя программу **БинДеревья**, сформировать бинарные деревья, содержащие 9, 10, 11 элементов, и выполнить для каждого из них балансировку.

6. Составить программу для формирования элементов массива.

## Варианты заданий

Составить программу для формирования элементов массива.

$$1; 13 \quad a_n = \frac{n!}{2^n} \quad 7; 19 \quad a_n = \left(\frac{2}{3}\right)^n$$

$$2; 14 \quad a_n = \frac{2^n}{n!}$$

$$8; 20 \quad a_n = \frac{3^n}{n!}$$

$$3; 15 \quad a_n = \frac{3^{n+1}}{n!}$$

$$9; 22 \quad a_n = \frac{n!}{2^{n-1}}$$

$$4; 16 \quad a_n = \frac{n!}{2^n 3^{n+1}}$$

$$10; 23 \quad a_n = \frac{2^{n+1}}{3^{n-1}}$$

$$5; 17 \quad a_n = \left(\frac{2}{3}\right)^n n!$$

$$11; 24 \quad a_n = 2^n (n-1)!$$

$$6; 18 \quad a_n = \frac{3^n}{2n!}$$

$$12; 25 \quad a_n = \frac{n!}{4^n}$$

Для определения времени выполнения программы воспользуйтесь стандартной встроенной функцией GETTIME.

### Контрольные вопросы

1. Каковы особенности итеративного и рекурсивного алгоритма?
2. В каких случаях целесообразно использовать рекурсивный или итеративный алгоритм? Приведите примеры итерации и рекурсии.
3. Все ли языки программирования дают возможность рекурсивного вызова процедур?
4. Приведите пример рекурсивной структуры данных.
5. Что такое указатели и динамические переменные в языке Турбо Паскаль?
6. Укажите виды обхода бинарных деревьев.
7. Приведите пример рекурсивной структуры данных.
8. Что такое указатели и динамические переменные в алгоритмических языках?



## Лабораторная работа № 4

### Алгоритмы построения остовного (покрывающего) дерева сети

**Цель работы:** ознакомление с алгоритмами построения остовного дерева графа (сети) и методикой оценки их эффективности.

**Продолжительность работы:** 2 часа.

#### Теоретические сведения

Пусть требуется принять решение, связанное с организацией сети компьютеров в различных территориальных пунктах. Это решение является довольно сложным и зависит от большого количества факторов, которые включают в себя вычислительные ресурсы, доступные в каждом пункте, соответствующие уровни потребностей, пиковые нагрузки на систему, возможное неэффективное использование основного ресурса в системе и, кроме того, стоимость предлагаемой сети. В эту стоимость входит приобретение оборудования, прокладка линий связи, обслуживание системы и т.д. Необходимо определить стоимость такой сети.

Нетрудно видеть, что сформулированная здесь задача имеет много аналогов. Например, требуется соединить несколько населенных пунктов линиями телефонной связи таким образом, чтобы все эти пункты были связаны в сеть и стоимость прокладки коммуникаций была минимальной. Можно также рассматривать ситуацию с прокладкой водопроводных коммуникаций, строительством дорог и т.д. Решение подобных задач возможно с использованием теории графов (сетей).

Пусть  $G = (V, E)$  - связный неориентированный граф, содержащий циклы, т.е. замкнутые маршруты, где  $V$  - множество вершин, а  $E$  - множество ребер. Остовным (покрывающим) деревом называется подграф, не содержащий циклов, включающий все вершины исходного графа, для которого сумма весов ребер минимальна.

Введем понятие цикломатического числа  $\gamma$ , показывающего, сколько ребер на графе нужно удалить, чтобы в нем не осталось ни одного цикла. Цикломатическое число равно увеличенной на единицу разности между количеством ребер  $n$  и количеством вершин графа  $m$ :

$$\gamma = n - m + 1.$$

Например, для графа, изображенного на рис.1, цикломатическое число равно  $\gamma = n - m + 1 = 8 - 5 + 1 = 4$ .

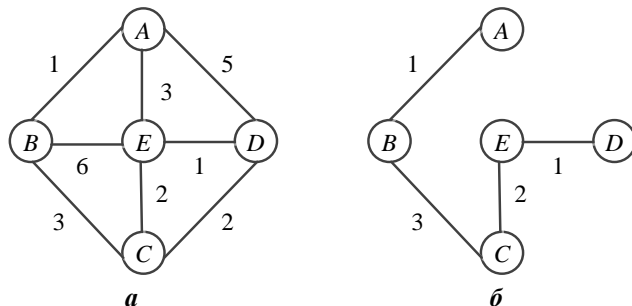


Рис.1. Исходный граф (а) и его остовное дерево (б)

Это значит, что если на графе (рис.1,а) убрать четыре ребра, то в нем не останется ни одного цикла, а суммарный вес ребер будет равен 7. Для построения остовного дерева графа используются методы Крускала и Прима.

### Метод Крускала

Вначале осуществляется предварительная сортировка весов ребер в порядке их возрастания. В начале работы алгоритма принимается, что в искомом остове не проведено ни одно ребро (т.е. остов состоит из изолированного множества вершин  $v_1, v_2, \dots, v_m$ , где  $m$  - количество вершин графа). Считается, что множество  $W_s$  имеет вид:

$$W_s = \{ \{v_1\}, \{v_2\}, \dots, \{v_n\} \},$$

где  $\{v_i\}$  обозначает множество, состоящее из единственной изолированной вершины  $v_i$ . Проверка  $v_k, v_l \in W_s$  предполагает установление факта: входят ли вершины  $v_k, v_l$  во множество  $W_s$  как изолированные, или они сами входят в подмножества постепенно увеличивающихся связных

вершин  $W_k, W_l$ , каждое из которых имеет вид:  $W_k = \{\dots, v_k, \dots\}$  и  $W_l = \{\dots, v_l, \dots\}$ .

Если обе вершины  $v_k, v_l$  содержатся в одном из подмножеств  $W_k, W_l$ , то ребро  $(k, l)$  в остов не включается, в противном случае данное ребро включается в остов, а множества  $W_k, W_l$  объединяются. Работа метода Крускала заканчивается, когда множество  $W_s$  совпадет по мощности с множеством всех вершин графа  $V$ . Нетрудно видеть, что это произойдет, когда все вершины графа окажутся связными.

**Пример.** Дана схема микрорайона. Необходимо соединить дома телефонным кабелем таким образом, чтобы его длина была минимальной.

Схему микрорайона представим взвешенным графом (рис.2).

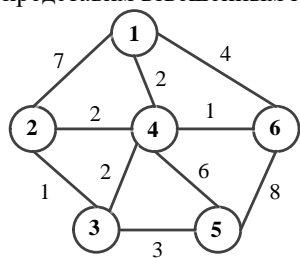


Рис.2. Граф-схема микрорайона

Определим цикломатическое число графа:

$$\gamma = n - m + 1 = 10 - 6 + 1 = 5,$$

т.е. на графе необходимо удалить пять ребер.

Первоначально каждая вершина исходного графа помещается в одноэлементное подмножество, т.е. считаем, что все вершины изолированы (не связаны). Ребро включается в остовное дерево, если оно связывает вершины, принадлежащие разным подмножествам, при этом вершины объединяются в новое подмножество.

В табл.1 последовательно включаются ребра в порядке возрастания их весов. Ребро  $(V_2, V_3)$  связывает две вершины, принадлежащие разным подмножествам  $\{V_2\}$  и  $\{V_3\}$ . Поэтому ребро включается в остовное дерево, а вершины объединяются в одно подмножество  $\{V_2, V_3\}$ . Ребро  $(V_4, V_6)$  также связывает вершины из разных подмножеств, оно включается в остовное дерево, а вершины образуют подмножество  $\{V_4, V_6\}$ .

Таблица 1

## Этапы построения остоного дерева методом Крускала

Ребро	Вес	Множества вершин	Операция
		$\{V_1\}, \{V_2\}, \{V_3\}, \{V_4\}, \{V_5\}, \{V_6\}$	
$(V_2, V_3)$	1	$\{V_2, V_3\}, \{V_1\}, \{V_4\}, \{V_5\}, \{V_6\}$	Включение
$(V_4, V_6)$	1	$\{V_2, V_3\}, \{V_4, V_6\}, \{V_1\}, \{V_5\}$	Включение
$(V_1, V_4)$	2	$\{V_2, V_3\}, \{V_1, V_4, V_6\}, \{V_5\}$	Включение
$(V_3, V_4)$	2	$\{V_1, V_2, V_3, V_4, V_6\}, \{V_5\}$	Включение
$(V_2, V_4)$	2	_____	Исключение
$(V_3, V_5)$	3	$\{V_1, V_2, V_3, V_4, V_5, V_6\}$	Включение

Вершины  $V_2$  и  $V_4$  находятся в одном подмножестве, поэтому ребро  $(V_2, V_4)$  исключается из рассмотрения.

Алгоритм заканчивает работу, когда все вершины объединяются в одно множество, при этом оставшиеся ребра не включаются в остоное дерево. Последовательно просматривая табл.1, получим схему соединения телефонным кабелем домов в микрорайоне (рис.3).

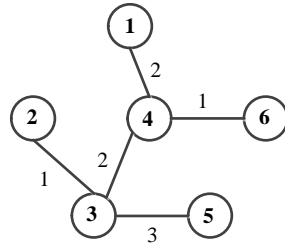


Рис.3. Остоное дерево минимального веса

**Метод Прима**

При использовании метода Прима от исходного графа переходим к его представлению в виде матрицы смежности. На графе выбирается ребро минимального веса. Выбранное ребро вместе с вершинами образует первоначальный фрагмент остоного дерева. Затем анализируются веса ребер от каждой вершины фрагмента до оставшихся невыбранных вершин. Выбирается минимальное ребро, которое присоединяется к первоначальному фрагменту и т.д. Процесс продолжается до тех пор, пока в остоное дерево не будут включены все вершины исходного графа.

**Пример.** Дана схема компьютерной сети (рис.4). Необходимо соединить компьютеры таким образом, чтобы длина проводки была минимальной.

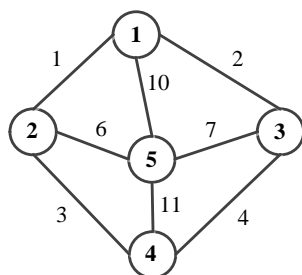


Рис.4. Схема компьютерной сети

Определим цикломатическое число графа:  $\gamma = n - m + 1 = 8 - 5 + 1 = 4$ . При просмотре любой строки табл.2 находим минимальное значение веса ребра, выделяем его и после этого столбец, в котором находится ребро минимального веса, в рассмотрении не участвует.

Таблица 2

Этапы работы метода Прима

Выбранные вершины	Невыбранные вершины			
	$V_1$	$V_2$	$V_3$	$V_4$
$V_5$	10	(6)	7	11
$V_2$	(1)	*	7	3
$V_1$	*	*	(2)	3
$V_3$	*	*	*	(3)

Для построения остоного дерева необходимо просмотреть столбцы табл.2 снизу вверх и зафиксировать первое появление минимальной величины:  $V_4 - V_2$ ;  $V_3 - V_1$ ;  $V_2 - V_5$ ;  $V_1 - V_2$ .

В итоге получим остоное дерево минимального веса (рис.5). Этапы работы метода Прима показаны на рис.6.

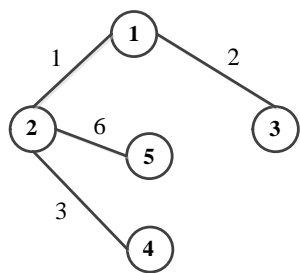


Рис.5. Остовное дерево графа

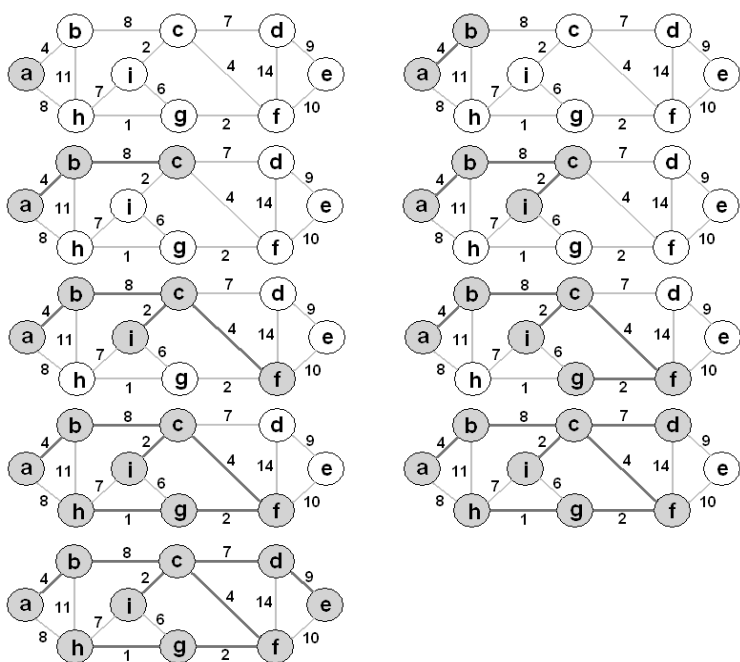
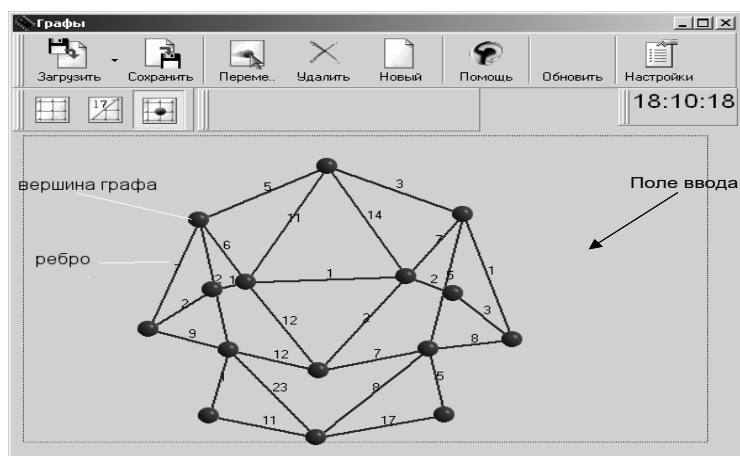


Рис.6. Работа алгоритма Прима

## Лабораторное задание

Данная программа предназначена для демонстрации работы алгоритмов и облегчения понимания терминов, употребляемых в теории графов (рис.7).



При нажатой кнопке «Выравнивать по сетке» новые вершины будут автоматически выравниваться по координатной сетке.

2. Вызвать программу **SiAOD**, включающую изучение методов Крускала и Прима.

Путь к файлу: D:\ИПОВС\АиСД\OSTOV\SiAOD

3. Вызвать обучающе-контролирующую программу **Ostov**, включающую в себя изучение методов Крускала и Прима. После окончания работы программы выставляется оценка.

Путь к файлу: D:\ИПОВС\АиСД\OSTOV\Ostov

Система работает в диалоговом режиме с использованием меню. Вся необходимая информация во время работы системы отображается на экране дисплея и не требует специальных пояснений.

Для графов, содержащих 10 и 12 вершин, построить остовные деревья методами Крускала и Прима (по заданию преподавателя).

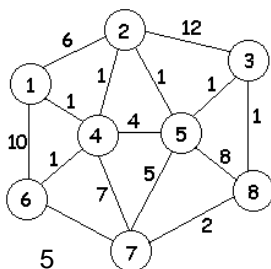
4. Составить программы для исследования методов Крускала и Прима.

5. Решить задачи, включенные в варианты заданий. Результаты записать в тетрадь.

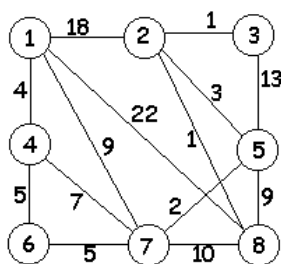
6. Построить остовное дерево графа методами Крускала и Прима.

### Варианты заданий

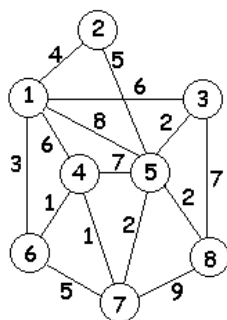
1; 2



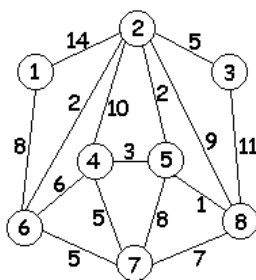
3; 4



5; 6

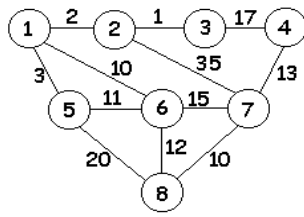


7; 8

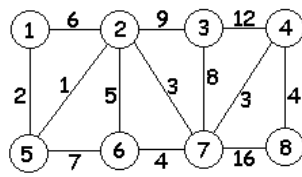




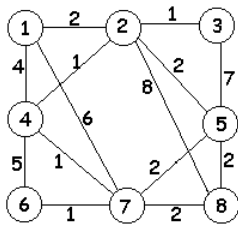
9; 10



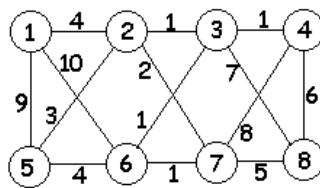
11; 12



13; 14



15; 16



## Контрольные вопросы

1. Что понимается под остовным деревом?
2. Каковы особенности методов Крускала и Прима?
3. В чем состоит методика анализа сложности алгоритмов построения остовного дерева графа?
4. Определить, является ли связным заданный граф.
5. Найти все вершины графа, к которым существует путь заданной длины от выделенной вершины графа.
6. Найти все вершины графа, достижимые из заданной.
7. Подсчитать количество компонент связности заданного графа.
8. Найти диаметр графа, т.е. максимум расстояний между всевозможными парами его вершин.
9. Найти такую нумерацию вершин орграфа, при которой всякая дуга ведет от вершины с меньшим номером к вершине с большим номером.

## Лабораторная работа № 5

### Алгоритмы нахождения кратчайших путей на графах

**Цель работы:** изучение некоторых алгоритмов нахождения кратчайших путей на графах; исследование эффективности этих алгоритмов.

**Продолжительность работы:** 2 часа.

#### Теоретические сведения

Задача отыскания в графе (как ориентированном, так и неориентированном) кратчайшего пути имеет многочисленные практические приложения. С решением подобной задачи приходится встречаться в технике связи (например, при телефонизации населенных пунктов), на транспорте (при выборе оптимальных маршрутов доставки грузов), в микроэлектронике (при проектировании топологии микросхем) и т.д.

Путь между вершинами  $i$  и  $j$  графа считается кратчайшим, если вершины  $i$  и  $j$  соединены минимальным числом ребер (случай невзвешенного графа) или если сумма весов ребер, соединяющих вершины  $i$  и  $j$ , минимальна (для взвешенного графа).

В настоящее время известны десятки алгоритмов решения поставленной задачи.

Важным показателем алгоритма является его эффективность. Применительно к поставленной задаче эффективность алгоритма может зависеть в основном от двух параметров графа: числа его вершин и числа весов его ребер.

В настоящей лабораторной работе для определения кратчайшего расстояния между вершинами графа исследуются два алгоритма: метод динамического программирования и метод Дейкстры.

#### *Метод динамического программирования*

Метод рассматривает многостадийные процессы принятия решения. При постановке задачи динамического программирования формируется некоторый критерий. Процесс разбивается на стадии (шаги),

в которых принимаются решения, приводящие к достижению общей цели. Таким образом, метод динамического программирования - метод пошаговой оптимизации.

Введем функцию  $f_i$ , определяющую минимальную длину пути из начальной вершины в вершину  $i$ . Обозначим через  $S_{ij}$  длину пути между вершинами  $i$  и  $j$ , а  $f_j$  - наименьшую длину пути между вершиной  $j$  и начальной вершиной. Выбирая в качестве  $i$  такую вершину, которая минимизирует сумму  $(S_{ij} + f_j)$ , получаем уравнение:

$$f_i = \min \{S_{ij} + f_j\}.$$

Трудность решения данного уравнения заключается в том, что неизвестная функция входит в обе части равенства. В такой ситуации приходится прибегать к классическому методу последовательных приближений (итераций), используя рекуррентную формулу:

$$f_i^{(k+1)} = \min \{S_{ij} + f_j^{(k)}\},$$

где  $f_j^{(k)}$  -  $k$ -е приближение функции.

Возможен другой подход к решению поставленной задачи с помощью метода стратегий. При движении из начальной точки  $i$  в конечную  $k$  получается приближение  $f_i^{(0)} = S_{ik}$ , где  $S_{ik}$  - длина пути между точками  $i$  и  $k$ . Следующее приближение - поиск решения в классе двухзвенных ломаных. Дальнейшие приближения ищутся в классе трехзвенных, четырехзвенных и других ломаных.

**Пример.** Определить кратчайший путь из вершины 1 в вершину 10 для взвешенного ориентированного графа (рис.1).

Начальные условия:  $f_1 = 0$ ,  $S_{11} = 0$ .

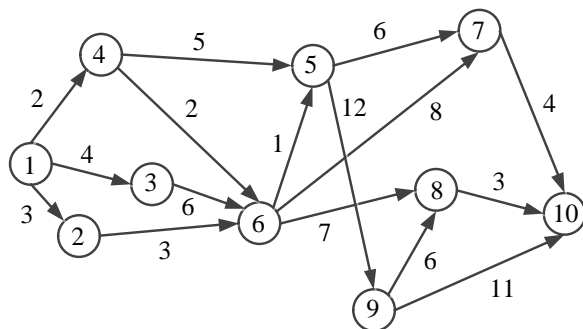


Рис.1. Взвешенный ориентированный граф

Находим последовательно значения функции  $f_i$  (в условных единицах) для каждой вершины ориентированного графа:

$$f_2 = \min\{S_{12} + f_1\} = \{3 + f_1\} = \{3 + 0\} = 3;$$

$$f_3 = \min\{S_{13} + f_1\} = \{4 + f_1\} = \{4 + 0\} = 4;$$

$$f_4 = \min\{S_{14} + f_1\} = \{2 + f_1\} = \{2 + 0\} = 2;$$

$$f_6 = \min\left\{\begin{matrix} S_{46} + f_4 \\ S_{36} + f_3 \\ S_{26} + f_2 \end{matrix}\right\} = \min\left\{\begin{matrix} 2 + f_4 \\ 6 + f_3 \\ 3 + f_2 \end{matrix}\right\} = \min\left\{\begin{matrix} 2 + 2 \\ 6 + 4 \\ 3 + 3 \end{matrix}\right\} = 4;$$

$$f_5 = \min\left\{\begin{matrix} S_{45} + f_4 \\ S_{65} + f_6 \end{matrix}\right\} = \min\left\{\begin{matrix} 5 + f_4 \\ 1 + f_6 \end{matrix}\right\} = \left\{\begin{matrix} 5 + 2 \\ 1 + 4 \end{matrix}\right\} = 5;$$

$$f_7 = \min\left\{\begin{matrix} S_{57} + f_5 \\ S_{67} + f_6 \end{matrix}\right\} = \min\left\{\begin{matrix} 6 + f_5 \\ 8 + f_6 \end{matrix}\right\} = \min\left\{\begin{matrix} 6 + 5 \\ 8 + 4 \end{matrix}\right\} = 11;$$

$$f_9 = \min\{S_{59} + f_5\} = \{12 + f_5\} = \{12 + 5\} = 17;$$

$$f_8 = \min\left\{\begin{matrix} S_{68} + f_6 \\ S_{98} + f_9 \end{matrix}\right\} = \min\left\{\begin{matrix} 7 + f_6 \\ 6 + f_9 \end{matrix}\right\} = \min\left\{\begin{matrix} 7 + 4 \\ 6 + 17 \end{matrix}\right\} = 11;$$

$$f_{10} = \min\left\{\begin{matrix} S_{7,10} + f_7 \\ S_{8,10} + f_8 \\ S_{9,10} + f_9 \end{matrix}\right\} = \min\left\{\begin{matrix} 4 + f_7 \\ 3 + f_8 \\ 11 + f_9 \end{matrix}\right\} = \min\left\{\begin{matrix} 4 + 11 \\ 3 + 11 \\ 11 + 17 \end{matrix}\right\} = 14.$$

Длина кратчайшего пути составляет 14 условных единиц.

Для выбора оптимальной траектории движения следует осуществить просмотр функций  $f_i$  в обратном порядке, т.е. с  $f_{10}$ .

Пусть  $f_i = f_{10}$ . В данном случае:

$$f_{10} = \min\left\{\begin{matrix} S_{7,10} + f_7 \\ S_{8,10} + f_8 \\ S_{9,10} + f_9 \end{matrix}\right\} = \min\left\{\begin{matrix} 4 + f_7 \\ 3 + f_8 \\ 11 + f_9 \end{matrix}\right\} = \min\left\{\begin{matrix} 4 + 11 \\ 3 + 11 \\ 11 + 17 \end{matrix}\right\} = 14.$$

Получаем, что  $(3 + f_8) = 14$ , т.е.  $f_j = f_8$ . Значит, из вершины 10 следует перейти к вершине 8. Имеем  $f_i = f_8$ . Рассмотрим функцию

$$f_8 = \min\left\{\begin{matrix} S_{68} + f_6 \\ S_{98} + f_9 \end{matrix}\right\} = \min\left\{\begin{matrix} 7 + f_6 \\ 6 + f_9 \end{matrix}\right\} = \min\left\{\begin{matrix} 7 + 4 \\ 6 + 17 \end{matrix}\right\} = 11, \text{ т.е. } f_j = f_6 \text{ и т.д.}$$

Таким образом, получаем кратчайший путь от вершины 1 к вершине 10:

(1, 4, 6, 8, 10).

Рассмотренный метод определения кратчайшего пути может быть распространен и на неориентированные графы.

### ***Метод Дейкстры***

Метод Дейкстры предназначен для нахождения кратчайшего пути между вершинами в неориентированном графе.

Сначала выбираем путь до начальной вершины равным нулю, и заносим эту вершину во множество уже выбранных, расстояние от которых до оставшихся невыбранных вершин определено. На каждом следующем этапе находим невыбранную вершину, расстояние до которой наименьшее, соединенную ребром с какой-нибудь вершиной из множества выбранных (это расстояние будет равно расстоянию до уже выбранной вершины плюс длина ребра).

**Пример.** Найти кратчайший путь на графе между 1-й и 7-й вершинами (рис.2).

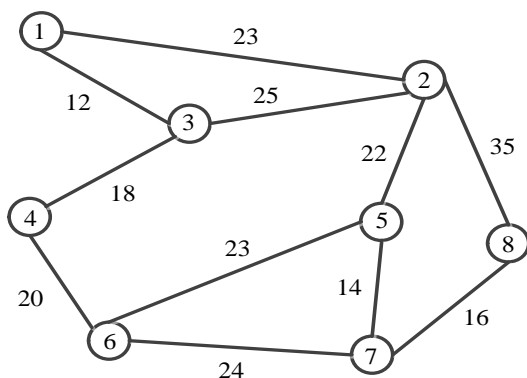


Рис.2. Исходный граф

В табл.1 представлена матрица смежности, соответствующая исходному графу.

Таблица 1

Матрица смежности, соответствующая графу

	Вершины							
	$V_1$	$V_2$	$V_3$	$V_4$	$V_5$	$V_6$	$V_7$	$V_8$
$V_1$	0	23	12	–	–	–	–	–
$V_2$	23	0	25	–	22	–	–	35
$V_3$	12	25	0	18	–	–	–	–
$V_4$	–	–	18	0	–	20	–	–
$V_5$	–	22	–	–	0	23	14	–
$V_6$	–	–	–	20	23	0	24	–
$V_7$	–	–	–	–	14	24	0	16
$V_8$	–	35	–	–	–	–	16	0

Определяем следующую выбранную вершину, расстояние до которой наименьшее и соединенную ребром с одной из вершин, принадлежащих множеству невыбранных (это расстояние будет равно расстоянию до выбранной вершины плюс длина ребра) (табл.2).

Таблица 2

Табличная реализация метода Дейкстры

Выбранная вершина	Кратчайший путь	Невыбранные вершины						
		$V_2$	$V_3$	$V_4$	$V_5$	$V_6$	$V_7$	$V_8$
$V_1$	0	23	(12)	–	–	–	–	–
$V_3$	12	(23)	*	30	–	–	–	–
$V_2$	23	*	*	(30)	45	–	–	58
$V_4$	30	*	*	*	(45)	50	–	58
$V_5$	45	*	*	*	*	(50)	59	58
$V_6$	50	*	*	*	*	*	59	(58)
$V_8$	58	*	*	*	*	*	(59)	*

Осуществляем обратный просмотр от конечной вершины к начальной. Просматриваем столбец, соответствующий вершине, снизу вверх и фиксируем первое появление минимальной величины. Кратчайший путь при этом будет равен  $(V_7 - V_5 - V_2 - V_1)$ .

### **Алгоритм Флойда**

Пусть в матрице  $A[i, j]$  записаны длины ребер графа (элемент  $A[i, j]$  равен весу ребра, соединяющего вершины с номерами  $i$  и  $j$ , если же такого ребра нет, то в соответствующем элементе записано некоторое очень большое число). Построим новые матрицы  $C_k[i, j]$  ( $k = 0, \dots, N$ ). Элемент матрицы  $C_k[i, j]$  будет равен минимально возможной длине такого пути из  $i$  в  $j$ , в котором в качестве промежуточных вершин используются вершины с номерами от 1 до  $k$ . В этом случае рассматриваются пути, которые могут проходить через вершины с номерами от 1 до  $k$ , но заведомо не проходят через вершины с номерами от  $(k + 1)$  до  $N$ . В матрицу записывается длина кратчайшего из таких путей. Если таких путей не существует, записывается то же большое число, которым обозначается отсутствие ребра.

Если вычислена матрица  $C_{k-1}[i, j]$ , то элементы матрицы  $C_k[i, j]$  можно определить по следующей формуле:

$$C_k[i, j] = \min(C_{k-1}[i, j], C_{k-1}[i, k] + C_{k-1}[k, j]).$$

Рассмотрим кратчайший путь из вершины  $i$  в вершину  $j$ , который в качестве промежуточных вершин использует только вершины с номерами от 1 до  $k$ . Тогда возможны два случая:

1) путь не проходит через вершину с номером  $k$ , значит его промежуточные вершины - это вершины с номерами от 1 до  $(k - 1)$ . Но длина этого пути уже вычислена в элементе  $C_{k-1}[i, j]$ ;

2) путь проходит через вершину с номером  $k$ . Но его можно разбить на две части: сначала из вершины  $i$  доходим оптимальным образом до вершины  $k$ , используя в качестве промежуточных вершины с номерами от 1 до  $(k - 1)$  (длина такого оптимального пути вычислена в  $C_{k-1}[i, k]$ ), а потом от вершины  $k$  идем в вершину  $j$  оптимальным способом, вновь используя в качестве промежуточных вершин только вершины с номерами от 1 до  $k$  ( $C_{k-1}[k, j]$ ).

Выбирая из этих двух вариантов кратчайший путь, получаем  $C_k[i, j]$ . Последовательно вычисляя матрицы  $C_0, C_1, C_2$  и т.д., получим искомую матрицу  $C_N$  кратчайших расстояний между всеми парами вершин в графе.



Алгоритм Флойда можно свести к последовательности шагов.

Присвоить  $c_{ij} = 0$  для всех  $i = 1, 2, \dots, n$  и  $c_{ij} = \infty$ , если в графе отсутствует дуга  $(x_i, x_j)$ .

*Задание начальных значений.*

**Шаг 1.** Присвоить  $k = 0$ .

**Шаг 2.**  $k = k + 1$ .

**Шаг 3.** Для всех  $i < k$ , таких, что  $c_{ik} < \infty$ , и для всех  $j < k$ , таких, что  $c_{ik} < \infty$ , вычислить

$$c_{ij} = [\min[c_{ij}, (c_{ik} + c_{kj})]].$$

*Проверка на окончание алгоритма.*

**Шаг 4.** Если  $k = n$ , то матрица  $[c_{ij}]$  дает длины всех кратчайших путей – останов, иначе перейти к шагу 2.

Пусть имеются три вершины  $v_i, v_j, v_k$  и заданы расстояния между ними. Если выполняется неравенство  $(w_{ik} + w_{kj}) < w_{ij}$ , то целесообразно заменить путь  $v_i \rightarrow v_j$  (путем  $v_i \rightarrow v_k \rightarrow v_j$ ). Такая замена выполняется систематически в процессе выполнения данного алгоритма (рис.3).

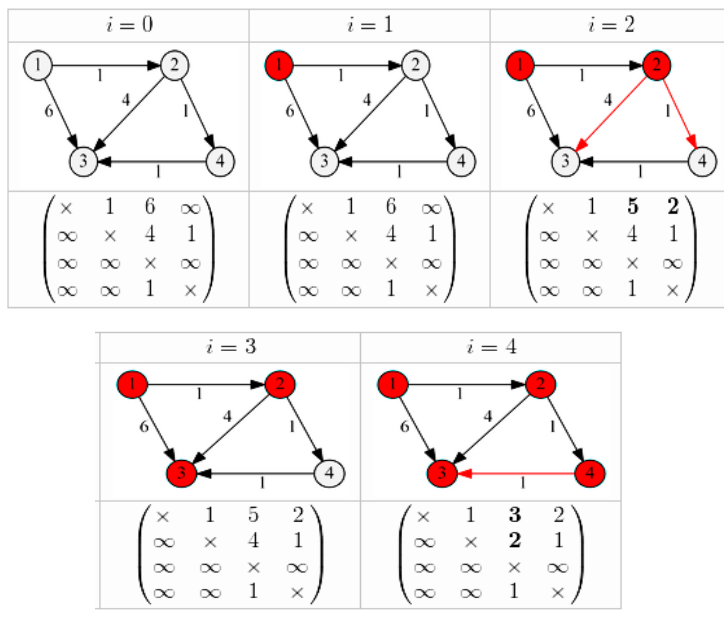


Рис.3. Алгоритм Флойда

### *Алгоритм Йена*

Пусть граф задан матрицей  $A[i, j]$ . Вершина  $s$  выбрана как начальная. Найдем длины  $k$  наименьших путей до каждой вершины от вершины  $s$  (ребра в одном пути могут повторяться неоднократно) при условии, что эти пути существуют. Результат будет храниться в матрице  $B$  размером  $N \times k$ , где  $N$  - количество вершин. Элемент массива  $B[i, j]$  равен  $j$ -му по длине пути до вершины  $i$ .

Для каждой вершины в массиве  $C$  будем хранить количество уже найденных до нее наименьших путей. Изначально все элементы массива  $C$  равны нулю. Все элементы матрицы  $B$  делаем равными какой-нибудь большой константе, заведомо большей всех возможных путей. Во время исполнения алгоритма в матрице  $B$  будем хранить лучшие  $k$  путей до каждой вершины, найденные во время исполнения, при этом первые  $C[i]$  путей для вершины  $i$  найдены уже окончательно (элементы матрицы  $B[i, 1], B[i, 2], \dots, B[i, k]$  для всех  $i$  упорядочены в возрастающем порядке). Таким образом, можно действовать на последующих шагах.

Пусть уже найдены какие-то кратчайшие пути. Тогда, чтобы найти следующий по длине путь, удлиним каждый из уже полученных на одно ребро. Найдем кратчайший из них, причем оканчивающийся на вершину, до которой найдено менее  $k$  путей и занесем его в таблицу результата.

Алгоритм Йена можно свести к последовательности шагов.

*Задание начальных значений.*

**Шаг 1.** Найти  $P^1$ . Присвоить  $k = 2$ . Если существует только один кратчайший путь  $P^1$ , то включить его в список  $L_0$  и перейти к шагу 2. Если таких путей несколько, но меньше, чем  $K$ , то включить один из них в список  $L_c$ , а остальные - в список  $L_1$ . Перейти к шагу 2. Если существует  $K$  или более кратчайших путей  $P^1$ , то задача решена. Останов.

*Нахождение отклонений.*

**Шаг 2.** Найти все отклонения  $P_i^k(k-1)$ -го кратчайшего пути  $P^{k-1}$  для всех  $i = 1, 2, \dots, q_{k-1}$ , выполняя для каждого  $i$  шаги с 3-го по 6-й.

**Шаг 3.** Проверить, совпадает ли подпуть, образованный первыми  $i$  вершинами любого из  $P^j$  путей ( $j = 1, 2, \dots, k-1$ ). Если совпадает, то присвоить  $c(x_i^{k-1}, x_{i+1}^j) = \infty$ ; иначе ничего не изменять. (При выполнении алгоритма вершина  $x_1$  обозначается строчной буквой  $s$ .)

**Шаг 4.** Найти кратчайшие пути  $S_i^k$  от  $x_i^{k-1}$  к  $t$ , исключая из рассмотрения вершины  $s, x_i^{k-1}, x_3^{k-1}, \dots, x_i^{k-1}$ . Если существует несколько кратчайших путей, то взять в качестве  $S_i^k$  один из них.

**Шаг 5.** Построить  $P_i^k$ , соединяя  $R_i^k(s, x^{-1})$  с  $S$  и поместить  $P$  в список  $L_1$ .

**Шаг 6.** Заменить элементы матрицы весов, измененные на шаге, их первоначальными значениями и возвратиться к шагу 3.

*Выбор кратчайших отклонений.*

**Шаг 7.** Найти кратчайший путь в списке  $L_1$ . Обозначить этот путь  $P^k$  и переместить его из  $L_1$  в  $L_0$ . Если  $k = K$ , то останов. Список  $L_0$  - список  $K$  кратчайших путей. Если  $k < K$ , то присвоить  $k = k + 1$ , перейти к шагу 2. Если в  $L_1$  имеется более чем один кратчайший путь ( $h$  путей), то поместить в  $L_0$  любой из них и продолжать выполнение действий, аналогичных приведенным выше, до тех пор, пока увеличенное на  $h$  число путей, уже находящихся в  $L_1$ , не совпадет с  $K$  или не превысит его. Тогда останов.

### **Алгоритм Беллмана - Форда**

Пусть в матрице  $A[i, j]$  записаны длины ребер графа. Найдем кратчайшие расстояния от заданной вершины  $s$  до всех остальных вершин графа. Алгоритм Беллмана - Форда решает эту задачу при наличии ребер отрицательного веса. Обозначим через  $\text{МинСт}(s, v, k)$  наименьшую стоимость проезда из  $s$  в  $v$  менее чем с  $k$  пересадками.

$$\text{МинСт}(s, v, k + 1) = \min(\text{МинСт}(s, v, k), \text{МинСт}(s, i, k) + A[i][v]), \\ (i = 1, \dots, n).$$

Искомый ответом является  $\text{МинСт}(s, i, n)$  для всех  $i = 1, \dots, n$ .

Чтобы найти не только длины наименьших путей до всех вершин, но и сами эти пути, используем следующий прием. Параллельно с вычислением массива  $x$  будем вычислять матрицу  $D[i, j]$ . Если между вершинами  $s$  и  $j$  существует путь, то в элементе массива  $D[j, 0]$  будет храниться количество вершин в этом пути, а цепочка вершин, составленная из элементов с  $D[j, 1]$  по  $D[j, D[j, 0]]$ , будет этим самым путем. Путь до вершины  $s$  содержит единственную вершину ( $D[s, 0] = 1, D[s, 1] = s$ ). Для вычисления матрицы  $D$  потребуется дополнить текст процедуры:

```
k:=1;
for i := 1 to n do begin x[i] := a[s][i]; end;
{ инвариант: x[i] := МинСт(s,i,k) }
```

Обозначим через  $\text{МинСт}(s, i, k)$  наименьшую стоимость проезда из  $s$  в  $i$  менее чем с  $k$  пересадками. Тогда выполняется следующий фрагмент:

```

for i := 1 to n do begin D[i,0]:=2; D[i,1]:=s; D[i,2]:=i; End;
D[s,0]:=1;
while k <> n do begin
  for i := 1 to n do begin
    for j := 1 to n do begin
      if x[i] > x[j]+a[j][i] then begin
        x[i] := x[j]+a[j][i];
        D[i]:=D[j];
        D[i,0]:=D[j,0]+1;
        D[i,D[i,0]]:=i;
      end;
    end
  end
  { x[i] = МинСт(s,i,k+1) }
end;
k := k + 1;
end;

```

## Лабораторное задание

Алгоритм выполнения настоящей лабораторной работы следующий.

1. Найти кратчайший путь методом динамического программирования для трех ориентированных графов. Зафиксировать параметры каждого графа (число вершин и ребер), значение пути и время решения задачи (варианты заданий, задание 1).

Путь к файлу: D:\ИПОВС\АиСД\KRATPUT\Kratput\_new

2. Найти кратчайший путь методом Дейкстры для трех неориентированных графов. Зафиксировать параметры каждого графа (число вершин и ребер), значение пути и время решения задачи (варианты заданий, задание 2).

Путь к файлу: D:\ИПОВС\АиСД\KRATPUT\Kratput1

Путь к файлу: D:\ИПОВС\АиСД\KRATPUT\Kratput2

Путь к файлу: D:\ИПОВС\АиСД\KRATPUT\Kratput3

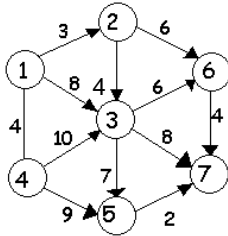
Программные модули работают в диалоговом режиме с использованием меню. Вся необходимая поясняющая информация отображается во время работы на экране монитора.

3. Ознакомиться с алгоритмами Флойда, Йена, Беллмана - Форда.

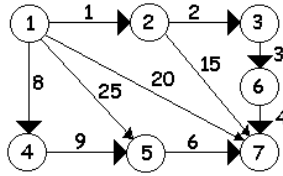
## Варианты заданий

**Задание 1.** Найти кратчайший путь на графе между двумя вершинами методом динамического программирования.

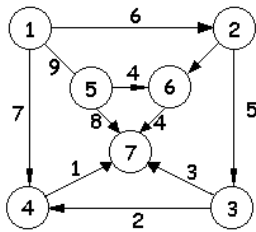
1; 2



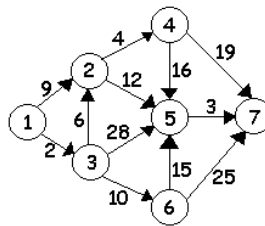
3; 4



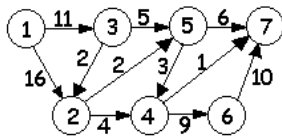
5; 6



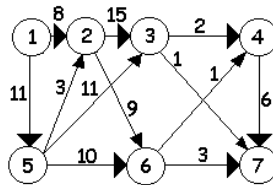
7; 8



9; 10

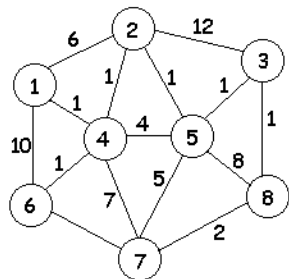


11; 12

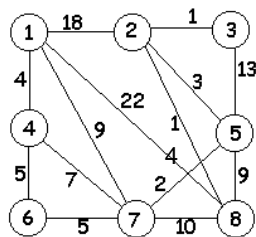


**Задание 2.** Найти кратчайший путь на графе между тремя парами вершин методом Дейкстры.

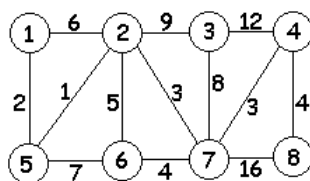
1; 2



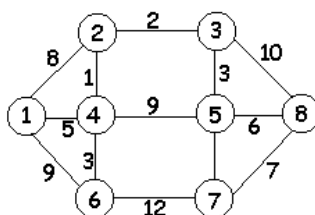
3; 4

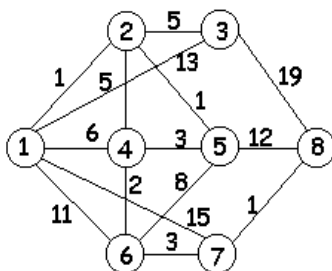
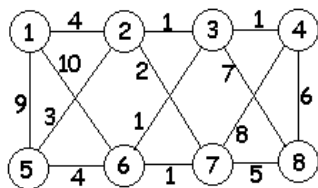


5; 6



7; 8





### Контрольные вопросы

1. Какова теоретическая сложность алгоритмов, рассмотренных в настоящей лабораторной работе?
2. В решении каких прикладных задач используются алгоритмы определения в графе кратчайших расстояний между заданными вершинами?
3. Может ли быть применен рассмотренный алгоритм Дейкстры при определении кратчайшего расстояния в ориентированном графе?
4. Как работает алгоритм Дейкстры?
5. Как работает алгоритм динамического программирования в задачах определения в графе кратчайших расстояний между вершинами?

## Лабораторная работа № 6

### Обработка геометрических объектов с помощью плексов

**Цель работы:** изучение эвристических алгоритмов и способов их разработки; сравнительная оценка использования эвристических и переборных алгоритмов для решения комбинаторных задач.

**Продолжительность работы:** 2 часа.

#### Теоретические сведения

Тема настоящей лабораторной работы относится к области обработки графических данных, потребность в анализе которых возникает при рассмотрении многих научно-технических и производственных задач. При построении различных программных систем общего и специального назначения (таких, например, как системы автоматизированного проектирования) могут оказаться необходимыми средства, обеспечивающие хранение графической информации на ЭВМ, возможность редактирования этой информации и использования ее в расчетах, а также получение изображений на экране монитора или на бумаге.

Настоящая лабораторная работа является введением в проблематику *структурного (векторного)* представления графической информации, когда изображения представляются не в виде растровых данных, а формируются в виде набора графических элементов. Такой подход позволяет существенно повысить эффективность анализа и обработки графических данных.

*Чертежом* в рамках настоящей лабораторной работы называют графическое изображение моделируемого объекта, характеризующее его форму и размеры и выполненное по определенным правилам проектирования с применением общепринятых соглашений по изображению и обозначению графических элементов.

По чертежам изготавливают детали и производят сборку узлов, машин и конструкций. Чертеж на производстве — это основной технический документ, в котором указаны необходимые сведения о де-



талях, их материале, термической обработке, технические условия на изготовление. По нему проверяют качество обработки деталей на всех стадиях технологического процесса. Различают чертеж детали, сборочный чертеж, чертеж общего вида и др.

В рамках настоящей лабораторной работы ставится задача разработки программного модуля (ПМ) для представления и редактирования отображения на экране монитора конкретного типа графических данных – *чертежей*, образуемых из ограниченного набора различных геометрических элементов (точек, линий, окружностей и т.п.). Разрабатываемый ПМ должен обеспечивать:

- представление чертежей в памяти ЭВМ;
- демонстрацию хранимых чертежей на экране монитора;
- редактирование чертежей (вставку и удаление линий);
- запись (чтение) данных о чертежах в файлах.

Освоение принципов командной работы является одной из ключевых учебных задач настоящей лабораторной работы. Для успешного выполнения лабораторной работы ее реализация должна быть поручена группе студентов из трех-четырех человек. Для координации выполняемых работ в каждой группе может быть выделен ответственный за разработку (*главный программист*), в задачу которого входило бы согласование заданий на разработку, распределение работ между участниками разработки, согласование спецификаций и т.п. Отдельный разработчик (*тестировщик*) может отвечать за тестирование разрабатываемого кода. Еще один разработчик (*технический писатель*) может быть ориентирован на подготовку документации и презентаций по программному коду.

Важной частью разработки программного модуля является реализация диалоговой программы для взаимодействия с пользователем. Данная часть разработки также может быть поручена участнику группы разработчиков. С другой стороны, возможный вариант выполнения лабораторной работы - анализ уже имеющихся (ранее разработанных) диалоговых программ. В этом случае группе разработчиков нужно освоить правила использования (API) переданных для применения программ.

Следует отметить важность выделенных ролей участников группы разработчиков. От тестировщика зависит безошибочность (надежность) разработанного кода программ. От разработчика диалоговой программы зависит удобство использования программы. И наконец, от технического редактора зависит простота сопровождения и развития проекта и

возможность более широкого привлечения потенциальных потребителей разработанных программ.

В рамках выполнения настоящей лабораторной работы могут быть использованы следующие основные допущения:

- в качестве чертежа будем рассматривать плоский геометрический объект, состоящий из отрезков прямых (линий) и граничных точек этих линий (представление других геометрических элементов может рассматриваться как тема дополнительных заданий);
- набор линий, образующих чертеж, должен быть связным, т.е. любая линия чертежа должна иметь общую точку хотя бы с одной другой линией чертежа.

Пример возможного чертежа представлен на рис.1.

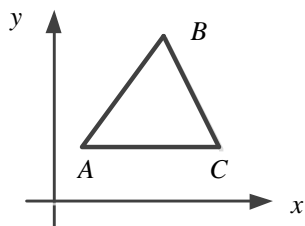


Рис.1. Пример чертежа, состоящего из трех линий и трех точек: A, B, C

### **Структуры данных**

Каждый чертеж может быть представлен в виде множества базовых геометрических объектов - точек, линий, окружностей и т.д. Информационное описание объектов включает в себя параметры фигуры (координаты, размер, радиус и др. - с различной степенью детализации в зависимости от требований, предъявляемых к чертежу в данной предметной области). В общем случае для описания фигуры необходимо предусмотреть хранение координат некоторой опорной точки. Операции обработки геометрических объектов состоят в задании и изменении параметров, а также визуализации фигур.

Для обеспечения динамической визуализации геометрических объектов в настоящей лабораторной работе вводится тип данных, значения которого вычисляются в соответствии с задаваемым формульным выражением (класс TFormValue).

**Схема иерархии** классов для представления базовых геометрических элементов может состоять в следующем.

Наиболее общие свойства геометрических фигур и методы их разработки выделяются в абстрактный базовый класс (TChartRoot). От него наследуются классы для представления точки (TChartPoint), на основе которого реализуется класс для представления линии (TChartLine) и классы для других геометрических элементов.

Точка определяется координатами на плоскости, значение координат определяется с помощью формульных объектов класса TFormValue. При формировании чертежа следует обратить внимание, что одна и та же точка в чертеже может присутствовать несколько раз - в этом случае следует обеспечить однократное представление точки без дублирования (для обеспечения корректной работы операций вставки и удаления).

Для представления линии в наиболее простом варианте необходимо хранить начальную и конечную точки. В более общем случае при формировании чертежа в объекте линии могут храниться не координаты точек, а указатели на другие линии чертежа. Данная ситуация допускается при выполнении следующих условий:

- начальная точка текущей линии может определяться с помощью указателя на линию, конечная точка которой является начальной точкой текущей линии;
- конечная точка текущей линии может определяться с помощью указателя на линию, конечная точка которой является конечной точкой текущей линии.

Наряду с базовыми геометрическими элементами, требующими для своего представления разработки отдельных классов, в настоящей лабораторной работе предлагается реализовать несколько различных способов формирования *составных геометрических объектов*, образуемых на основе уже существующих геометрических фигур (как базовых, так и составных) и рассматриваемых при выполнении операции обработки как единое целое. Возможные способы формирования составных объектов состоят в следующем.

- *Группирование*, когда составной объект представляет собой набор уже существующих (как базовых, так и составных) геометрических фигур - подобной операцией является, например, группирование в графическом редакторе текстового процессора MS Word.
- *Конструирование*, когда составной объект определяется на основе уже существующих геометрических фигур (например, ломаная мо-

жет быть определена через набор конечных точек составляющих отрезков). Конструирование приводит к созданию новой сложной геометрической фигуры, которая не может быть разделена на составные геометрические элементы.

- *Комбинирование*, когда составной объект формируется с помощью сборки существующих объектов (так чертеж может быть образован из точек и линий), в рамках сконструированного объекта входящие в состав элементы по-прежнему могут обрабатываться как отдельные геометрические фигуры.

Для представления данных, описывающих структуру чертежа, предлагается использовать структуру хранения типа **плекс**. Плекс (сплетение) - структура хранения, включающая звенья разных типов, отношения между которыми задаются с помощью сцепления (указателей). Таким образом, плекс является разновидностью многосвязного списка и используется для хранения сетевых моделей данных.

Основу плексов для представления чертежей, состоящих из линий и точек, составляют вершины (узлы), в каждой из которых располагается информация о той или иной линии чертежа (имя линии, координаты начальной и конечной точек).

Пример разработанной структуры хранения (см. плекс для чертежа из рис. 1) приведен на рис.2.

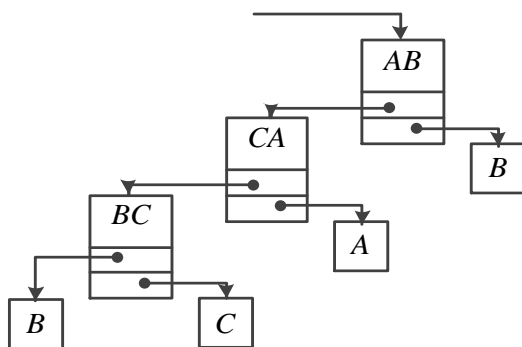


Рис.2. Пример плекса

В качестве основных алгоритмов обработки плексов в рамках настоящей лабораторной работы должны быть рассмотрены операции обхода плексов, необходимых при отрисовке чертежа. Реализация алго-

ритмов обхода плексов может быть выполнена поэтапно: на первом этапе можно реализовать частный алгоритм обхода плексов без подплексов, и только затем приступить к разработке общего алгоритма обхода.

### ***Обход плекса без подплексов***

Плексы без подплексов соответствуют представлению чертежей, которые можно нарисовать без отрыва карандаша от бумаги. Пример такого объекта – треугольник (см. рис.1). Для того чтобы нарисовать линию, необходимо знать ее начальную и конечную точки. Так как в плексе допускается вместо указателя точки хранить указатель линии, то следует найти первую линию, у которой указатель начала - точка (например, линия *BC*). Тогда конечная точка этой линии будет начальной точкой предыдущей линии (линия *CA*).

На первом этапе необходимо отыскать линию, у которой указатель начала - точка, при этом указатели на все пройденные линии нужно запомнить в стеке.

На втором этапе из стека извлекаются линии, для которых в качестве начальной точки выступает конечная точка обработанной линии. Процесс заканчивается, когда в стеке больше не содержится указателей на линию.

### ***Обход плекса с подплеками***

Плекс с подплеками служит для представления чертежей, которые нельзя нарисовать без отрыва карандаша от бумаги (рис.3). Для его представления требуется плекс с подплеками (верхняя горизонтальная линия состоит из двух отрезков *DB* и *BF*). Возможная структура плекса для чертежа показана на рис.4.

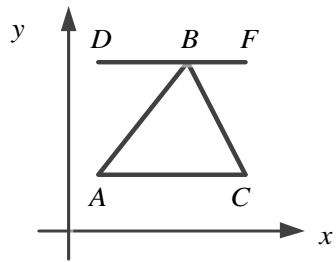


Рис.3. Пример чертежа (плекс с подплеками)

Для таких плексов указатель на конечную точку может также указывать на линию, т.е. конечная точка линии верхнего узла плекса может являться конечной точкой линии правого подплекса. Для обхода плекса с подплеками можно предложить рекурсивный и нерекурсивный варианты.

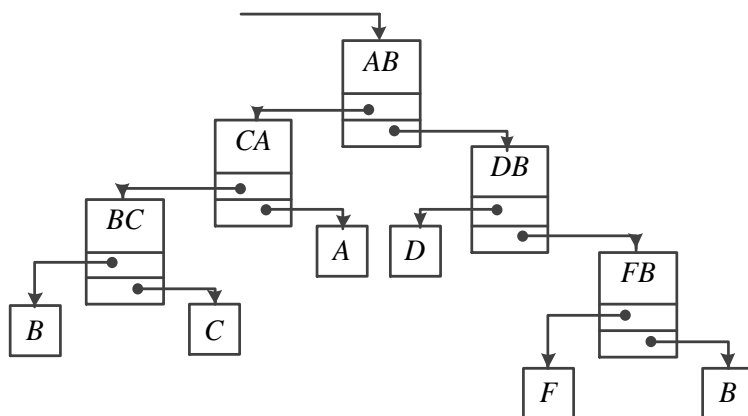


Рис.4. Пример плекса с подплеками для чертежа

Общая схема рекурсивного алгоритма состоит в следующем:

```
TChartPoint *Show ( TChart *pN ) {
if ( pN != NULL ) pL = NULL;
else if ( pN ∈ TChartPoint ) pL = pN;
else {
```

```

pF = Show(pN->GetFirstPoint());
pL = Show(pN->GetLastPoint());
// рисование линии <pN, pF, pL>
}
return pN;
}

```

- отрисовать подплекс, на который указывает указатель начальной точки; запомнить конечную точку подплекса;
- отрисовать подплекс, на который указывает указатель конечной точки; запомнить конечную точку подплекса;
- отрисовать корневую линию.

Для нерекурсивного алгоритма определим класс TChartLine для представления линии, которая подлежит отрисовке.

```

class TChartLine // класс для методов отрисовки рисунков
{
    TChart *pLine; // линия
    TChartPoint *pFp; // начальная точка
    TChartPoint *pLp; // конечная точка
    friend class TChart;
};

```

Общая схема алгоритма обхода состоит в следующем: линии, определяемые при обходе плекса, помещаются в стек.

Комментарии к схеме алгоритма.

1) При линии, извлекаемой из стека, последовательно определяются начальная и конечная точки.

2) Для определения начальной точки используется метод GetFirstPoint линии; если получаемый указатель указывает на линию, обработка текущей линии откладывается (линия помещается в стек) и начинается анализ новой линии; данная процедура выполняется итеративно до получения линии с известной начальной точкой.

3) Для определения конечной точки используется метод GetLastPoint линии; если получаемый указатель указывает на линию, обработка текущей линии снова откладывается – текущая линия помещается в стек; после этого формируется описание новой линии, которая также помещается в стек и цикл алгоритма обхода повторяется.

4) При получении линии с определенными граничными точками следует выполнить обработку линии (отрисовать); из стека извлекается новая линия, для которой конечная точка обработанной линии исполь-

зается в качестве первой неизвестной граничной точки. Выполнение обхода завершается при опустошении стека линий.

С учетом перечисленных предложений к реализации целесообразной представляется следующая структура программы:

- FTRAN.h, FTRAN.cpp - модуль с классом для формульного транслятора;
- TFormValue.h - шаблонный класс для формульных значений;
- TChartRoot.h - шаблонный базовый абстрактный класс;
- TChartPoint.h - класс для точки;
- TChartGroup.h - класс для составного геометрического объекта;
- TChart.h, TChart.cpp - модуль с классами конструируемого геометрического объекта и линии;
- TPlexTestkit.cpp - модуль программы тестирования;
- UserComm.h, UserComm.cpp - модуль функций, реализующих визуальный диалоговый интерфейс для взаимодействия с пользователем.

В схемах отношения между классами показываются следующим образом: стрелками с незакрашенным треугольником показаны отношения наследования (базовый класс - производный класс); ромбовидными стрелками - отношения ассоциации (класс-владелец – класс-компонент); обычными стрелками показано наличие поля типа указатель на другой класс.

### ***Спецификации классов***

С учетом предложенных к реализации алгоритмов спецификации классов могут состоять в следующем.

Класс **FTRAN** для формульного транслятора:

```
class FTRAN
{
protected:
unsigned char Buffer[SIZE_POST];
public:
int expression(char *text);
int computing(double vars[], double *result);
};
```

Шаблонный класс для формульных значений (**TFormValue.h**):

```
#define FormLen 40
template <class TValue>
```



```

class TFormValue: public TDatValue
{
private:
FTRAN ft; // формульный транслятор
protected:
TValue Value; // значение
char Formula[FormLen]; // формула для вычисления значения
double Param; // значение параметра формулы
virtual void Print(ostream &os);
public:
TFormValue(TValue val = 0, const char *f = "");
virtual ~TFormValue();
void SetValue(TValue val = 0, const char *f = "");
void SetFormula(const char *f = ""); // установить формулу
TValue GetValue(); // получить значение
TFormValue& operator=(const TValue &val); // перегрузка присваивания
TFormValue& operator=(const TFormValue &fval); // перегрузка =
operator TValue() const;
TValue GetValue(double par); // вычислить и вернуть значение
TDatValue* GetCopy() { return new TFormValue(Value, Formula); }
};

```

Шаблонный базовый абстрактный класс (**TChartRoot.h**):

```

class TChartRoot: public TDatValue
{
protected:
int Visible; // видимость
TFormValue<int> Active; // обрабатываются только активные объекты
virtual void Print(ostream &os) { };
public:
TChartRoot();
virtual ~TChartRoot();
int IsVisible(void) const; // проверка визуальности
int IsActive (void) const; // проверка активности
void SetActiveValue(int val = 1, char *f = NULL);
virtual void Show() = 0; // визуализация объекта
virtual void Hide() = 0; // скрытие объекта
virtual void CalcParams(double t = -1) // вычислить параметры
{ if (t >= 0) Active.GetValue(t); }
virtual void ViewTimeShot(double t = -1); // визуализация объекта
};

```

Класс **TChartPoint** для точки:

```

class TChartPoint: public TChartRoot
{
protected:
TFormValue<int> X, Y; // координаты точки
public:
TChartPoint(int a = 0, int b = 0);
virtual ~TChartPoint();
int GetValueX(void);
int GetValueY(void);
void SetValueX(int val = 0, char *f = NULL);
void SetValueY(int val = 0, char *f = NULL);
virtual void Show(); // визуализация объекта
virtual void Hide(); // скрытие объекта
virtual void CalcParams(double t = -1); // вычислить параметры
virtual TDatValue* GetCopy(); // создание копии
};

```

Класс **TChartGroup** для составного геометрического объекта:

```

class TChartGroup: public TChartRoot
{
protected:
TDatList Group; // список графических элементов группы
public:
TChartGroup() { }
virtual ~TChartGroup();
void InsUnit(TChartRoot *pUnit); // вставить объект в группу
virtual void Show(); // визуализация объекта
virtual void Hide(); // скрытие объекта
virtual void CalcParams(double t = -1); // вычислить параметры
virtual TDatValue* GetCopy(); // создание копии
};

```

Класс **TChart** для конструируемого геометрического объекта:

```

class TChart;
class TChartLine // класс для методов отрисовки рисунков
{
TChart *pLine; // линия
TChartPoint *pFp; // начальная точка
TChartPoint *pLp; // конечная точка
friend class TChart;
};
class TChart: public TChartGroup
{

```

```

protected:
stack<TChartLine> St;
public:
TChart() {}
virtual ~TChart() {}
TChartRoot *GetFirstPoint(void); // получить начальную точку
TChartRoot *GetLastPoint(void); // получить конечную точку
void SetFirstPoint(TChartRoot *pUnit); // вставить начальную точку
void SetLastPoint (TChartRoot *pUnit); // вставить конечную точку
virtual void Show(); // визуализация рисунка
virtual void Hide(); // скрытие рисунка
};

```

## Лабораторное задание

Алгоритм проведения настоящей лабораторной работы следующий.

Выполнение лабораторной работы должно быть разделено на несколько этапов. Каждый из этапов должен иметь достаточно небольшую длительность, при этом должна обеспечиваться возможность решения поставленной прикладной задачи (в той или иной ограниченной постановке) на как можно более ранних этапах выполнения работ.

**Рекомендации по разработке.** Набор программ для работы с плексами представляет собой достаточно сложный программный модуль. При выполнении настоящей лабораторной работы следует уделять особое внимание технологическим аспектам разработки программного обеспечения: этапность разработки, тщательное тестирование, командная методика выполнения работ, обеспечение простого и дружелюбного интерфейса, использование инструментов поддержки процесса разработки (система контроля версий, система поддержки тестирования и др.) (варианты заданий).

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

- реализовать построение чертежа с использованием дополнительных геометрических элементов (окружностей и их дуг, прямоугольников и др.);
- реализовать проверку тождественности чертежей для разных схем структур данных.

**Критерии оценивания лабораторной работы.** Лабораторная работа считается выполненной при выполнении следующего минимального набора требований:

- реализованы все классы, разработанные программы обеспечивают демонстрацию методов отрисовки плекса без подплексов, данные для построения считываются из файла (в качестве примера может быть использован чертеж, представленный на рис.1);
- в расширенном варианте выполнения лабораторной работы должен быть реализован обход плекса с подплексами, организован визуальный пользовательский интерфейс с функциями чтения-записи данных о плексе в файл, обеспечена поддержка анимации и др.

С учетом перечисленных рекомендаций разработка программ должна быть групповой (три-четыре человека). Варианты заданий 1-9 приведены на с.97.

**Таблица 1**

Варианты заданий

Вариант	Задание
1	Реализовать программу для работы с плексами простой структуры (для одной, двух или трех линий)
2	Реализовать программу для работы с составными геометрическими объектами (поддержка группирования)
3	Реализовать программу отрисовки чертежа на основе алгоритма обхода плексов без подплексов. Этап должен включать развитие диалоговой формы управления
4	Разработать программу для операций динамического изменения (вставка и удаление линий) для чертежа (плекса)
5	Разработать программу представления арифметических выражений с использованием плексов
6	Реализовать программу операций чтения и записи информации в файл
7	Реализовать программу общего обхода плекса
8	Реализовать программу построения схемы линейного, ветвящегося или циклического алгоритма
9	Реализовать программу дополнительных операций обхода плексов

Результаты выполнения каждого этапа требуют проведения тщательного тестирования, поскольку возможные ошибки могут проявляться только после некоторого длительного времени работы с плексами;

ошибочные ситуации могут возникать не при каждом сценарии выполнения программ.

### **Контрольные вопросы**

1. Какие возможные модели можно использовать для представления чертежа?
2. В чем состоит модель чертежа?
3. Какая структура хранения используется для чертежа?
4. Какие преимущества и недостатки есть у рекурсивного и нерекурсивного алгоритмов обхода плекса?
5. Как можно модифицировать алгоритм обхода плекса для проверки чертежа на связность?

## Лабораторная работа № 7

### Эвристические алгоритмы

**Цель работы:** ознакомление с эвристическими алгоритмами и методикой оценки их эффективности.

**Продолжительность работы:** 2 часа.

#### Теоретические сведения

Эвристический алгоритм - это алгоритм, в котором на определенном этапе используется интуиция разработчика. Если решение, принятое разработчиком, окажется неверным, результат все равно будет получен, но за большее число шагов. Таким образом, в эвристических алгоритмах можно увеличить скорость получения правильного результата.

К эвристическим алгоритмам относятся волновой, двухлучевой, четырехлучевой, маршрутный, алгоритмы составления расписания.

#### Волновой алгоритм

Волновой алгоритм, или алгоритм Ли, первоначально использовался для поиска пути в лабиринте или в игровых задачах. В настоящее время алгоритм Ли (волновой) является основным в микроэлектронике для трассировки, или соединения элементов интегральных схем (ИС). Особенность алгоритма состоит в следующем.

	1	
1	A	1
	1	

Рис.1. Направления распространения волны (первый шаг)

В лабиринте (на подложке ИС) выбираются две точки: *A* (начальная) и *B* (конечная). Из начальной точки в четырех направлениях выходит волна (рис.1). Цифрами обозначается номер фронта волны или ее путевые координаты.

Путевые координаты определяют шаг распространения волны. Каждый элемент первого фронта волны является источником вторичной волны (рис.2).

Элементы второго фронта генерируют третий фронт и т.д. От запрещенных элементов волна не распространяется. Процесс продолжается до тех пор, пока не будет достигнут конеч-

ный элемент. Траектория пути (трасса) определяется обратным просмотром, от конечного элемента к начальному. При этом разработчик задает приоритеты движения: **вверх, вниз, влево, вправо**.

		2		
	2	1	2	
2	1	<i>A</i>	1	2
	2	1	2	
		2		

Рис.2. Второй шаг волнового алгоритма

От того в каком порядке заданы приоритеты, зависит скорость решения задачи.

При построении траектории используются два принципа:

- 1) движение осуществляется строго по заданным приоритетам;
- 2) при построении трассы, т.е. траектории движения, значения путейых координат должны уменьшаться.

**Пример.** Пусть задан лабиринт, где запрещенные элементы имеют сплошную черную заливку (рис.3). Найти путь (трассу) между элементами *A* и *B*.

6		10	9	8		10			
5				7		9	10		
4			5	6		8	<b>9</b>	<i>B</i>	
3	2		4	5		<b>7</b>	<b>8</b>		10
	1	2	3	4		<b>6</b>			9
1	<i>A</i>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	6	7	8
2	1		3				7		
3	2		4	5	6		8	9	10

Рис.3. Нахождение пути в лабиринте

На первом этапе от элемента *A* распространяется волна до тех пор, пока она не достигнет конечного элемента *B*. На втором этапе выбира-

ются приоритеты движения от конечной точки  $B$  к начальной  $A$ , исходя из взаимного расположения начального и конечного элемента. Предположим, что выбраны парадоксальные (логически неверные) приоритеты: вверх, вправо, вниз, влево. В этом случае трасса все равно будет построена, но за большее число шагов (сравнений) (см. рис.3).

### Двухлучевой алгоритм

В двухлучевом алгоритме из начального и конечного элементов одновременно выходят по два луча. Трасса считается проведенной, если пересекаются два разноименных луча (от разных источников). Если на пути луча встречается запрещенный элемент, то его обход осуществляется по второму приоритетному направлению, характерному для лучей, выходящих из одной точки. Если же оба направления оказываются заблокированными запрещенными элементами либо достигнут край координатной сетки, то движение луча прекращается.

Существуют четыре варианта распространения лучей (рис.4). Коэффициенты  $\alpha$  и  $\beta$  вычисляются следующим образом:

$$\alpha = \begin{cases} 1, & \text{если } X_A - X_B \geq 0, \\ 0, & \text{если } X_A - X_B < 0; \end{cases} \quad \beta = \begin{cases} 1, & \text{если } Y_A - Y_B \geq 0, \\ 0, & \text{если } Y_A - Y_B < 0, \end{cases}$$

где  $(X_A, Y_A)$  - координаты начального элемента;  $(X_B, Y_B)$  - координаты конечного элемента.

Исходя из значений  $\alpha$  и  $\beta$  выбирается вид распространения лучей (рис.4).

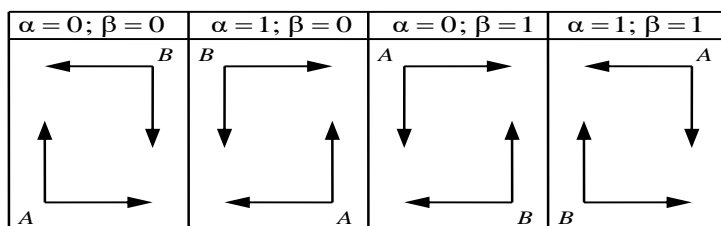


Рис.4. Варианты распространения лучей

**Пример.** Осуществить трассировку элементов  $A$  и  $B$  двухлучевым алгоритмом (рис.5). Вычислим коэффициенты  $\alpha$  и  $\beta$ :

$$\alpha = 3 - 10 = 7; \quad \beta = 9 - 2 = 7.$$



Принимаем  $\alpha = 0$ ;  $\beta = 1$ , после чего из рис.4 выбираем направление распространения лучей. Номер путевой координаты с символом \* означает, что дальнейшее распространение луча невозможно.

	A	1	2	3		5*			
	1			4		4			
	2			5	6	3			
	3					2			
	4*					1			
			3*	2	1	B			

Рис.5. Двухлучевой алгоритм

### Четырехлучевой алгоритм

В четырехлучевом алгоритме из начальной и конечной точек одновременно выходят по четыре луча (рис.6). Лучи движутся строго по заданным направлениям и «затухают» (прекращают движение), если достигли края координатной сетки либо встретили запрещенный элемент.

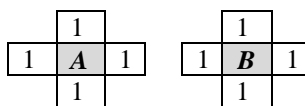


Рис.6. Распространение лучей в четырехлучевом алгоритме

**Пример 3.** Осуществить трассировку элементов *A* и *B* четырехлучевым алгоритмом (рис.7).

	2*					
	1					
1*	A					
	1			1*		
	2	2	1	B	1	2*

Рис.7. Четырехлучевой алгоритм

Все четыре луча «затухли», значит трассу данным алгоритмом построить нельзя.

## Маршрутный алгоритм

Маршрутный алгоритм получил свое название потому, что осуществляет одновременно и формирование фронта и прокладывание трассы. Источником анализируемых элементов на каждом шаге является конечный элемент участка трассы, проложенной на предыдущем шаге. В маршрутном алгоритме рассматривается восьмиэлементная окрестность исходного элемента (рис.8).

$i-1, j-1$	$i, j-1$	$i+1, j-1$
$i-1, j$	<b>A</b>	$i+1, j$
$i-1, j+1$	$i, j+1$	$i+1, j+1$

Рис.8. Восьмиэлементная окрестность

От каждого элемента окружения оценивается расстояние  $d$  до конечного элемента  $B$ :

$$d = \sqrt{(x_i - x_b)^2 + (y_i - y_b)^2}$$

или  $d = |x_i - x_b| + |y_i - y_b|$ .

Таким образом определяется восемь значений расстояний, из которых выбирается минимальное. Элемент, для которого значение  $d$  оказалось минимальным, считаем элементом трассы. Процесс повторяется до тех пор, пока расстояние не будет равным нулю ( $d = 0$ ), т.е. пока не будет достигнут конечный элемент. Обход запрещенных элементов осуществляется исходя из интуиции разработчика.

**Пример.** Осуществить трассировку элементов  $A$  и  $B$  маршрутным алгоритмом (рис.9).

Обозначим элементы, анализируемые в процессе построения трассы, числами от 1 до 10. Эти элементы попадают в восьмиэлементные окрестности исходного и помеченных на рис.9 элементов.

			9	<b>B</b>		
1	2		7			
3	<b>A</b>	4	8	10		
5		6				

Рис.9. Маршрутный алгоритм

Найдем расстояния  $d$  от элементов, обозначенных числами, до конечного  $B$ :

$$\begin{aligned}
d_1 &= \sqrt{(1-5)^2 + (4-5)^2} = \sqrt{17}; & d_7 &= \sqrt{2}; \\
d_2 &= \sqrt{(2-5)^2 + (4-5)^2} = \sqrt{10}; & d_8 &= \sqrt{5}; \\
d_3 &= \sqrt{(1-5)^2 + (3-5)^2} = \sqrt{20}; & d_9 &= 1; \\
d_4 &= \sqrt{(3-5)^2 + (3-5)^2} = \sqrt{8}; & d_{10} &= 2; \\
d_5 &= \sqrt{(1-5)^2 + (2-5)^2} = \sqrt{25} = 5; & \min &= d_7; \\
d_6 &= \sqrt{(3-5)^2 + (2-5)^2} = \sqrt{13}; & d_B &= 0. \\
\min &= d_4;
\end{aligned}$$

Результаты трассировки элементов  $A$  и  $B$  маршрутным алгоритмом:  
( $A - 4 - 7 - B$ ).

### ***Алгоритм составления расписания***

Предположим, что имеется множество  $n$  одинаковых процессоров, обозначенных  $P_1, P_2, \dots, P_n$ , и  $m$  независимых заданий  $J_1, J_2, \dots, J_m$ , которые нужно выполнить. Процессоры могут работать одновременно, и любое задание можно выполнять на любом процессоре. Если задание загружено в процессор, то оно остается там до конца обработки. Время обработки задания  $J_i$  известно и равно  $t_i$  ( $i=1, 2, \dots, m$ ). Необходимо организовать обработку заданий таким образом, чтобы выполнить весь набор заданий как можно быстрее.

Система работает следующим образом: первый освободившийся процессор берет из списка следующее задание. Если одновременно освобождаются два или более процессоров, то выполнять очередное задание из списка будет процессор с наименьшим номером.

**Пример.** Пусть имеется три процессора и шесть заданий, время выполнения каждого из которых равно:

$$\begin{aligned}
t_1 &= 2; \quad t_2 = 5; \quad t_3 = 8; \\
t_4 &= 1; \quad t_5 = 5; \quad t_6 = 1.
\end{aligned}$$

Рассмотрим расписание  $L = (J_2, J_5, J_1, J_4, J_6, J_3)$ . В начальный момент времени  $T = 0$  процессор  $P_1$  начинает обработку задания  $J_2$ , процессор  $P_2$  - задания  $J_5$ , а процессор  $P_3$  - задания  $J_1$ .

Процессор  $P_3$  заканчивает выполнение задания  $J_1$  в момент времени  $T=2$  и начинает обрабатывать задание  $J_4$ , в то время как процессоры  $P_1$  и  $P_2$  продолжают работать над первоначальными заданиями. При  $T=3$  процессор  $P_3$  заканчивает задание  $J_4$  и начинает обрабатывать задание  $J_6$ , которое завершается в момент  $T=4$ . Тогда он приступает к выполнению последнего задания  $J_3$ .

Процессоры  $P_1$  и  $P_2$  заканчивают задания при  $T=5$ , но, так как список  $L$  пуст, они останавливаются. Процессор  $P_3$  завершает выполнение задания  $J_3$  при  $T=12$ . Очевидно, что расписание неоптимальное.

Используя диаграмму Ганта, можно подобрать расписание  $L^* = (J_3, J_2, J_5, J_1, J_4, J_6)$ , которое позволяет завершить все задания за  $T^* = 8$  единиц времени (рис.10).

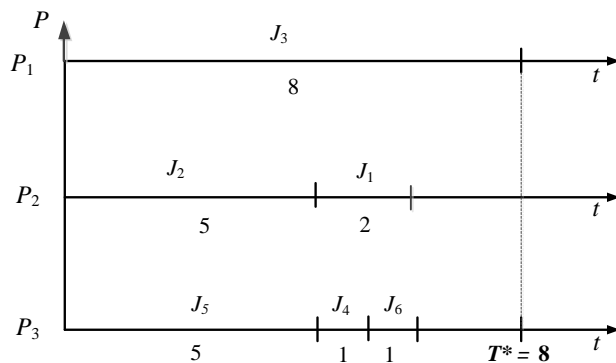


Рис.10. Схема Ганта для оптимального расписания  $L^*$

Рассмотрим другой тип задач по составлению расписания для многопроцессорных систем. Вместо вопроса о быстрейшем завершении набора заданий фиксированным числом процессоров поставим вопрос о минимальном числе процессоров, необходимых для завершения данного набора заданий за фиксированное время  $T_0$ . Конечно, время  $T_0$  будет не меньше времени выполнения самого трудоемкого задания.

В такой формулировке задача составления расписания эквивалентна задаче упаковки. Пусть каждому процессору  $P_j$  соответствует ящик  $B_j$  размером  $T_0$ . Пусть каждому заданию  $J_i$  соответствует предмет размером  $t_i$ , равным времени выполнения задания  $J_i$ , где  $i = 1, 2, \dots, n$ . Для решения задачи по составлению расписания нужно построить алгоритм, позволяющий разместить все предметы в минимальном количестве ящиков. Конечно, нельзя заполнять ящики сверх их объема  $T_0$ , и нельзя предметы дробить на части.

### Лабораторное задание

Алгоритм выполнения лабораторной работы следующий.

1. Ознакомиться с эвристическими алгоритмами.
2. Осуществить трассировку элементов интегральных схем размером  $10 \times 10$ ,  $20 \times 20$ ,  $30 \times 30$ .
3. Зафиксировать параметры трассировок всеми рассмотренными методами с использованием обучающе-контролирующих программ:

Путь к файлу: D:\ИПОВС\АиСД\HEVRIST\Hevrist1

Путь к файлу: D:\ИПОВС\АиСД\HEVRIST\Hevrist2

Путь к файлу: D:\ИПОВС\АиСД\HEVRIST\Hevrist3

Путь к файлу: D:\ИПОВС\АиСД\HEVRIST\Hevrist4

Работа осуществляется в диалоговом режиме с использованием меню. Вся необходимая поясняющая информация отображается во время работы системы на экране монитора. В обучающе-контролирующей программе варианты заданий генерируются ЭВМ.

4. Составить оптимальное расписание работы четырех процессоров, для которых известно  $t_1, \dots, t_{11}$ .

5. Составить алгоритм оптимальной упаковки 12 предметов размером от 1 до 4 в ящики размером 6.

6. Составить программу моделирования эвристического алгоритма (по заданию преподавателя).

### Контрольные вопросы

1. Какова теоретическая сложность алгоритмов, рассмотренных в настоящей лабораторной работе?
2. В чем состоят особенности работы волнового, лучевых и маршрутного алгоритмов?
3. Каковы принципы составления оптимального расписания работы параллельных процессоров?
4. В чем заключаются особенности задачи упаковки?

## Библиографический список

1. *Ахо А., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. – М.: СПб.: Киев: ИД Вильямс, 2001.
2. *Вирт Н.* Алгоритмы и структуры данных / пер. с англ. – М.: Мир, 2001.
3. *Гагарина Л.Г., Колдаев В.Д.* Алгоритмы и структуры данных: учеб. пособие. – М.: Финансы и статистика; Инфра-М, 2009.
4. *Кнут Д.* Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. – М.: Мир, 2000.
5. *Колдаев В.Д.* Основы алгоритмизации и программирования: учеб. пособие / под ред. Л.Г. Гагариной. - М.: ИД «ФОРУМ» – Инфра-М, 2009.
6. *Колдаев В.Д.* Численные методы и программирование: учеб. пособие. / Под ред. Л.Г. Гагариной. - М.: ИД «ФОРУМ» – Инфра-М, 2010.
7. *Колдаев В.Д., Лупин С.А.* Архитектура ЭВМ: учеб. пособие. – М.: ИД «ФОРУМ» – Инфра-М, 2009.
8. *Колдаев В.Д.* Основы логического проектирования: учеб. пособие. – М.: ИД «ФОРУМ» – Инфра-М, 2011.
9. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. – М.: МЦНМО, 2000.
10. *Мейн М., Савитч У.* Структуры данных и другие объекты в C++ / пер. с англ. – М.: ИД Вильямс, 2002.
11. *Хусаинов Б.С.* Структуры и алгоритмы обработки данных. Примеры на языке Си: учеб. пособие. – М.: Финансы и статистика, 2004.
12. *Шень А.* Программирование. Теоремы и задачи. – М.: МЦНМО, 2004.

## Приложение

### Решение логических задач

**Пример 1.** Дан текстовый файл размером не более 64 Кб, содержащий действительные числа, по одному в каждой строке. Переписать содержимое файла в массив, разместив его в динамически распределяемой памяти; вычислить среднее значение элементов массива; очистить динамическую память; создать целый массив размером 10000, заполнить его случайными целыми числами в диапазоне от  $-100$  до  $100$  и вычислить его среднее значение.

#### Программа на языке C++

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <iostream.h>
#define NMax 10000
typedef int MasInt;
typedef float MasReal;
MasInt *PInt; MasReal *PReal;
int I, n, MidInt; float MidReal; char S[255];
FILE *t; char *endptr;
void main()
{ cout << "Введите имя файла: "; cin >> S;
  t=fopen(S, "r");
  MidReal = 0; MidInt = 0;
  randomize(); I=0;
  /*Выделение памяти под вещественный массив*/
  PReal = (MasReal*) malloc (sizeof(MasReal));
  /*Ввод и суммирование вещественного массива*/
  while (!feof(t))
  {fgets(S, 255, t); // вводим из файла строку
   PReal[I] = strtod(S, &endptr); /*Преобразование строки в
    вещественное число*/
    MidReal += PReal[I]; I++;}
  n=I+1;
  free (PReal); /*Удаление вещественного массива*/
  PInt = (MasInt*) malloc(sizeof(MasInt)); /*Выделение памяти
   под целый массив*/
  /* Вычисление и суммирование целого массива */
```



```

for (I=0; I < NMax; I++)
{ PInt[I] = -100 + random(201);
  MidInt += PInt[I];}
/*Вывод средних значений*/
cout << "\nсреднее целое равно " << MidInt / double(NMax) << "\n";
cout << "среднее вещественное равно: " << MidReal / n << "\n";
fclose(t);
}

```

**Пример 2.** Составить программу, которая на основе заданного списка формирует два других, помещая в первый из них положительные, а во второй - отрицательные элементы исходного списка.

**Программа на языке C++**

```

#include "SPIS.CPP"
void main()
{Zveno *S1, *S2, *S3, *V1, *V2, *V3;
 BT a; int i, n;
 clrscr();
 randomize();
 S1=NULL;
 // создаем первый элемент
 a=-100+random(201);
 S1=V_Nachalo(S1, a);
 n=1+random(20);
 // формируем список произвольной длины и выводим на печать
 V1=S1;
 for (i=2; i<=n; i++)
 {
  a=-100+random(201);
  V1=V_Spisok(V1, a);
 }
 Print(S1);
 V1 = S1; S2 = NULL; S3 = NULL;
 while (V1)
 {if (V1->Inf > 0)
  if (!S2)
   {S2=V_Nachalo(S2, V1->Inf); V2 = S2;}
  else {V_Spisok(V2, V1->Inf); V2 = V2->Next;};
 if (V1->Inf < 0)
  if (!S3)
   {S3=V_Nachalo(S3, V1->Inf); V3 = S3;}
  else {V_Spisok(V3, V1->Inf); V3 = V3->Next;};
 V1= V1->Next;}
}

```

```

cout << "Результирующий список из положительных элементов: \n";
Print(S2);
cout << "Результирующий список из отрицательных элементов: \n";
Print(S3);
S1=Ochistka(S1); S2=Ochistka(S2); S3=Ochistka(S3);
}

```

**Пример 3.** Написать программу, которая вычисляет как целое число значение выражений (без переменных), записанных (без ошибок) в постфиксной форме в текстовом файле. Каждая строка файла содержит ровно одно выражение.

*Алгоритм решения.* Выражение просматривается слева направо. Если встречается число, то его значение (как целое) заносится в стек, а если встречается знак операции, то из стека извлекаются два последних элемента (это операнды данной операции), над ними выполняется операция и ее результат записывается в стек. В конце в стеке остается только одно число - значение всего выражения.

#### Программа на языке C++

```

#include "STACK.CPP"
#include < string.h >
#include < stdio.h >
void main(void)
{
    char S[255];
    FILE *T;
    int I; BT X, Y;
    Zveno *NS;
    clrscr();
    cout << "Введите имя файла: ";
    cin >> S;
    T=fopen(S, "r");
    NS = NULL;
    while (!feof(T))
    {
        fgets(S, 255, T);
        I = 0;
        while (I <= strlen(S)-1)
        {
            If (S[I]>='0'&&S[I]<='9')
            {
                X=0;
                while(S[I]>='0'&&S[I]<='9') {X=X*10+(S[I]-'0'); I++;}
                NS=V_Stack(NS, X);
            }
        }
    }
}

```

```

    }
    else
if(S[I]=='+'||S[I]=='-'||S[I]=='/'||S[I]=='*')
{
    X=V_Vershine(NS);
    NS=Iz_Stack(NS);
    Y=V_Vershine(NS);
    NS=Iz_Stack(NS);
    switch (S[I]) {
    case '+': X += Y; break;
    case '-': X = Y - X; break;
    case '*': X *= Y; break;
    case '/': X = Y / X; break;}
        NS=V_Stack(NS, X);
    }
    I++;
}
X=V_Vershine(NS);
NS=Iz_Stack(NS);
cout << S << " => " << X << "\n";}
}

```

**Пример 4.** Напечатать в порядке возрастания первые  $n$  натуральных чисел, в разложение которых на простые множители входят только числа 2, 3, 5.

*Алгоритм решения.* Введем три очереди  $X_2, X_3, X_5$ , в которых будем хранить элементы, которые соответственно в 2, 3, 5 раз больше напечатанных, но еще не напечатаны. Рассмотрим наименьший из ненапечатанных элементов; пусть это  $X$ . Тогда он делится нацело на одно из чисел 2, 3, 5. Элемент  $X$  находится в одной из очередей и, следовательно, является в ней первым (меньшие напечатаны, а элементы очередей не напечатаны). Напечатав  $X$ , нужно его изъять и добавить его кратные. Длины очередей не превосходят числа напечатанных элементов.

**Программа на язык C++**

```

#include "spis2.cpp"
void PrintAndAdd(BT T);
BT Min (BT A, BT B, BT C);
U * X2, *X3, *X5;
void main ()
{ BT X; long I, N;
  X2 = NULL; X3 = NULL; X5 = NULL;
  cout << "Сколько чисел напечатать? "; cin >> N;
  PrintAndAdd(1);

```

```

for (I=1;I<=N; I++)
{ X = Min(X2->Inf, X3->Inf, X5->Inf);
  PrintAndAdd(X);
  if (X==X2->Inf) X2=Iz_Och(X2, X);
  if (X==X3->Inf) X3=Iz_Och(X3, X);
  if (X==X5->Inf) X5=Iz_Och(X5, X);
}
X2=Ochistka(X2); X3=Ochistka(X3); X5=Ochistka(X5);
cout << endl;
}
void PrintAndAdd(BT T)
{ if (T!=1) {cout.width(6); cout << T;}
  X2=V_Och(X2, T*2);
  X3=V_Och(X3, T*3);
  X5=V_Och(X5, T*5);
}
BT Min (BT A, BT B, BT C)
{ BT vsp;
  if (A < B) vsp=A; else vsp=B;
  if (C < vsp) vsp=C;
  return vsp;
}

```

**Пример 5. (Задача о джипе).** Пусть необходимо пересечь на джипе 1000-километровую пустыню, израсходовав при этом минимум горючего. Объем топливного бака джипа 500 л, горючее расходуется равномерно, по одному литру на километр. При этом в точке старта имеется неограниченный резервуар с топливом. Так как в пустыне нет складов с горючим, необходимо установить собственные хранилища и наполнять их топливом из бака машины. Конечно, проще было бы ехать на грузовике, загруженном бочками с бензином, но тогда не было бы задачи о джипе.

Нужно из точки старта отъезжать с полным баком на некоторое расстояние, устраивать там первый склад, оставлять там какое-то количество горючего из бака, но такое, чтобы его хватило на возвращение.

В точке старта вновь производится полная заправка и делается попытка второй склад продвинуть в пустыню дальше. Но где устраивать эти склады и сколько горючего оставлять в каждом из них?

**Пример 6. (Задача о кодовом замке).** Пусть кодовый замок состоит из набора  $N$  переключателей, каждый из которых может быть в положе-

нии «вкл» или «выкл». Замок открывается только при одном наборе положений переключателей, из которых не менее  $\lceil N/2 \rceil$  (целая часть  $N/2$ ), находятся в положении «вкл». Построить алгоритм перебора комбинаций, чтобы не пропустить нужную и не набирать ту, которая заведомо к успеху не приведет.

Промоделируем каждую возможную комбинацию вектором из  $N$  нулей и единиц. На  $i$ -м месте будет 1, если  $i$ -й переключатель находится в положении «вкл», и 0, если  $i$ -й переключатель - в положении «выкл». Множество всех возможных  $N$ -векторов моделируется с помощью бинарного (или двоичного) дерева. Если количество переключателей в замке равно  $N$ , то в дереве просмотра будет  $N$  уровней. Решить задачу для  $N = 4$ .

## Содержание

Предисловие.....	
Лабораторная работа № 1. Методы сортировки .....	
Лабораторная работа № 2. Методы поиска .....	
Лабораторная работа № 3. Итеративные и рекурсивные алгоритмы.....	
Лабораторная работа № 4. Алгоритмы построения остовного (покрывающего) дерева сети .....	
Лабораторная работа № 5. Алгоритмы нахождения кратчайших путей на графах.....	
Лабораторная работа № 6. Обработка геометрических объектов с помощью плексов.....	
Лабораторная работа № 7. Эвристические алгоритмы.....	
Библиографический список .....	
Приложение. Решение логических задач.....	

Учебное издание

**Колдаев** Виктор Дмитриевич

**Лабораторный практикум по курсу «Алгоритмы и структуры данных».**

**Часть 1**

Редактор *Н.А. Кузнецова*. Технический редактор *Е.Н. Романова*. Корректор *Л.Г. Лосякова*. Верстка автора.

Подписано в печать с оригинал-макета 10.10.2019. 2019. Формат 60×84 1/16.

Печать офсетная. Бумага офсетная. Гарнитура Times New Roman.

Усл. печ. л. 6,73. Уч.-изд. л. 5,8. Тираж 200 экз. Заказ 58.

Отпечатано в типографии ИПК МИЭТ.

124498, г. Москва, Зеленоград, площадь Шокина, дом 1, МИЭТ.