Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Московский государственный институт электронной техники
(технический университет)

В.Д. Колдаев

Лабораторный практикум по курсу "Структуры и алгоритмы обработки данных"

Часть 1

Утверждено редакционно-издательским советом института
в качестве методических указаний

Москва 2006

УДК 004.42(075.8)

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

Колдаев В.Д.

Лабораторный практикум по курсу "Структуры и алгоритмы обработки данных". Часть 1. - М.: МИЭТ, 2006. - 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. В конце каждой лабораторной работы даны варианты заданий и задачи для самостоятельного решения. Выполнение лабораторных работ предусматривает использование компьютерных обучающе-контролирующих программ, разработанных на кафедре ИПОВС.

Практикум предназначен для студентов всех специальностей МИЭТ.

Колдаев *Виктор Дмитриевич*
Лабораторный практикум по курсу "Структуры и алгоритмы обработки данных". Часть 1.
Редактор *Л.М. Рогачева*. Технический редактор *Л.Г. Лосякова.* Верстка автора.
Подписано в печать с оригинал-макета 30.06.06. Формат 60x84 1/16. Печать офсетная. Бумага офсетная. Гарнитура Times New Roman. Усл. печ. л. 6,73.
Уч.-изд. л. 5,8. Тираж 200 экз. Заказ 114.
Отпечатано в типографии ИПК МИЭТ.
124498, Москва, Зеленоград, проезд 4806, д. 5, МИЭТ

Лабораторная работа № 1. Методы сортировки

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

Продолжительность работы - 2 ч.

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).

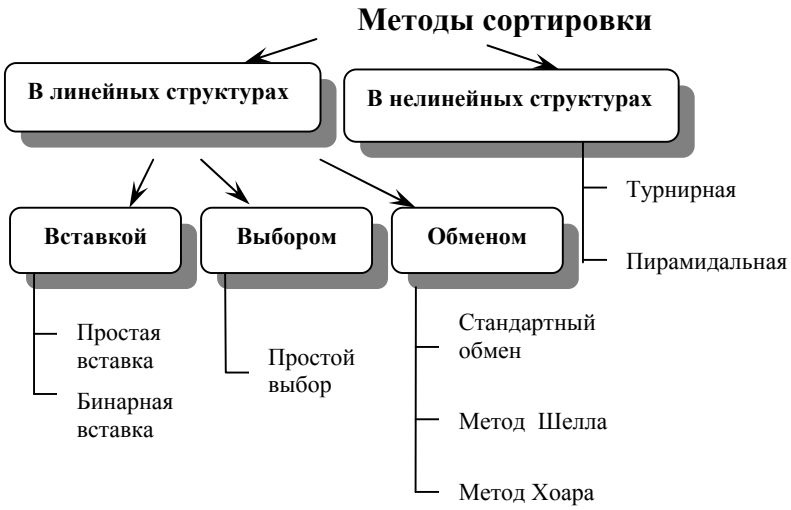


Рис.1. Классификация методов сортировки

Обычно сортируемые элементы множества называют записями и обозначают k_1, k_2, …,k_n.

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 строчке выписаны сравниваемые пары. Выбираемые элементы с меньшим весом обведены кружком. Нетрудно видеть, что число сравнений соответствует на рисунке числу строк, а число перемещений - количеству изменений выбранного элемента.
{5, 11, 6, 4, 9, 2, 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 нового списка (*i* = 1, 2, …., *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, …., *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 1 элемент меньше предыдущего;
- выбранный минимальный элемент, как и в предыдущем случае, перемещается на *i*-е место заданного списка, а чтобы это *i*-е место освободилось для записи очередного минимального элемента, левая от выбранного элемента часть списка перемещается вправо на 1 позицию так, чтобы заполнилось место, занимаемое до этого выбранным элементом.

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 кружком на каждом этапе обведен анализируемый элемент, стрелкой сверху отмечено место перемещения анализируемого элемента, в рамку заключены упорядоченные части последовательности.

На первом шаге 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 втором шаге 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 (седьмом) шаге.

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 после-

довательно сравнивается с элементами из другого массива. В новый массив после

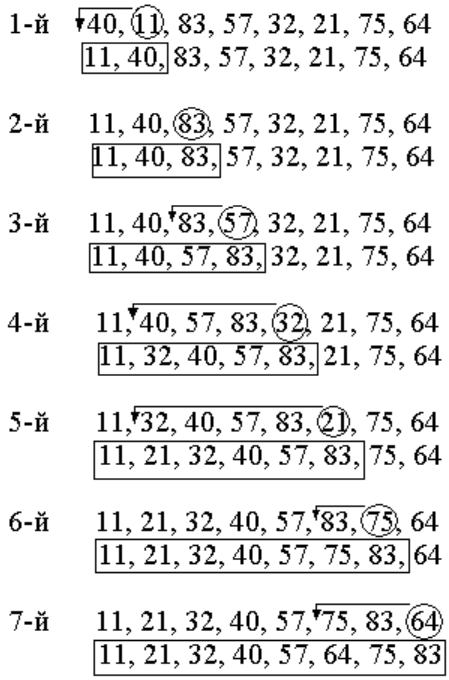


Рис.3. Последовательность шагов сортировки вставкой

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 исходные.

Пусть имеются два отсортированных в порядке возрастания массива $p[1], p[2], ..., p[n]$ и $q[1], q[2], ..., q[n]$ и имеется пустой массив $r[1], r[2], ..., r[2xn]$, который нужно заполнить значениями массивов p и q в порядке возрастания. Для слияния выполняются следующие действия: сравниваются $p[1]$ и $q[1]$, и меньшее из значений записывается в $r[1]$. 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 4 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.

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

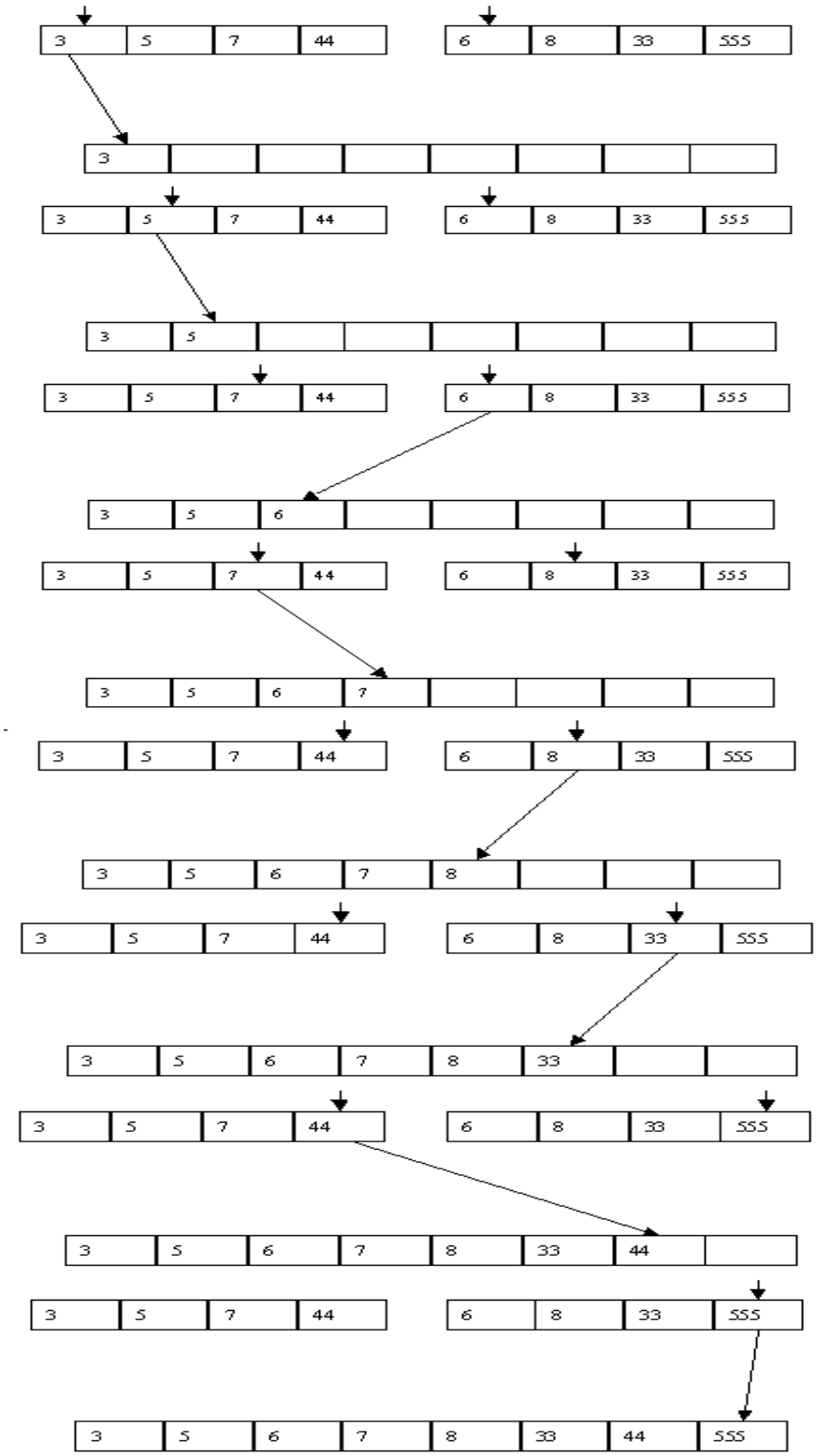


Рис.4. Сортировка слиянием (фон Неймана)

Сортировка обменом

В сортировке обменом элементы списка последовательно сравниваются между собой и меняются местами в том случае, если предшествующий 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 ⮍ обмениваемые элементы, а ⌴ - сравниваемые элементы. Первый этап сортировки показан на рис.5.

Исходный список	40	11	83	57	32	21	75	64
Первый просмотр	11	40 40	83 57	83 32	83 21	83 75	83 64	83
Получен-ный спи-сок	11	40	57	32	21	75	64	⑧③

Рис.5. Сортировка обменом (первый просмотр)

Нетрудно видеть, что после каждого просмотра списка все элементы, начиная с последнего, занимают свои окончательные позиции, поэтому их не следует проверять при следующих просмотрах. Каждый последующий просмотр исключает очередную позицию с найденным максимальным элементом, тем самым укорачивая список. После первого просмотра в последней позиции оказался больший элемент, равный 83 (исключается из дальнейшего рассмотрения). Второй просмотр выявляет максимальный элемент, равный 75 (рис.6).

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

Исходный список	11	40	57	32	21	75	64
Второй просмотр	11	40 40	57 32	57 21	57 57	75 64	75
Полученный список	11	40	32	21	57	64	⑦⑤

Рис.6. Сортировка обменом (второй просмотр)

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 элемен-

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

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

Сортировка Шелла

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

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

Первоначальный шаг вычисляется по формуле, т.е. $d = [n/2] = 4$.

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

Рис.7. Метод Шелла (шаг $d = 4$)

При первом просмотре сравниваются элементы, отстоящие друг от друга на $d = 4$ (рис.7), 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 выбирается шаг $d = [d/2] = 2$ (рис.8). Затем выбирается шаг $d = [d/2] = 1$ (рис.9), т.е. налицо аналогия с методом стандартного обмена.

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

Исходный массив	32	11	75	57	40	21	83	64
$d = 2$	32	11	40	21	75	57	83	64
Полученный массив	32	11	40	21	75	57	83	64

Рис.8. Метод Шелла (шаг $d = 2$)

Исходный список	32	11	40	21	75	57	83	64
$d = 1$	11 32	32 40	21 40	40 75	57 75	75 83	64 83	
Полученный список	11	32	21	40	57	75	64	(83)

Рис.9. Метод Шелла (шаг $d = 1$)

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 ($i = 0$), а j - справа ($j = n$).

Сравниваем базовый ключ k_0 и текущий ключ k_j. Если $k_0 <= k_j$, то устанавливаем $j = j - 1$ и проводим следующее сравнение 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 (шаг 3 на рис.10).

Номер шага	i →				← 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

Теперь начинаем изменять индекс $i = i + 1$ и сравнивать элементы k_i и k_0. Продолжаем увеличение i до тех пор, пока не получим условие $k_i > k_0$, после чего следует обмен k_i и k_0 (шаг 5 на рис.10). Снова возвращаемся к индексу j, уменьшаем его. Чередуя уменьшение j и увеличение i, продолжаем этот процесс с обоих концов к середине до тех пор, пока не получим $i = j$ (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 **40** **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

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.

Пример. Дано исходное множество {7, 1, 9, 3, 6, 5, 8}.

Производится попарное сравнение элементов снизу вверх.

Найденный максимальный элемент помещается в результирующее множество (рис.11).

В результате будет получено упорядоченное множество
{9, 8, 7, 6, 5, 3}.

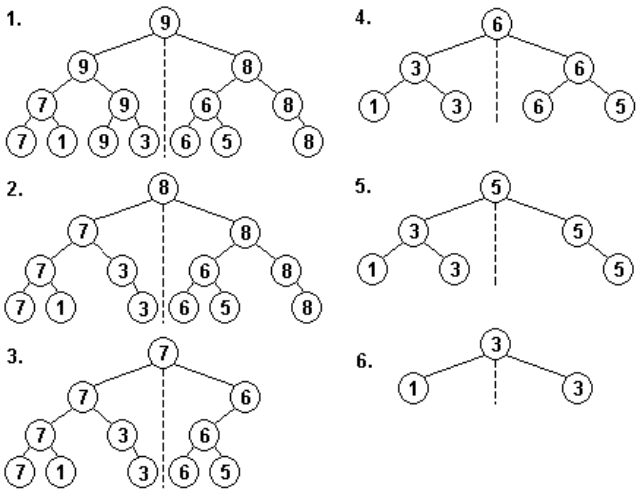


Рис.11. Турнирная сортировка

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 большим весом.

2. Листья бинарного дерева находятся либо в одном уровне, либо в двух соседних.
3. Листья нижнего уровня располагаются левее листьев более высокого уровня, т.е. заполнение каждого уровня осуществляется слева направо.

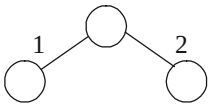


Рис.12. Сравнение в триаде: 1 - первое сравнение; 2 - второе сравнение

В ходе преобразования элементы триад сравниваются дважды, при этом элемент с большим весом перемещается вверх, а с меньшим - вниз (рис.12).

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.13), 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}.

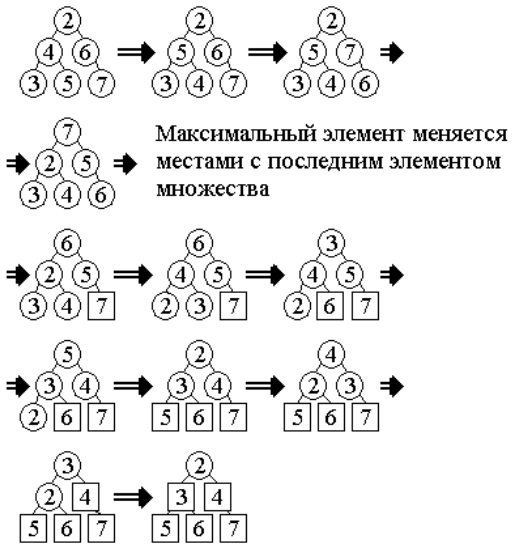


Рис.13. Пирамидальная сортировка

Функция сложности алгоритма

Для оценки эффективности алгоритмов используется функция сложности алгоритма, которая обозначается заглавной буквой "O", в круглых скобках записывается аргумент. Например, функция сложности $O(n^2)$ читается как функция сложности порядка n^2. Функция сложности алгоритма - это функция, которая определяет количество сравнений, перестановок, а также временные и ресурсные затраты на реализацию алгоритма.

Функция сложности принимает ряд значений, показанный на рис.14.

Чем правее на оси расположена функция сложности, тем менее эффективен алгоритм.

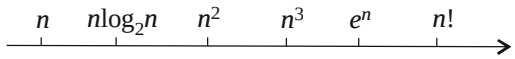


Рис.14. Ряд значений функции сложности

Лабораторное задание

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

Проведение лабораторной работы осуществляется по следующему алгоритму.

1. Вызвать систему **Sort_new**, 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:

K - конец работы;

? - выдача краткого сообщения о командах;

U - задание условий для генерации неупорядоченного массива (задаются число элементов и границы элементов неупорядоченного массива). Заданные условия сохраняются до следующего вызова функции "U";

G - генерации неупорядоченного массива (необходима перед каждой сортировкой);

P - вывод массива (неупорядоченного до сортировки или упорядоченного после нее);

W - вывод времени сортировки и количества элементов массива;

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 указанных в лабораторной работе методов сортировки провести следующие исследования: выполнить сортировку массивов из *N* чисел (варианты заданий даны в Приложении; номер варианта соответствует номеру компьютера). Результаты занести в форму табл.1.

Форма таблицы 1

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

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

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

Результаты занести в форму табл.1.

3. Провести исследование методов сортировки с использованием программ **WinSort** и **Sort**.

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.

Требования к отчету

Отчет должен содержать:

1) конспект лабораторной работы;
2) примеры сортировки;
3) результаты выполнения работы (таблицы и графики);
4) выводы по работе;

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?

Приложение

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

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

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

Произвести расчет функции сложности разработанного алгоритма по полученным временным затратам.

Пояснения к выполнению работы

Определить время выполнения какого-либо фрагмента программы можно с помощью библиотечной функции gettime(), прототип которой находится в заголовочном файле dos.h. Эта функция должна получать аргумент - адрес структуры типа time, в поля которой и заносится информация о текущем времени. В обобщенном виде программа, в которой выполняется контроль времени, может выглядеть следующим образом:

```
#include <dos.h>
void main()
```

```
{
    time t1,t2;
    gettime(&t1);
// Фрагмент, время работы которого измеряется.
    gettime(&t2);
// ...
}
```

Для получения интервала времени следует вычесть поля структуры t1 из полей структуры t2. Представить этот интервал в секундах можно следующим образом:

```
double delta_t=(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.;
```

Современные процессоры выполняют сортировку небольших массивов за время, значительно меньшее, чем тысячная доля секунды (t1.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 занимает определенное время, которое необходимо учесть. После завершения всего процесса следует проверить правильность сортировки,
выведя содержимое второго массива на экран.

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;

    gettime(&t1);
// Копирование из массива arr1 в массив arr.
    gettime(&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.;

    gettime(&t1);
// Цикл сортировки из большого числа проходов.
    gettime(&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.
}
```

Лабораторная работа № 2. Методы поиска

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

Продолжительность работы - 2 ч.

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 $\{k_1, k_2, k_3, ... k_n\}$. Необходимо отыскать во множестве ключ k_i. Поиск может быть завершен в двух случаях:

1) ключ во множестве отсутствует;

2) ключ найден во множестве.

Последовательный поиск

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

Бинарный поиск

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

$\{k_1 \le k_2 \le k_3 \le k_4 ... \ k_{n-1} \le k_n\}$.

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

$N_{\text{элемента}} = [n/2] + 1$,

где квадратные скобки обозначают, что от деления берется только целая часть (всегда округляется в меньшую сторону). В методе бинарного поиска анализируются только центральные элементы.

Пример. Дано множество
{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.

$N_{элемента} = [n/2] + 1 = [22/2] + 1 = 12$.

$K \sim k_{12}$.

Символ "~" обозначает сравнение элементов (чисел, значений).

61 > 60. Дальнейший поиск в правом подмножестве
{61,69,75,79,80,81,95,101,123,198}.

Шаг 2.

$N_{элемента} = [n/2] + 1 = [12/2] + 1 = 7$.

$K \sim k_{19}$.

61 < 95. Дальнейший поиск в левом подмножестве
{61,69,75,79,80,81}
(относительно предыдущего подмножества).

Шаг 3.

$N_{элемента} = [n/2] + 1 = [6/2] + 1 = 4$.

$K \sim k_{16}$.

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

Шаг 4.

$N_{элемента} = [n/2] + 1 = [4/2] + 1 = 3$.

$K \sim k_{15}$.

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

Шаг 5.

$N_{элемента} = [n/2] + 1 = [2/2] + 1 = 2$.

$K \sim k_{14}$.

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,…}.

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,…}.

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

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

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

Шаг 4. $K \sim k_5$. 42 > 11 => сравнение продолжается.
Шаг 5. $K \sim k_8$. 42 > 19 => сравнение продолжается.
Шаг 6. $K \sim k_{13}$. 42 > 35 => сравнение продолжается.
Шаг 7. $K \sim k_{18}$. 42 < 52 => найден интервал, в котором находится отыскиваемый ключ: от 13 до 18 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 упорядочено по возрастанию весов. Первоначальное сравнение осуществляется на расстоянии шага d, который определяется по формуле

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

где i - номер первого рассматриваемого элемента; j - номер последнего рассматриваемого элемента; K - отыскиваемый ключ; K_i, K_j - значения ключей в i-й и j-й позициях; [ ] - целая часть от числа.

Идея метода заключается в следующем: шаг d меняется после каждого этапа по формуле, приведенной выше.

Алгоритм заканчивает работу при $d = 0$, при этом анализируются соседние элементы, после чего делается окончательно решение (рис.2).

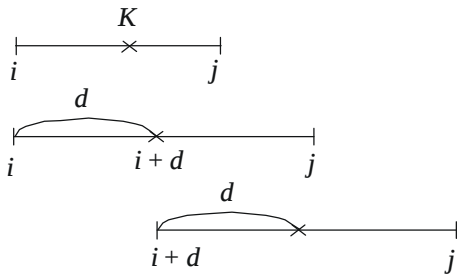


Рис.2. Интерполяционный поиск

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

Пример. Дано множество ключей:
{2, 9, 10, 12, 20, 24, 28, 30, 37, 40, 45, 50, 51, 60, 65, 70, 74, 76}.

Пусть искомый ключ равен 70 (K = 70).

Шаг 1. Определяется шаг d для исходного множества ключей:

$$\boldsymbol{d = [(18 - 1)(70 - 2) / (76 - 2)] = 15.}$$

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

$k_{16} \sim K$, 70 = 70. Ключ найден.

Поиск по бинарному дереву

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

Пусть дано бинарное дерево (рис.3).

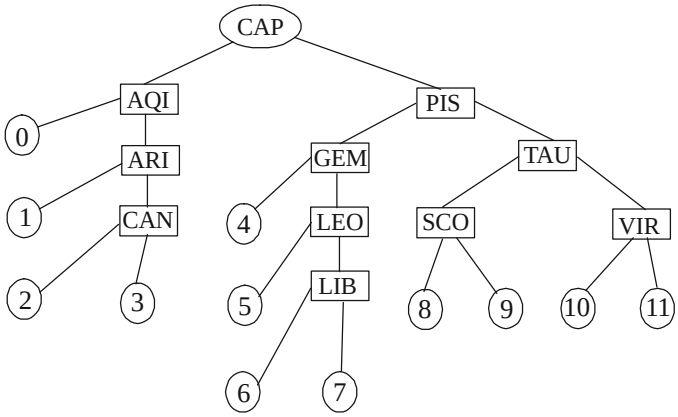


Рис.3. Бинарное дерево

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.

При этом узлы дерева имеют структуру, показанную на рис.4.

Ключ	Информационная часть	Указатель на левое поддерево	Указатель на правое поддерево
KEY		LLINK	RLINK

Рис.4. Структура узлов бинарного дерева

Бинарное дерево является сбалансированным, если высота левого поддерева каждого узла отличается от высоты правого не более чем на ±1 (рис.5).

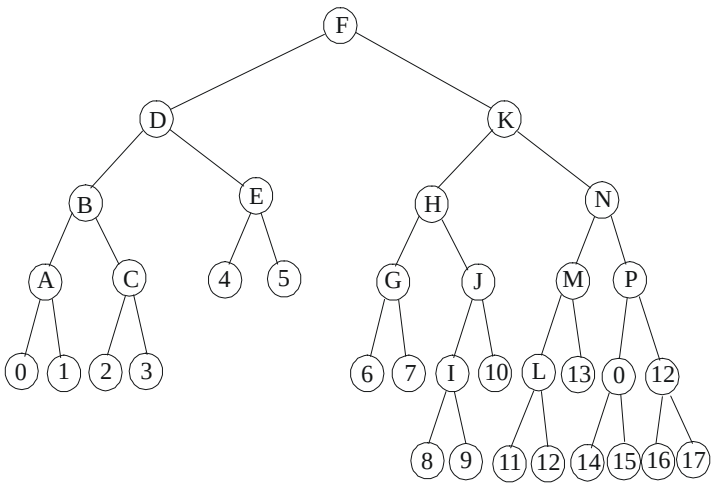


Рис.5. Сбалансированное бинарное дерево

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, показанную на рис.6, где B - показатель сбалансированности узла, т.е. разность высот правого и левого поддерева (B = +1; 0; –1).

Ключ	Указатель на левое поддерево	Указатель на правое поддерево	Показатель сбалансированности узла
KEY	LLINK	RLINK	B

Рис.6. Структура узлов сбалансированного дерева

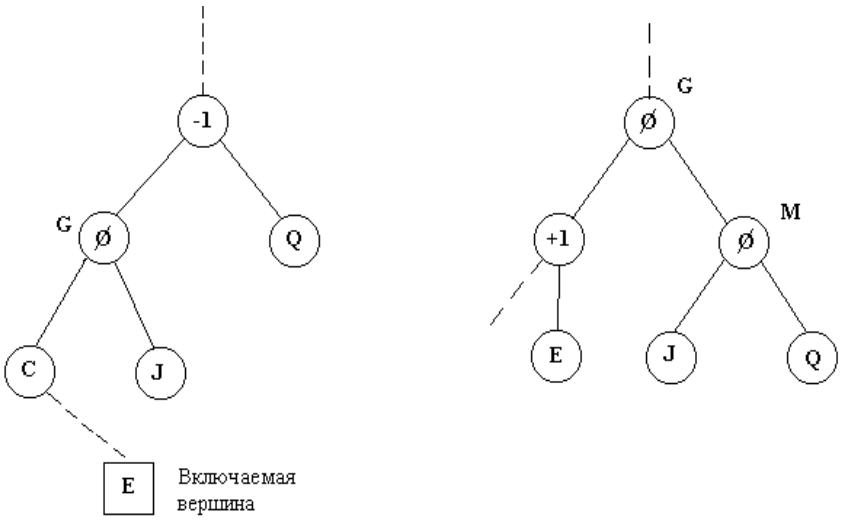


Рис.7. Учет показателей сбалансированности

При восстановлении баланса дерева по высоте учитывается показатель B (рис.7).

Символы +1, ∅, –1 указывают, что левое поддерево выше правого, поддеревья равны по высоте, правое поддерево выше левого.

Поиск по бору

Особую группу методов поиска образует представление ключей в виде последовательности цифр и букв. Рассмотрим, например, имеющиеся во многих словарях буквенные высечки. 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:

Форма таблицы 1

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 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 побуквенно не разбивается.

Таблица 2

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

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 оказалась, например, буква B, то из табл.2 видно, что ей соответствует слово BOR. Если же первая буква A, то первый узел передает управление к узлу 2, где аналогичным образом отыскивается вторая буква. Узел 2 указывает, что вторыми буквами будут _, A, B и т.д.

Поиск хешированием

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

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

где k - ключ; m - целое число; mod - целочисленный остаток от деления.

Пример. Дано 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 $m = [10/2] = 5$,

$h(k) = \{4, 1, 4, 0, 3, 0\}$.

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).

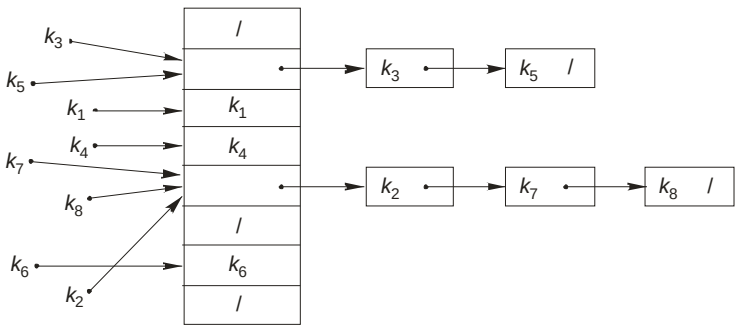


Рис.8. Разрешение коллизий

Пример. Дано множество
{7, 13, 6, 3, 9, 4, 8, 5}.

Найти ключ $K = 27$, используя поиск хешированием.

Хеш-функция равна $h(k) = K \bmod (m)$;

$m = [13/2] = 6$ (так как 13 - максимальный ключ),

$h(k) = \{1, 1, 0, 3, 3, 4, 2, 5\}$.

Для устранения коллизий построим табл.3.

Таблица 3

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

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

Например, если отыскивается ключ $K = 27$, тогда $h(k) = 27 \bmod 6 = 3$.

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
$X = x[1]x[2] \ldots x[n]$
и просматривает его слева направо буква за буквой, заполняя при этом массив натуральных чисел $l[1] \ldots l[n]$, где $l[i]$ = длина слова $l(x[1]\ldots x[i])$.

Таким образом, $l[i]$ есть длина наибольшего начала слова $x[1]\ldots x[i]$, 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 # - специальная буква, не встречающаяся ни в A, ни в 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 $l[1]\ldots l[i]$ уже найдены. Читается очередная буква слова (т.е. $x[i+1]$) и вычисляется $l[i+1]$.

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

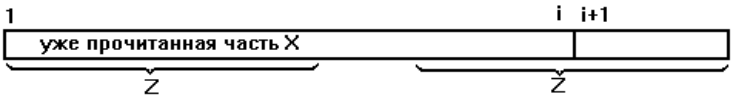


Рис.9. Анализ слова

Получаем следующий способ отыскания слова Z. Рассмотрим все начала слова $x[1]\ldots x[i]$, являющиеся одновременно его концами. Из них выберем 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 пора воспользоваться сделанными приготовлениями и вспомнить, что все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции l.

Получим следующий фрагмент программы.

```
i:=1; 1[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 $X = x[1]...x[n]$ подсловом слова $Y = y[1]...y[m]$.

Решение. Вычисляем таблицу $l[1]...l[n]$, как раньше.

```
        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 отсутствии}
                        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 делает то, что на первый взгляд кажется невозможным: в типичной ситуации он читает лишь небольшую часть всех букв слова, в котором ищется заданный об-

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

Приведем самый простой вариант этого алгоритма, который не гарантирует быстрой работы во всех случаях. Пусть $x[1]...x[n]$ - образец, который надо искать. 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 будет удобно предположить 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 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 сдвигаем на всю длину образца}
    end else begin
        если нынешнее положение подходит, т.е. если
        x[i]..x[n]=y[last-n+1]..y[last],
        то сообщить о совпадении;
        last:=last+1;
    end;
end;
```

Алгоритм Рабина

Этот алгоритм основан на простой идее. Представим себе, что в слове длиной m ищется образец длиной n. Вырежем окошко размером n и будем двигать его по входному слову. При этом проверяем, 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 окошке 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 окошке, нужно прочесть все буквы этого слова. Так уж лучше их сразу сравнить с образцом. Тем не менее выигрыш возможен, так как при сдвиге окошка 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 (заменив буквы кодами). Эти числа будем рассматривать как коэффициенты многочлена степени $n - 1$ и вычислим значение этого многочлена по модулю p в точке x. Это и будет одна из функций семейства (для каждой пары p и x получается, таким образом, своя

функция). Сдвиг окошка на 1 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. Тогда им соответствуют различные многочлены (предполагаем, что коды всех букв различны - это возможно, если p больше числа букв алфавита). Совпадение значений функции означает, что в точке x эти два различных многочлена совпадают, т.е. их разность обращается в 0. 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 провести анализ временных затрат для списков различной размерности.

Алгоритм проведения лабораторной работы следующий.

1. Выполнить тестовый пример поиска с использованием программы **LAB_FIND**.

Путь к файлу: D:\ИПОВС\АиСД\POISK\ Lab_Find.exe:

а) выполнить поиск в массиве из N чисел (варианты заданий даны в Приложении; номер варианта соответствует номеру компьютера). Результаты занести в форму табл.4;

Форма таблицы 4

Метод поиска			
Количество элементов массива N			
Время поиска t, с			

б) оценить сложность рассмотренных методов поиска;

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

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

2. Исследовать методы поиска с использованием программ **PoiskNew** и **Poisk.**

Путь: D:\ИПОВС\АиСД\POISK\ PoiskNew (Poisk).

3. Провести исследование метода хеширования.

Путь: D:\ИПОВС\АиСД\POISK\ Хеширование.

Требования к отчету

Отчет должен содержать:

1) конспект лабораторной работы;
2) примеры методов поиска;
3) результаты выполнения работы;
4) выводы по работе.

Контрольные вопросы

1. Что понимается под поиском?
2. Каковы особенности поиска: последовательного, бинарного, интерполяционного, фибоначчиевого, по бинарному дереву, по бору, хешированием?
3. В чем состоит методика анализа сложности алгоритмов поиска?

Приложение

Варианты заданий

Провести анализ методов поиска для списков различной размерности и построить зависимости времени от числа элементов исходного множества (табл.П1).

Таблица П1

Вариант	n_1	n_2	n_3
1; 15	1000	4000	6000
2; 16	800	3000	7000
3; 17	2000	5000	6800
4; 18	1500	3500	6000
5; 19	1000	2800	8500
6; 20	500	2500	6500
7; 21	900	3000	8500
8; 22	1200	2000	7500
9; 23	2500	3500	6500
10; 24	1600	2800	8800
11; 25	700	3400	7200
12; 26	1900	2700	8000
13; 27	1300	3500	6000
14; 28	1700	2800	7500

Составить программу для реализации метода поиска (табл.П2). (Использовать алгоритмы Кнута - Морриса - Пратта, Бойера - Мура, Рабина для поиска текстовой информации.)

Таблица П2

Вариант	Составить программу
1; 15	Бинарный поиск
2; 16	Интерполяционный поиск
3; 17	Поиск хешированием
4; 18	Фибоначчиев поиск
5; 19	Поиск по бинарному дереву
6; 20	Поиск по бору
7; 21	Фибоначчиев поиск
8; 22	Поиск по бору
9; 23	Интерполяционный поиск
10; 24	Поиск по бору
11; 25	Поиск по бинарному дереву
12; 26	Поиск хешированием
13; 27	Бинарный поиск
14; 28	Фибоначчиев поиск

Лабораторная работа № 3. Итеративные и рекурсивные алгоритмы

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

Продолжительность работы - 2 ч.

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

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

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 0 или 1;
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 ПЭВМ является описание его с использованием цикла. Программа итеративного алгоритма вычисления факториала представлена ниже. Программа состоит из процедуры-функции FACTORIAL и основной программы. В основной программе происходит ввод значения *N*, вызов процедуры-функции и печать результата.

```
{ИТЕРАТИВНОЕ  ВЫЧИСЛЕНИЕ  ФАКТОРИАЛА }
 PROGRAM FI;
      VAR
      FAC:  LONGINT;
      N:  INTEGER;
{  ФУНКЦИЯ  ВЫЧИСЛЕНИЯ  ФАКТОРИАЛА }
FUNCTION   FACTORIAL (N: INTEGER): LONGINT;
         VAR    F:  LONGINT;
                I:  INTEGER;
      BEGIN
```

```
        IF   (N=O) OR  (N=1) THEN   FACTORIAL:=1 ELSE
        BEGIN
        F:= 1;
        FOR   I: = 2 TO N DO
            F: = F * I;
            FACTORIAL:=F;
        END;
     END;
{   ОСНОВНАЯ   ПРОГРАММА }
BEGIN
   WRITELN ('ВВЕДИТЕ   ЗНАЧЕНИЕ N')
   READLN (N);
   FAC:= FACTORIAL(N);   {  ВЫЗОВ ФУНКЦИИ   FACTORIAL   }
   WRITELN('  ФАКТОРИАЛ =' , FAC) ;
   READLN;
END.
```

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. $n!$ вычисляется через меньшую размерность этой же задачи $(n - 1)!$

Программа рекурсивного алгоритма вычисления факториала представлена ниже.

```
{РЕКУРСИВНОЕ ВЫЧИСЛЕНИЕ ФАКТОРИАЛА}
PROGRAM FR ;
VAR
     fac: longint;
     n: integer;
{Функция вычисления факториала}
   FUNCTION factorial (n: integer): longint ;
   Begin
   If (n=0) or (n=1) then factorial: =1
                        Else factorial: = factorial (n–1)*n;
   End;
{   ОСНОВНАЯ   ПРОГРАММА }
BEGIN
  Writeln (' введите значение n');
  Readln (n);
   fac: = factorial (n); { ВЫЗОВ ФУНКЦИИ   FACTORIAL   }
  Writeln ('факториал = ', fac);
Readln;
END.
```

Процедура-функция имеет следующие особенности:

- при вычислении факториала происходит обращение функции к самой себе (подчеркнуто в выражении), но с меньшим значением аргумента $n - 1$ по сравнению с первым вызовом n:

factorial : = factorial (n–1)*n;

- при вычислении факториала не используется цикл, что является существенной особенностью рекурсивного алгоритма.

Рассмотрим последовательность действий при рекурсивном вычислении факториала для $n = 3$:

1) внешний вызов из основной программы factorial (3);

2) первый рекурсивный вызов factorial (2); в операторе
factorial: = factorial (n–1) *n,
где не происходят никакие вычисления - только вызов (подчеркнуто);

3) второй рекурсивный вызов factorial (1);

4) получение значения factorial (1): = 1;

5) возврат из второго рекурсивного вызова и вычисление факториала factorial (2): = 1*2 = 2;

6) возврат из первого рекурсивного вызова и вычисление факториала factorial (3): = 2*3 = 6;

7) возврат в основную программу fac: = 6.

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 список - это линейный список, в котором все исключения и добавления происходят в любом месте списка.

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

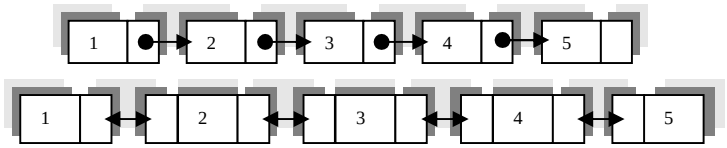


Рис.1. Однонаправленный и двунаправленный списки

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

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

Очередь называют циклической памятью или списком 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. Очередь по сути - однонаправленный список, только добавление и исключение элементов происходит на концах списка.

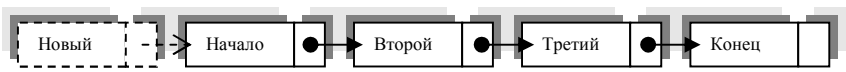


Рис.2. Очередь

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").

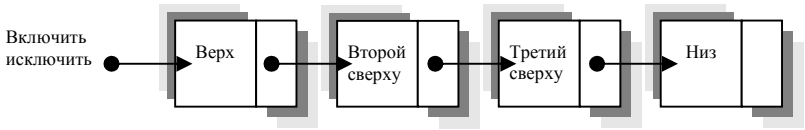


Рис.3. Стек

Стек - часть памяти ОЗУ компьютера, которая предназначается для временного хранения байтов, используемых микропроцессором. Действия со стеком производятся при помощи регистра указателя стека. Любое повреждение этой части памяти приводит к фатальному 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).

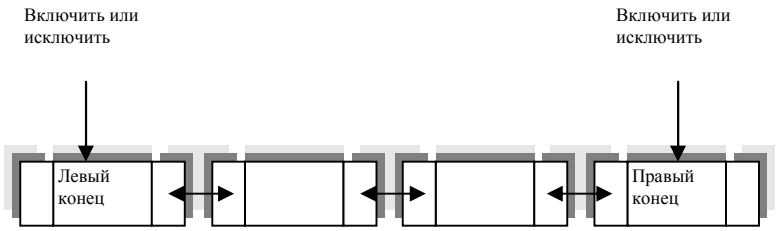


Рис.4. Дек

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

Предварительно обратимся к понятиям 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 ^. Структуру данных, рассмотренную в виде списка, можно представить на языке Турбо Паскаль следующим образом:

```
TYPE LINK = ^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;
```

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;
```

Следует обратить внимание на важный факт: 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 существует 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).

Задача. Составить программу на языке Турбо Паскаль для формирования обхода узлов бинарного дерева. Набор способа обхода дерева позволяет ввести отношение порядка для узлов дерева.

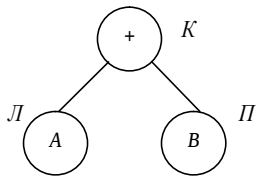
Решение. Для размещения узлов дерева в памяти ПЭВМ воспользуемся списковой структурой данных. Каждый элемент этой структуры соответствует некоторому узлу дерева и описывается записью следующего вида:

```
TREE=RECORD
       LEFT: ^TREE;
       RIGHT: ^ TREE;
       IDENT: CHAR;
      END;
```

где LEFT и RIGHT - указатели (адреса) элементов структуры данных, представляющие узлы, которые являются соответственно корнями левого и правого поддерева данного узла; в поле IDENT находится односимвольный идентификатор узла.

Наиболее распространены три способа обхода узлов дерева, которые 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).



КЛП (Корень – Левая – Правая)
Прямой + *AB* (префиксная)
ЛКП Обратный *A* + *B* (инфиксная)
ЛПК Концевой *AB* + (постфиксная)

Рис.5. Три различных обхода по бинарному дереву и соответствующие формы записи

В результате обхода дерева, приведенного на 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).

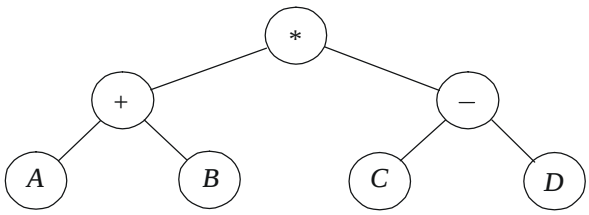


Рис.6. Бинарное дерево

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, а 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.

Ниже приведены программы для создания в памяти ПЭВМ списковой структуры бинарного дерева (процедура CREATE) и обхода его узлов в порядке сверху вниз (процедура PREORDER).

Листинг 1.

{ ФОРМИРОВАНИЕ БИНАРНОГО ДЕРЕВА }

```
TYPE                    { ОПИСАНИЕ ДЕРЕВА }
NODE  =  ^TREE;         { ТИП УКАЗАТЕЛЯ УЗЛА ДЕРЕВА }
TREE  =  RECORD         { СТРУКТУРА УЗЛА БИНАРНОГО ДЕРЕВА }
    LEFT  :^TREE;     { УКАЗАТЕЛЬ ЛЕВОГО ПОДДЕРЕВА  }
    RIGHT :^TREE;     { УКАЗАТЕЛЬ ПРАВОГО ПОДДЕРЕВА }
    IDENT : CHAR;     { ИДЕНТИФИКАТОР УЗЛА ДЕРЕВА }
    END;
PROCEDURE CREATE (VAR  A:NODE);
```

```
{ РЕКУРСИВНАЯ ПРОЦЕДУРА СОЗДАНИЯ В ОПЕРАТИВНОЙ ПАМЯТИ  СТРУКТУРЫ БИНАРНОГО
ДЕРЕВА В ПЕРЕМЕННОЙ "A" ФОРМИРУЕТСЯ АДРЕС КОРНЯ ПОЛУЧЕННОГО ДЕРЕВА ИЛИ
ПОДДЕРЕВА }
VAR
      B : NODE;      { АДРЕС ПОДДЕРЕВА ДАННОГО УЗЛА }
      R : CHAR;                { РАБОЧАЯ ПЕРЕМЕННАЯ }
 BEGIN
          NEW(A);  { РЕЗЕРВИРОВАНИЕ ПАМЯТИ ДЛЯ
                              НОВОГО УЗЛА ДЕРЕВА }
          WRITE(‘ ВВЕДИТЕ ИМЯ УЗЛА> ’);
          READLN(A^.IDENT);
          WRITE(‘ ЕСТЬ ЛЕВОЕ ПОДДЕРЕВО У УЗЛА ‘, A^.IDENT,
                   ’  ? (Y/N) ‘);
          READLN(R);
          IF (R='Y')  THEN
          BEGIN
          CREATE(B); { ФОРМИРОВАНИЕ ЛЕВОГО ПОДДЕРЕВА
                              УЗЛА }
          A^.LEFT:=B;
          END
          ELSE
          A^.LEFT:=NIL;  { У ДАННОГО УЗЛА НЕТ ЛЕВОГО
                               ПОДДЕРЕВА  }
                WRITE(‘ ЕСТЬ ПРАВОЕ ПОДДЕРЕВО У УЗЛА  ’,A^.IDENT, ’ ? (Y/N)  ‘);
          READLN(R);
          IF (R='Y')  THEN
          BEGIN
          CREATE(B);           { ФОРМИРОВАНИЕ ПРАВОГО
                                    ПОДДЕРЕВА УЗЛА }
          A^.RIGHT:=B;
          END
          ELSE
        A^.RIGHT:=NIL;  { У ДАННОГО УЗЛА НЕТ ПРАВОГО
                               ПОДДЕРЕВА }
   END;
```

Листинг 2.

```
{ РЕКУРСИВНАЯ ПРОЦЕДУРА ОБХОДА УЗЛОВ ДЕРЕВА
     СВЕРХУ   ВНИЗ  }

PROCEDURE   PREORDER (A: NODE);
BEGIN
          IF A<>NIL    THEN
          BEGIN
          WRITE(A^.IDENT:2);  { ПЕЧАТЬ ИДЕНТИФИКАТОРА УЗЛА }
     PREORDER(A^.LEFT);     { ОБХОД ЛЕВОГО ПОДДЕРЕВА }
     PREORDER(A^.RIGHT);   { ОБХОД ПРАВОГО ПОДДЕРЕВА }
          END;
END;
```

В процедуре CREATE функция NEW(A) резервирует в памяти ПЭВМ 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 обработ-

ке 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.

Лабораторное задание

Алгоритм проведения лабораторной работы следующий.

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

Используя программу **DataStruct**, задать исходное множество элементов, содержащее 12 - 15 элементов. Путь к программе:

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 **Obhod1** либо **Obhod2**, сформировать бинарные деревья, содержащие 8, 10, 12 элементов и выполнить для каждого из них три вида обхода дерева.

Путь к программе: D:\ИПОВС\АиСД\ Obhod1 (Obhod2).

3. Разработать и отладить программу анализа арифметического выражения (рис.7).

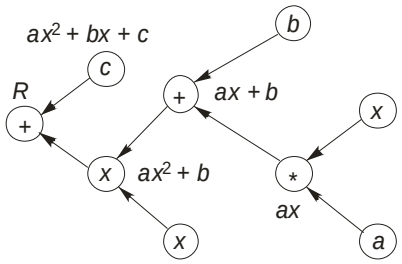


Рис.7. Запись арифметических выражений

Программа должна:

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

б) при помощи стека проанализировать правильность расстановки круглых скобок: если в строке встретилась "(", то записать ее в стек; если встретилась ")", то извлечь один символ из стека; операции записи и извлечения символа из стека реализовать 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 а) - в) до тех, пока пользователь не введет пробел.

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

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

Требования к отчету

Отчет должен содержать:

1) конспект лабораторной работы;
2) три вида обхода дерева для своего варианта;
3) программу итеративного рекурсивного вычисления суммы;
4) результаты выполнения работы;
5) вывод по работе.

Контрольные вопросы

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

Приложение

Варианты заданий

1; 13	$a_n = \frac{n!}{2^n}$
2; 14	$a_n = \frac{2^n}{n!}$
3; 15	$a_n = \frac{3^{n+1}}{n!}$
4; 16	$a_n = \frac{n!}{2^n 3^{n+1}}$
5; 17	$a_n = \left(\frac{2}{3}\right)^n n!$
6; 18	$a_n = \frac{3^n}{2n!}$
7; 19	$a_n = \left(\frac{2}{3}\right)^n$
8; 20	$a_n = \frac{3^n}{n!}$
9; 22	$a_n = \frac{n!}{2^{n-1}}$
10; 23	$a_n = \frac{2^{n+1}}{3^{n-1}}$
11; 24	$a_n = 2^n (n-1)!$
12; 25	$a_n = \frac{n!}{4^n}$

Задание выполняется при домашней подготовке. Для определения времени выполнения программы необходимо воспользоваться стандартной встроенной функцией GETTIME.

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 ч.

Теоретические сведения

Предположим, что необходимо 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 входят приобретение оборудования, прокладка линий связи, обслуживание системы и т.д. Необходимо определить стоимость такой сети.

Нетрудно видеть, что сформулированная здесь задача имеет много других аналогов. Например, требуется соединить несколько населенных пунктов линиями телефонной связи таким образом, чтобы все эти пункты были связаны в сеть и чтобы 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 γ 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 $\gamma = n - m + 1 = 7 - 5 + 1 = 3$.

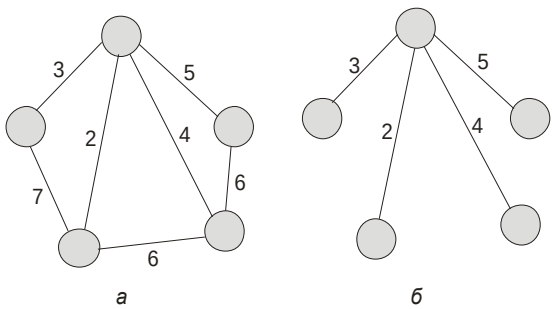


Рис.1. Исходный граф (*а*) и его остовное дерево (*б*)

Это значит, что если на графе (рис.1,*а*) убрать три ребра, то в нем не останется ни одного цикла, а суммарный вес ребер будет равен 14. Для построения остовного дерева графа используются алгоритмы Крускала (рис.2) и Прима.

Блок *Sort* e_i предполагает предварительную сортировку 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 $v_1, v_2, \ldots v_m$, где m -

количество вершин графа). Поэтому считается, что множество W_s имеет вид: $W_s = \{\{v_1\},\{v_2\},\ldots\{v_n\}\}$, 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: $W_k = \{\ldots, v_k, \ldots\}$ и $W_l = \{\ldots, v_l, \ldots\}$.

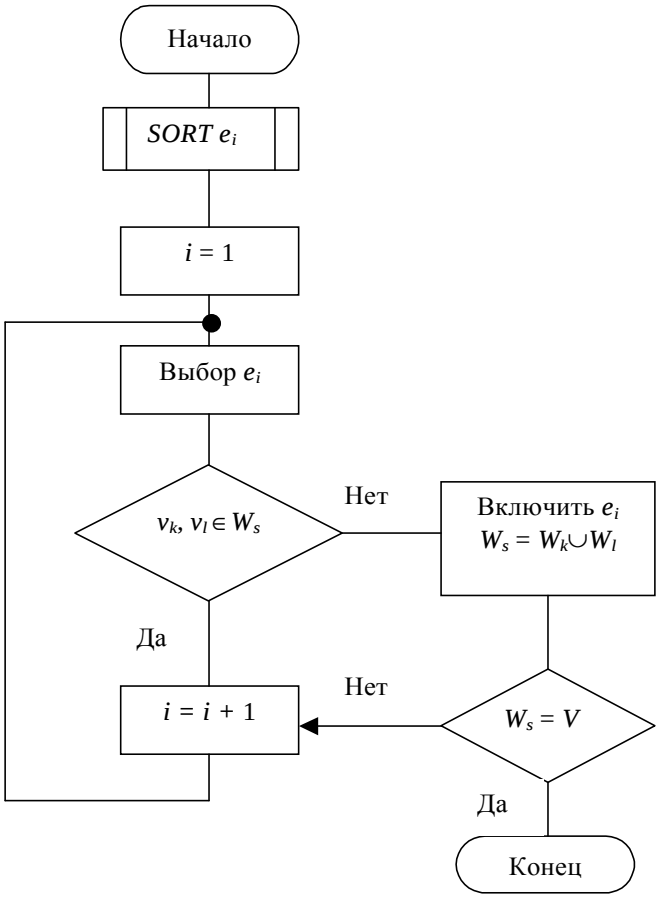


Рис.2. Схема алгоритма Крускала

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 алгоритма 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 1. Дана схема микрорайона. Необходимо соединить дома телефонным кабелем таким образом, чтобы его длина была минимальной.

Схему микрорайона представим взвешенным графом (рис.3).

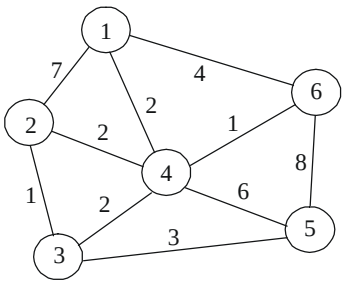


Рис.3. Схема микрорайона

Определим цикломатическое число графа:

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

т.е. на графе необходимо удалить 5 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.

Таблица 1

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

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}. 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 остовное дерево.

Последовательно просматривая табл.1, получим схему соединения телефонным кабелем домов в микрорайоне (рис.4).

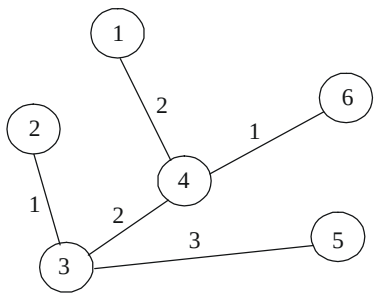


Рис.4. Остовное дерево графа

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 остовного 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 остовное дерево не будут включены все вершины исходного графа.

Пример 2. Дана схема компьютерной сети (рис.5) Необходимо соединить компьютеры таким образом, чтобы длина проводки была минимальной.

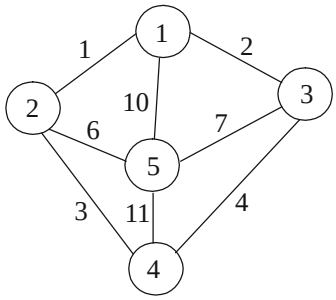


Рис.5. Схема компьютерной сети

Определим цикломатическое число графа: $\gamma = n - m + 1 = 8 - 5 + 1 = 4$. При просмотре 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

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**

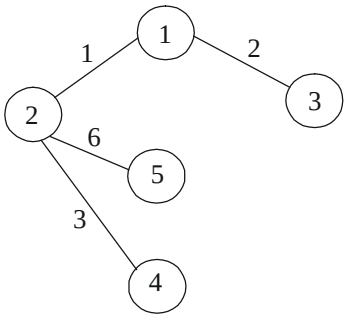


Рис.6. Остовное дерево графа

Для построения остовного 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 остовное дерево минимального веса (рис.6).

Лабораторное задание

Алгоритм проведения лабораторной работы следующий:

1) изучить основные понятия теории графов.

Путь к файлу: D:\ИПОВС\АиСД\OSTOV**Теория графов**;

2) вызвать систему **Ostov1**, включающую в себя изучение методов Крускала и Прима.

Путь к файлу: D:\ИПОВС\АиСД\OSTOV\Ostov1.

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 двумя методами. По окончании работы выставляется оценка;

3) вызвать системы **Ostov2 и Ostov3** для исследования методов Крускала и Прима (см. Приложение 1).

Путь к файлу: D:\ИПОВС\АиСД\OSTOV\Ostov2 (Ostov3).

Результаты записать в тетрадь.

4) решить задачи из Приложения 2.

Требования к отчету

Отчет должен содержать:

1) конспект лабораторной работы;
2) примеры работы методов Крускала и Прима;
3) результаты выполнения работы;
4) выводы по работе.

Контрольные вопросы

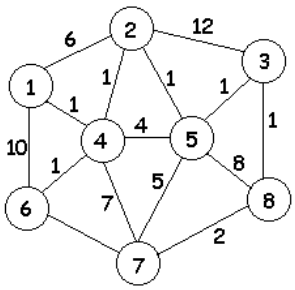
1. Что понимается под остовным деревом?
2. Каковы особенности методов Крускала и Прима?
3. В чем состоит методика анализа сложности алгоритмов построения остовного дерева графа?

Приложение 1

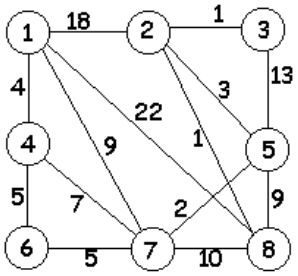
Варианты заданий

Построить остовное дерево графа методами Крускала и Прима.

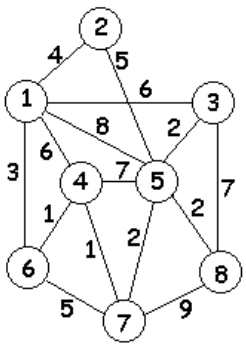
1; 11



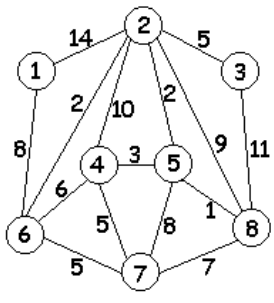
2; 12



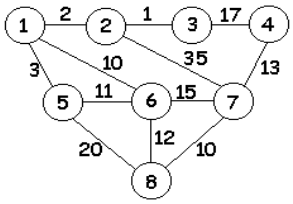
3; 13



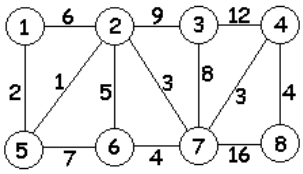
4; 14



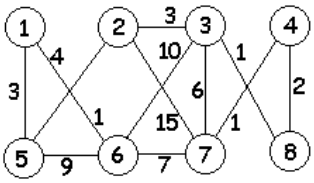
5; 15



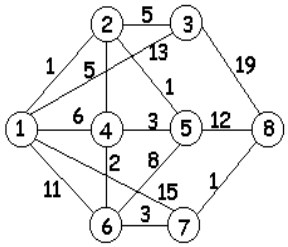
6; 16



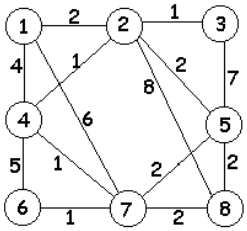
7; 17



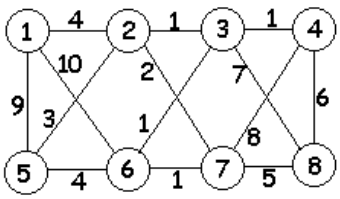
8; 18



9; 19



10; 20



Приложение 2

Решить задачи

Задача 1. Дана карта дорог между городами (рис.П2.1), где указана условная длина каждой дороги (данные не совпадают с настоящими). Соединить все города автострадой минимальной длины, т.е. построить остовное дерево методами Прима и Крускала.

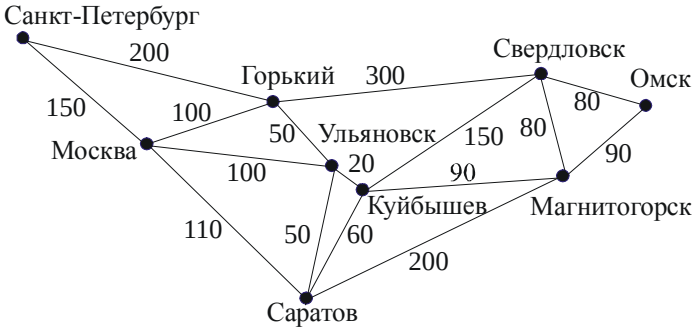


Рис. П2.1. Карта автомобильных дорог

Задача 2. Задать веса ребрам и построить остовное дерево графа (рис.П2.2).

Задача 3. Между 9 планетами Солнечной системы введено космическое сообщение. Ракеты летают по следующим маршрутам: Земля - Меркурий, Плутон - Венера, Земля - Плутон, Плутон - Меркурий, Меркурий - Венера, Уран - Нептун, Нептун - Сатурн, Сатурн - Юпитер, Юпитер - Марс и Марс - Уран. Можно ли добраться с Земли до Марса?

Задача 4. Может ли в государстве, в котором из каждого города выходит 3 дороги, быть ровно 100 дорог?

Задача 5. Какое максимальное число висячих вершин может иметь дерево, обладающее 9 вершинами? Какое минимальное число висячих вершин оно может иметь? Сделайте рисунки таких деревьев.

Задача 6. В доме отдыха 57 корпусов. Электрик решил соединить телефонными проводами каждый корпус с пятью другими. Сможет ли он это сделать?

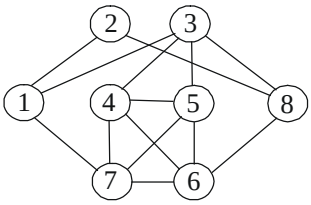


Рис.П2.2. Исходный граф

Задача 7. У предприятия имеется 4 завода и 9 торговых точек. Каждый завод должен поставлять свою продукцию во все торговые точки. Сколько всего дорог необходимо проложить, чтобы осуществить задуманное? Определите вид полученного графа.

Задача 8. На участке 3 дома и 3 колодца. От каждого дома к каждому колодцу ведет тропинка. Когда владельцы домов поссорились, они задумали проложить дорогу от каждого дома к каждому колодцу так, чтобы не встречаться на пути к колодцам. Может ли осуществиться их намерение?

Задача 9. Муравей забрался в банку из-под сахара, имеющую форму куба. Сможет ли он последовательно обойти все ребра куба, не проходя дважды по одному ребру?

Задача 10. На занятии 20 школьников решили каждый по 6 задач, причем каждая задача была решена ровно двумя школьниками. Можно ли организовать разбор всех задач таким образом, чтобы каждый школьник рассказал ровно по 3 задачи.

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 ч.

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 данной 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 поставленной цели. Таким образом, метод динамического программирования - метод пошаговой оптимизации.

Введем функцию f_i, определяющую минимальную длину пути из начальной 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:

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

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 1. Определить кратчайший путь из вершины 1 в вершину 10 для взвешенного ориентированного графа, представленного на рис.1.

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

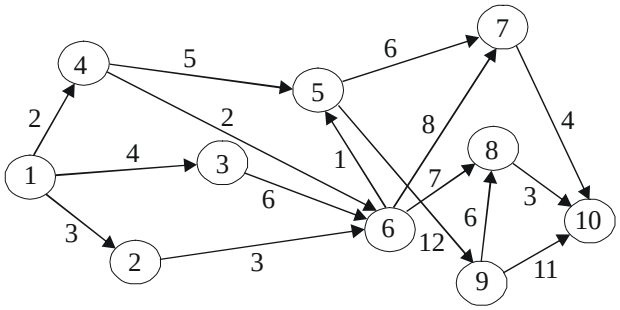


Рис.1. Взвешенный ориентированный граф

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

$$f_2 = \min\{S_{21} + f_1\} = \{3 + f_1\} = \{3 + 0\} = 3;$$

$$f_3 = \min\{S_{31} + f_1\} = \{4 + f_1\} = \{4 + 0\} = 4;$$

$$f_4 = \min\{S_{41} + f_1\} = \{2 + f_1\} = \{2 + 0\} = 2;$$

$$f_6 = \min\begin{Bmatrix} S_{64} + f_4 \\ S_{63} + f_3 \\ S_{62} + f_2 \end{Bmatrix} = \min\begin{Bmatrix} 2 + 2 \\ 6 + 4 \\ 3 + 3 \end{Bmatrix} = 4;$$

$$f_5 = \min\begin{Bmatrix} S_{54} + f_4 \\ S_{56} + f_6 \end{Bmatrix} = \min\begin{Bmatrix} 5 + 2 \\ 1 + 4 \end{Bmatrix} = 5;$$

$$f_7 = \min\begin{Bmatrix} S_{75} + f_5 \\ S_{76} + f_6 \end{Bmatrix} = \min\begin{Bmatrix} 6 + 5 \\ 8 + 4 \end{Bmatrix} = 11;$$

$$f_9 = \min\{S_{95} + f_5\} = \min\{12 + 5\} = 17;$$

$$f_8 = \min\begin{Bmatrix} S_{86} + f_6 \\ S_{89} + f_9 \end{Bmatrix} = \min\begin{Bmatrix} 7 + 4 \\ 6 + 17 \end{Bmatrix} = 11;$$

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

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

$$f_{10} = \min\begin{Bmatrix} 4 + f_7 \\ 3 + f_8 \\ 11 + f_9 \end{Bmatrix} = \min\begin{Bmatrix} 4 + 11 \\ 3 + 11 \\ 11 + 17 \end{Bmatrix} = 14.$$

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

$$f_8 = \min\begin{Bmatrix} 7 + f_6 \\ 6 + f_9 \end{Bmatrix} = \min\begin{Bmatrix} 7 + 4 \\ 6 + 17 \end{Bmatrix} = 11,$$

т.е. $f_j = f_6$ 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.

Пример 2. Для графа (см. рис.1) найти кратчайший путь от вершины 1 до вершины 10, рассматривая граф как неориентированный. Матрица смежности весов графа в этом случае имеет вид:

	1	2	3	4	5	6	7	8	9	10
1	-	**3**	**4**	**2**	-	-	-	-	-	-
2	**3**	-	-	-	-	**3**	-	-	-	-
3	**4**	-	-	-	-	**6**	-	-	-	-
4	**2**	-	-	-	**5**	**2**	-	-	-	-
5	-	-	-	**5**	-	**1**	**6**	-	**12**	-
6	-	**3**	**6**	**2**	**1**	-	**8**	**7**	-	-
7	-	-	-	-	**6**	**8**	-	-	-	**4**
8	-	-	-	-	-	**7**	-	-	**6**	**3**
9	-	-	-	-	**12**	-	-	**6**	-	**11**
10	-	-	-	-	-	-	**4**	**3**	**11**	-

Определим выражения для функции f_i: $f_1 = 0$;

$$f_2 = \min\begin{Bmatrix} f_1 + 3; \\ f_6 + 3; \end{Bmatrix} \quad f_3 = \min\begin{Bmatrix} f_1 + 4; \\ f_6 + 6; \end{Bmatrix} \quad f_4 = \min\begin{Bmatrix} f_1 + 2; \\ f_5 + 5; \\ f_6 + 2; \end{Bmatrix}$$

$$f_5 = \min\begin{Bmatrix} f_4 + 5; \\ f_6 + 1; \\ f_7 + 6; \\ f_9 + 12; \end{Bmatrix} \quad f_6 = \min\begin{Bmatrix} f_2 + 3; \\ f_3 + 6; \\ f_4 + 2; \\ f_5 + 1; \\ f_7 + 8; \\ f_8 + 7; \end{Bmatrix}$$

$$f_7 = \min\begin{Bmatrix} f_5 + 6; \\ f_6 + 8; \\ f_{10} + 4; \end{Bmatrix}$$

$$f_8 = \min\begin{Bmatrix} f_6 + 7; \\ f_9 + 6; \\ f_{10} + 3; \end{Bmatrix} \quad f_9 = \min\begin{Bmatrix} f_5 + 12; \\ f_8 + 6; \\ f_{10} + 11; \end{Bmatrix} \quad f_{10} = \min\begin{Bmatrix} f_7 + 4; \\ f_8 + 3; \\ f_9 + 11. \end{Bmatrix}$$

Указанные целевые функции представляют собой систему линейных алгебраических уравнений (в примере имеется 10 уравнений и 10 неизвестных). Рассмотрим один из вариантов ее решения, учитывая, что $f_1 = 0$. Тогда имеем

$$f_2 = 3; \qquad f_3 = 4; \qquad f_4 = \min \begin{cases} 2; \\ f_5 + 5 = 2; \\ f_6 + 2; \end{cases}$$

$$f_5 = \min \begin{cases} 7; \\ f_6 + 1; \\ f_9 + 12; \\ f_7 + 6; \end{cases} = \min \begin{cases} 7; \\ f_6 + 1; \end{cases} \qquad f_6 = \min \begin{cases} 6; \\ 10; \\ 4; \\ f_5 + 1 \\ f_7 + 8; \\ f_8 + 7; \end{cases} = 4;$$

Подставив выражение для f_6 в f_5, получим

$$f_5 = \min \begin{cases} 7; \\ 4 + 1 \\ f_5 + 1 + 1. \end{cases} = 5$$

Тогда $f_7 = 11$;

$$f_8 = \min \begin{cases} 11; \\ f_9 + 6; \end{cases} \qquad f_9 = \min \begin{cases} 17; \\ f_8 + 6; \end{cases} \qquad f_{10} = \min \begin{cases} 15; \\ f_8 + 3; \\ f_9 + 11. \end{cases}$$

Подставляя f_9 в f_8, получаем

$$f_8 = \min \begin{cases} 11; \\ 17 + 6 \\ f_8 + 6 + 6. \end{cases} = 11;$$

Окончательно имеем $f_9 = 17$; $f_{10} = 14$. Таким образом кратчайший путь между вершинами 1 и 10 равен: (1, 4, 6, 8, 10).

Метод Дейкстры

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

Идея алгоритма следующая. 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 следующую 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).

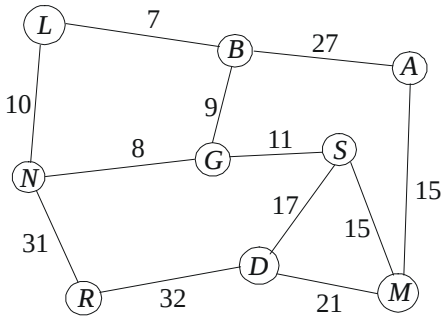


Рис.2. Взвешенный неориентированный граф

Найти кратчайший путь на графе от вершины *L* до вершины *D* (рис.2).

Запишем алгоритм в виде последовательности шагов.

Шаг 1. Определяются расстояния от начальной вершины *L* до всех остальных невыбранных вершин (табл.1).

Таблица 1

Длина пути до выбранной вершины	Выбранная вершина	Невыбранные вершины							
		B	*G*	*N*	*R*	*D*	*S*	*M*	*A*
0	*L*	7	∞	10	∞	∞	∞	∞	∞
7	*B*								

Шаг 2. Выбирается наименьшее расстояние от *L* до *B*. Найденная вершина *B* принимается за вновь выбранную. Найденное наименьшее расстояние добавляется к длинам ребер от новой вершины *B* до всех остальных. Выбирается минимальное расстояние от вершины *B* до *N*. Новая вершина *N* принимается за выбранную (табл.2).

Таблица 2

Длина пути до выбранной вершины	Выбранная вершина	Невыбранные вершины							
		B	*G*	*N*	*R*	*D*	*S*	*M*	*A*
0	*L*	7	∞	10	∞	∞	∞	∞	∞
7	*B*	*	16	10	∞	∞	∞	∞	34
10	*N*								

Знак * означает, что вершина удаляется из списка невыбранных, т.е. в дальнейшем данный столбец таблицы не рассматривается. В каждой строке особо выделяется найденный минимальный путь.

Шаг 3. Определяются расстояния от вершины *N* до всех оставшихся (за исключением *L* и *B*) (табл.3).

Таблица 3

Длина пути до выбранной вершины	Выбранная вершина	Невыбранные вершины							
		B	*G*	*N*	*R*	*D*	*S*	*M*	*A*
0	*L*	7	∞	10	∞	∞	∞	∞	∞
7	*B*	*	16	10	∞	∞	∞	∞	34
10	*N*	*	16	*	41	∞	∞	∞	34
16	*G*								

Расстояние от вершины *L* через вершину *N* до вершины *G* равно 18. Это расстояние больше, чем расстояние *LB* + *BG* = 16, поэтому оно не учитывается в дальнейшем. Для наглядности в дальнейшем вместо знака "∞" будем ставить знак "-". Продолжая аналогич-

ные построения, составим табл.4. Таким образом, найдена длина кратчайшего пути между вершинами *L* и *D* (44 условных единицы).

Таблица 4

Длина пути до выбранной вершины	Выбранная вершина	Невыбранные вершины							
		B	*G*	*N*	*R*	*D*	*S*	*M*	*A*
0	*L*	7	-	10	-	-	-	-	-
7	*B*	*	16	10	-	-	-	-	34
10	*N*	*	16	*	41	-	-	-	34
16	*G*	*	*	*	41	-	16+11 =27	-	34
27	*S*	*	*	*	41	27+17 =44	*	27+15=42	34
34	*A*	*	*	*	41	44	*	42 [34+15]	*
41	*R*	*	*	*	*	44 [41+32]	*	42	*
42	*M*	*	*	*	*	44 [42+21]	*	*	*

Траектория пути определяется следующим образом. Осуществляется обратный просмотр от конечной вершины к начальной. Просматривается столбец, соответствующий вершине, снизу вверх и фиксируется первое появление минимальной величины (табл.4). В столбце, соответствующем вершине *D*, впервые минимальная длина 44 появилась снизу в четвертой строке. В этой строке указана вершина *S*, к которой следует перейти, т.е. следующим нужно рассматривать столбец, соответствующий вершине *S*.

В этом столбце минимальная длина, равная 27, указывает следующую вершину *G*, к которой следует перейти, и т.д. Таким образом, получаем траекторию пути: (*L*, *B*, *G*, *S*, *D*).

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 $C_k[i, j]$ ($k = 0,\ldots,N$). 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) этот 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) этот 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 опять же оптимальным способом, и 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 $i = 1,2,...,n$ и $c_{ij} = \infty$, если в графе отсутствует дуга $(x_i\ x_j)$.

Присвоение 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.

Шаг 4. Если $k = n$, то матрица $[c_{ij}]$ дает длины всех кратчайших путей.

Остановка. Иначе перейти к шагу 2.

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 $N*k$, 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 $B[i, 1]$, $B[i, 2]$, …, $B[i, k]$ для всех i, упорядочены в возрастающем порядке). Следовательно, можно действовать следующим образом. Пусть уже найдены какие-то кратчайшие пути. Тогда, чтобы найти следующий по длине, удлиним каждый из уже полученных на одно ребро. Найдем кратчайший из них, причем оканчивающийся на вершину, до которой найдено менее k путей. Его можно вносить в таблицу результата.

Алгоритм Йена можно свести к последовательности шагов.

Присвоение начальных значений.

Шаг 1. Найти P^1. Присвоить $k = 2$. Если существует только один кратчайший путь P^1, включить его в список L_0 и перейти к шагу 2. Если таких путей несколько, но меньше, чем K, 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. Остановка.

Нахождение отклонений.

Шаг 2. Найти все отклонения $P_i^k(k - 1)$-го кратчайшего пути P^{k-1} для всех $i = 1, 2, …, q_{k-1}$, 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 ($j = 1, 2,…, k - 1$). Если да, то присвоить $c(x_i^{k-1}, x_{i+1}^j) = \infty$; иначе ничего не изменять. (При выполнении алгоритма вершина x_1 обозначается 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},..., 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 $R_i^k(s, x^{-1}, x^{-1},\ldots, x^{-1})$ 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 остановка. 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 действия, аналогичные 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 остановка.

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 Мин Ст(s, v, k) наименьшую стоимость проезда из s в v менее чем с k пересадками.

Мин Ст(s, v, k + 1) = min(Мин Ст(s, v, k), Мин Ст(s, i, k) + $A[i][v]$), (i = 1,..., n).

Искомым ответом является Мин Ст(s, i, n) для всех i = 1,..., 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 хранится 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 МинСт(s, i, k) наименьшую стоимость проезда из s в i менее чем с k пересадками. Тогда выполняется следующее соотношение:

```
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
    {x[i] = МинСт(s,i,k+1)}
  end;
  k := k + 1;
end;
```

Лабораторное задание

Алгоритм выполнения 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 (Приложение 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 (Приложение 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**.

Системы работают в диалоговом режиме с использованием "меню". Вся необходимая поясняющая информация отображается во время работы на экране монитора;

3) изучить алгоритмы работы с графовыми структурами.

Путь к файлу: D:\ИПОВС\АиСД\KRATPUT**Grafmod**.

Данная программа предназначена для демонстрации работы алгоритмов и облегчения понимания терминов теории графов (рис.3).

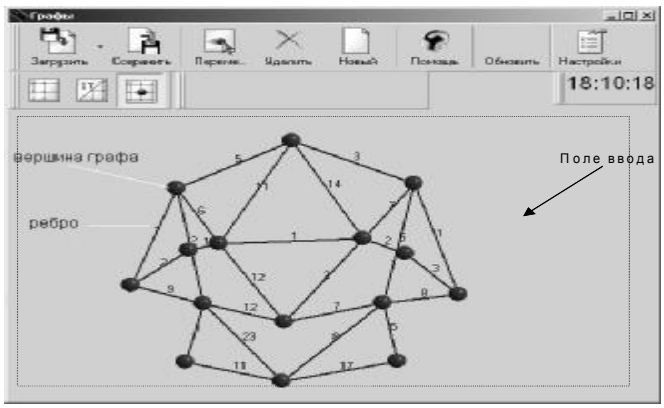


Рис.3. Экранная форма

Программа позволяет вводить, редактировать, сохранять графы. Реализуются алгоритмы построения эйлерова цикла, нахождения кратчайшего пути, нахождение пути, содержащего наименьшее количество вершин, решение задачи о раскраске вершин, "задачи коммивояжера".

Для ввода вершины следует щелкнуть левой кнопкой мыши в свободное от кнопок поле. Для ввода ребра щелкнуть правой кнопкой мыши на одной вершине, затем на другой. Для ввода длины ребра щелкнуть левой кнопкой мыши на введенном ребре. При перемещении вершины нужно нажать кнопку "Переместить", щелкнуть левой кнопкой мыши на перемещаемую вершину, затем на свободное место, куда необходимо переместить вершину, и нажать кнопку "Переместить". Для удаления вершины нажать на кнопку "Удалить", затем щелкнуть левой кнопкой мыши на вершинах, которые нужно удалить и еще раз нажать кнопку "Удалить".

При включенной кнопке "Выравнивать по сетке" новые вершины будут автоматически выравниваться по координатной сетке.

Третья панель служит для запуска алгоритмов программы. Первая кнопка запускает алгоритм решения "задачи коммивояжера". При нажатии на вторую кнопку программа раскрасит граф минимальным количеством цветов. Если выбрать две различные вершины (щелчком левой кнопки мыши) и нажать на третью кнопку, то программа найдет путь с

наименьшим количеством вершин, а если четвертую, то найдет кратчайший путь между вершинами. При нажатии на пятую кнопку построится эйлеровый цикл;

4) ознакомиться с алгоритмами Флойда, Йена, Беллмана - Форда.

Требования к отчету

Отчет должен содержать:

1) конспект лабораторной работы;
2) схемы алгоритмов определения в графе кратчайшего расстояния между заданными вершинами для метода динамического программирования и метода Дейкстры;
3) результаты выполнения работы;
4) выводы по работе.

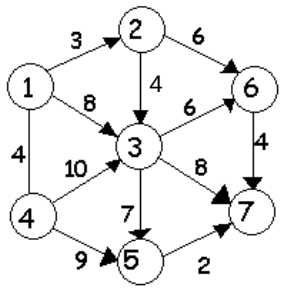
Контрольные вопросы

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

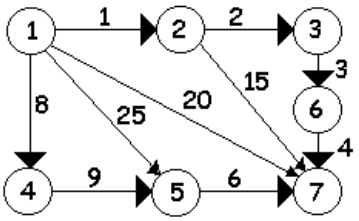
Приложение 1

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

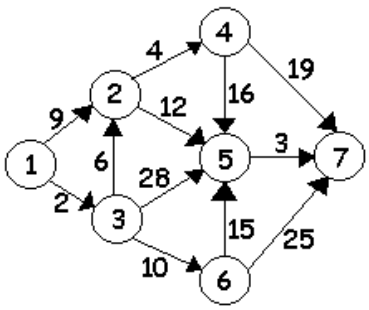
1; 13



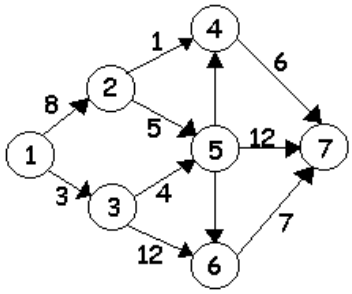
2; 14



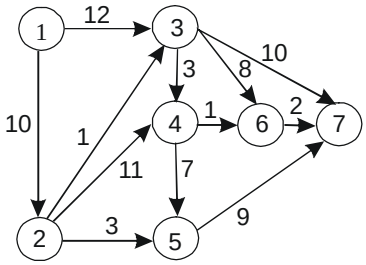
3; 15



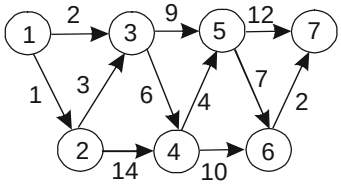
4; 16



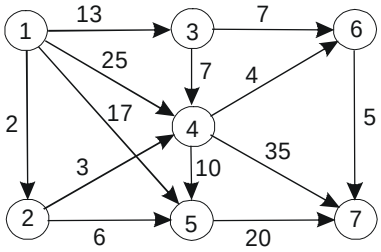
5; 17



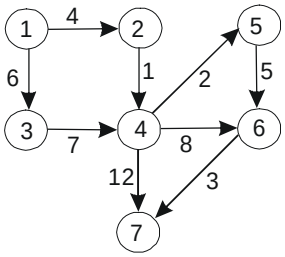
6; 18



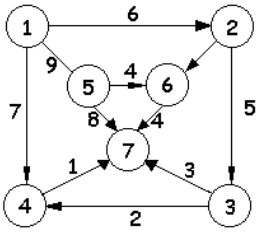
7; 19



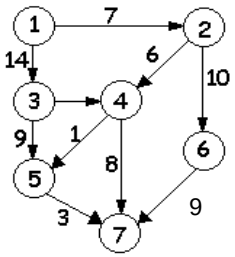
8; 20



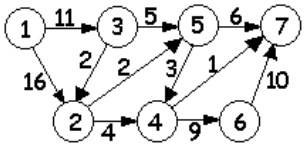
9; 21



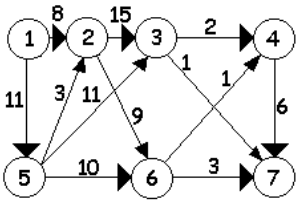
10; 22



11; 23

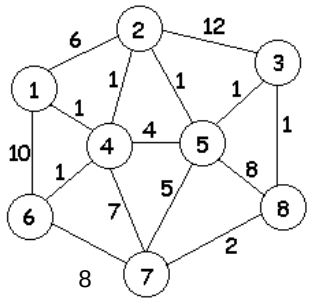


12, 24

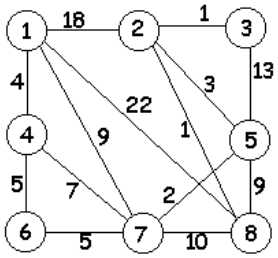


Задание 2. Найти кратчайший путь между тремя парами вершин методом Дейкстры.

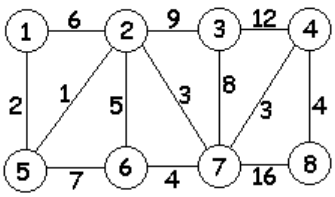
1; 13



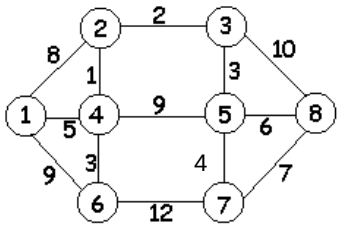
2; 14



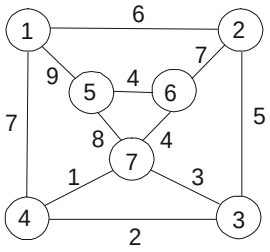
3; 15



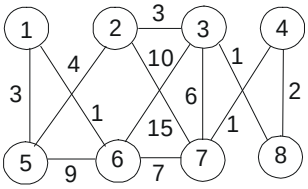
4; 16



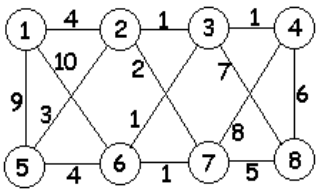
5; 17



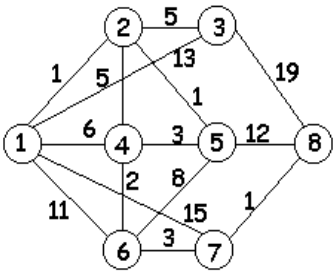
6; 18



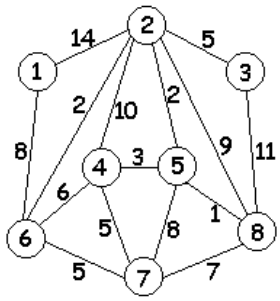
7; 19



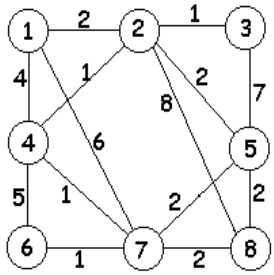
8; 20



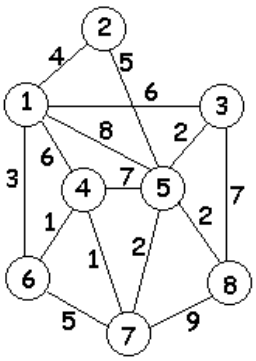
9; 21



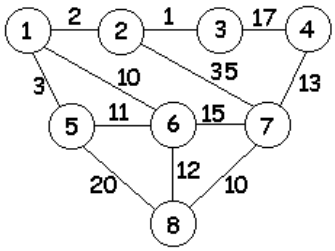
10; 22



11; 23



12; 24



Решить задачу

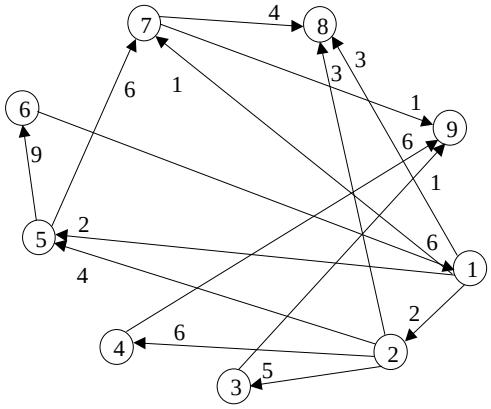
Дана карта автомобильных дорог (рис.П1.1). Найти кратчайший путь, учитывая длины дорог, из Ярославля до Новосибирска. Для построения пути использовать только те города, которые отмечены флажком.

Рис.П1.1. Карта автомобильных дорог

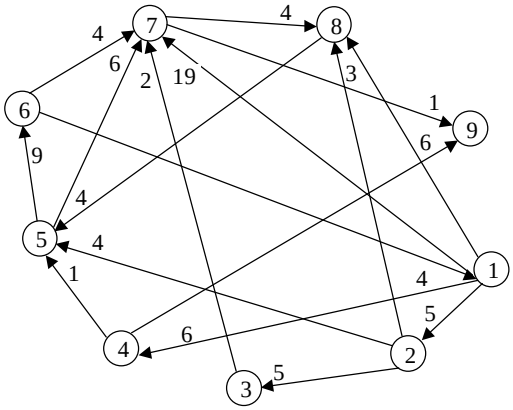
Приложение 2

Составить программу для нахождения кратчайшего пути между двумя выбранными вершинами.

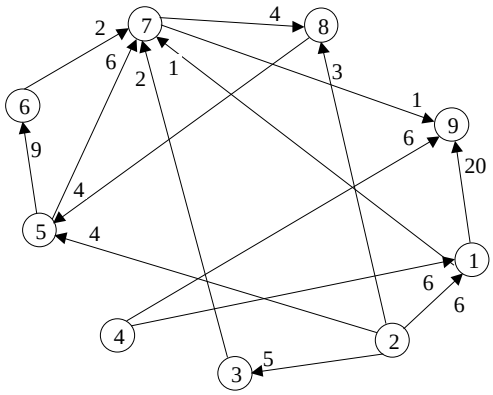
1; 7; 13



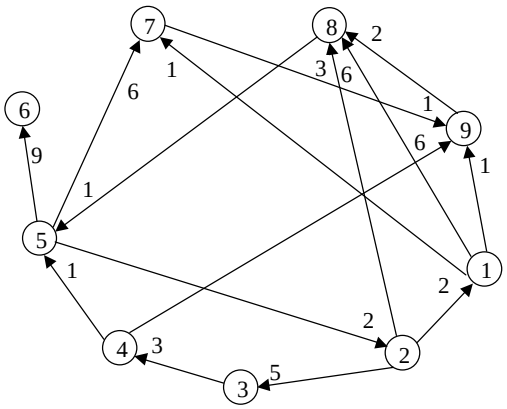
2; 8; 14



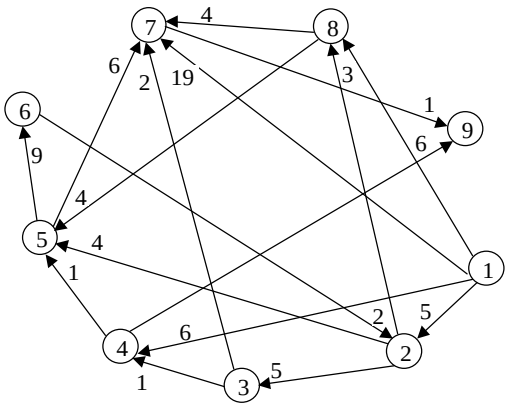
3; 9; 15



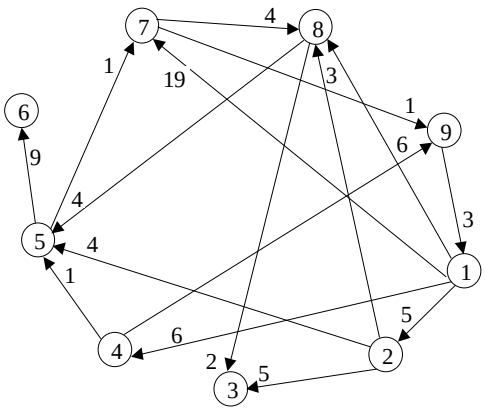
4; 10; 16



5; 11; 17



6; 12; 18



Лабораторная работа № 6. Эвристические алгоритмы

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

Продолжительность работы - 2 ч.

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 (соединения) элементов интегральных схем (ИС). Особенность алгоритма состоит в следующем.

	1	
1	***A***	**1**
	1	

Рис.1. Направления распространения волны

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 конечный элемент.

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

Рис.2. Второй шаг волнового алгоритма

Траектория пути определяется обратным просмотром, от конечного к начальному. При этом разработчик задает приоритеты движения:

Вверх, Вниз, Влево, Вправо.

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 путевых координат должны уменьшаться.

Пример 1. Пусть задан лабиринт, где запрещенные элементы заштрихованы (рис.3). Найти путь между элементами *A* и *B*.

На первом этапе от элемента *A* распространяется волна до тех пор, пока она не достигнет конечного элемента *B*. На втором этапе выбираются приоритеты движения от конечной точки *B* к начальной *A*. Приоритеты выбираются 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).

6		**10**	**9**	**8**		**10**			
5				**7**		**9**	**10**		
4			**5**	**6**		**8**	**9**	***B***	
3	**2**		**4**	**5**		**7**	**8**		**10**
	1	**2**	**3**	**4**		**6**			**9**
1	***A***	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
2	**1**		**3**				**7**		
3	**2**		**4**	**5**	**6**		**8**	**9**	**10**

Рис.3. Нахождение пути в лабиринте

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, 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.

Существует четыре варианта распространения лучей, которые показаны на рис.4.

Коэффициенты α и β вычисляются следующим образом:

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

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

Исходя из значений α и β выбирается вид распространения лучей.

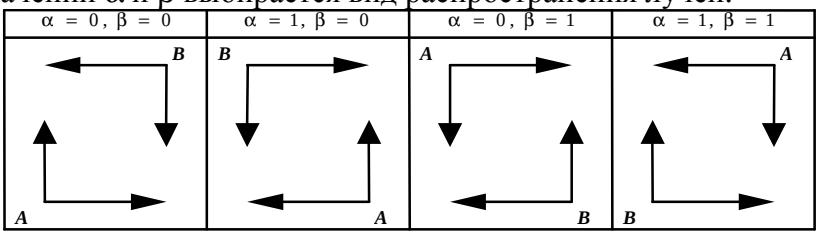


Рис.4. Варианты распространения лучей

Пример 2. Осуществить трассировку элементов A и B двухлучевым алгоритмом (рис.5). Вычислим коэффициенты α и β: $\alpha = 3 - 10 = 7$; $\beta = 9 - 2 = 7$; Принимаем $\alpha = 0$; $\beta = 1$, после чего из рис.4 выбираем направление распространения лучей.

10									
9			**A**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
8			1							**6**
7			2							**5**
6			3				7	6		**4**
5			4					5		**3**
4			5	6	7			4		**2**
3								3		**1**
2								2	1	**B**
1										

1	2	3	4	5	6	7	8	9	10

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.

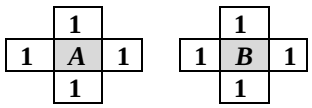


Рис.6. Распространение лучей в четырехлучевом алгоритме

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

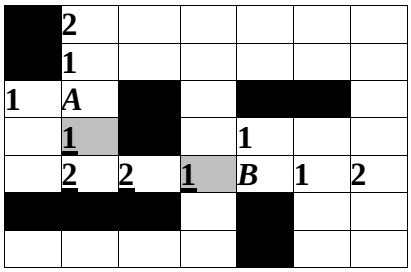


Рис.7. Четырехлучевой алгоритм

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.

$i-1$, $j-1$	i, $j-1$	$i+1$, $j-1$
$i-1$, j	***A***	$i+1$, j
$i-1$, $j+1$	i, $j+1$	$i+1$, $j+1$

Рис.8. Восьмиэлементная окрестность

В маршрутном алгоритме рассматривается восьмиэлементная окрестность исходного элемента (рис.8).

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 определяются восемь значений расстояний, из которых выбирается минимальное. Элемент, для которого значение d оказалось минимальным, считаем элементом трассы. Процесс повторятся 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 4. 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.

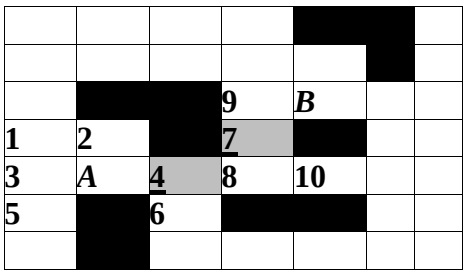


Рис.9. Маршрутный алгоритм

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

$d_1 = \sqrt{(1-5)^2 + (4-5)^2} = \sqrt{17};$
$d_2 = \sqrt{(2-5)^2 + (4-5)^2} = \sqrt{10};$
$d_3 = \sqrt{(1-5)^2 + (3-5)^2} = \sqrt{20};$
$d_4 = \sqrt{(3-5)^2 + (3-5)^2} = \sqrt{8};$
$d_5 = \sqrt{(1-5)^2 + (2-5)^2} = \sqrt{25} = 5;$
$d_6 = \sqrt{(3-5)^2 + (2-5)^2} = \sqrt{13};$
$\min = d_4;$

$d_7 = \sqrt{2};$
$d_8 = \sqrt{5};$
$d_9 = 1;$
$d_{10} = 2;$
$\min = d_7;$
$d_b = 0.$

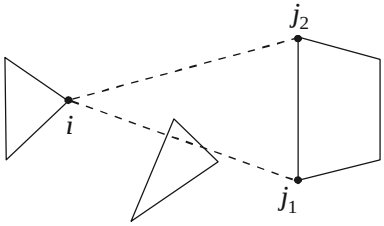
Геометрическая модель задачи о лабиринте

В лабиринте с произвольными препятствиями найти кратчайший путь между заданными точками.

Решение. Так как препятствия на местности образуют многоугольники или другие геометрические фигуры (их с некоторыми погрешностями тоже можно изобразить в виде многоугольников), то кратчайшая трасса будет являться ломаной с узлами в вершинах этих многоугольников. Звено ломаной - это либо сторона многоугольника, либо прямолинейный отрезок, проходящий вне многоугольников и соединяющий две вершины одного и того же или разных многоугольников. Для решения этой задачи нужно построить сеть (ломаную), а также соединить точки s и t (рис.10) с просматриваемыми из них вершинами, если эти точки не являются вершинами многоугольников.

Рис.10. Лабиринт

Формирование сети, т.е. матрицы расстояний C размером n x n (n - общее число вершин всех многоугольников плюс два для учета старта и финиша), представляет собой тройной цикл. Внешний - по i - перебор вершин, откуда осуществляется выбор направления просмотра; средний - по j (j от i + 1 до n, чтобы не повторяться) - это перебор вершин, внутренний по k - это проверка, не пересекает ли k-я сторона какого-либо многоугольника отрезок соединения (рис.11).



(i, j_1) - не входит в сеть,
(i, j_2) - входит в сеть

Рис.11. Выбор направления просмотра

Последнее условие проверяется по стандартным формулам аналитической геометрии: выписывается уравнение прямой, проходящей через *i, j*, выписывается уравнение прямой, проходящей через концы отрезка *k*. Решением системы из этих двух уравнений находится точка пересечения и устанавливается, лежит ли точка пересечения внутри рассматриваемых отрезков. Если да, то $d_{ij} = \infty$, конец цикла по *k*, если нет пересечения по окончании цикла по *k*, то вычисляется евклидово расстояние d_{ij}.

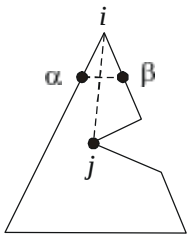


Рис.12. Расположение вершин

Пусть между вершинами *i* и *j* не проходит никакой стены, а *j* из *i* не просматривается (рис.12). Чтобы преодолеть эту трудность, нужно ввести характеристику *i* угла препятствия g_i, присвоив $g_i = 0$, если $\alpha_i < \pi$ ("вогнутый" угол), или $g_i = 1$, если $\alpha_i > \pi$ ("выпуклый" угол). Так, для угла с вершиной *i* $g_i = 1$, а для угла с вершиной *j* $g_i = 0$. Если крайние вершины x_i и x_{i+3} (x_i, x_{i+1}, x_{i+2}, x_{i+3} - последовательные вершины многоугольника) лежат по одну сторону от прямой, проходящей через соседние вершины x_{i+1}, x_{i+2}, то $g_{i+1} = g_{i+2}$, иначе $g_{i+1} <> g_{i+2}$.

$$(x - x_{i+1})(y_{i+2} - y_{i+1}) - (x_{i+2} - x_{i+1})(y - y_{i+1}) = 0.$$

Если при подстановке в это уравнение точек (x_i, y_i) и (x_{i+3}, y_{i+3}) в левой части получаются числа с одинаковым знаком, то $g_{i+1} = g_{i+2}$, иначе $g_{i+1} <> g_{i+2}$. После этого цикла будут известны все g_i точно или с точностью до наоборот. Остается абсолютно установить g_i хотя бы для одной вершины. Это легко сделать, потому что экстремальная вершина $g_0 = 1$. Теперь можно решать вышеизложенную проблему. Из вершины *i* не просматривается никакая вершина *j*, защищенная углом с вершиной *i*. Чтобы исключить из рассмотрения загороженные вершины, нужно отступить от вершины *i* по сторонам угла на величину ε, заведомо меньшую, чем длина стороны, построив таким образом точки α и β, после чего ввести бинарную величину *B*:

- *B* = 1, если отрезки $\alpha\beta$ и *ij* пересекаются;
- *B* = 0, если отрезки $\alpha\beta$ и *ij* не пересекаются.

Всего имеется четыре возможности (рис.13):

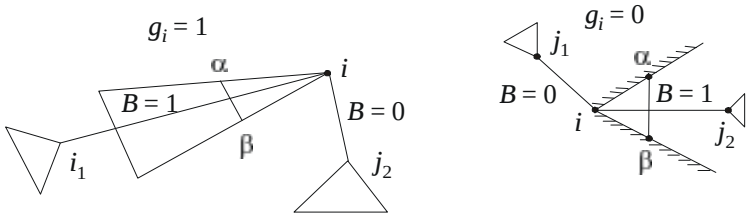


Рис.13. Значения бинарной величины *B*

1) $B = 1$ и $g_i = 0$;
2) $B = 0$ и $g_i = 1$;
3) $B = 1$ и $g_i = 0$;
4) $B = 1$ и $g_i = 1$.

Ясно, что вершина *j* не просматривается - в случаях 2 и 3 (при нечетном $B + g$). Теперь можно построить сеть.

После того как сеть построена, можно приступать к нахождению кратчайших путей (рис.14), воспользовавшись любым из выше рассмотренных алгоритмов (в зависимости от поставленной задачи).

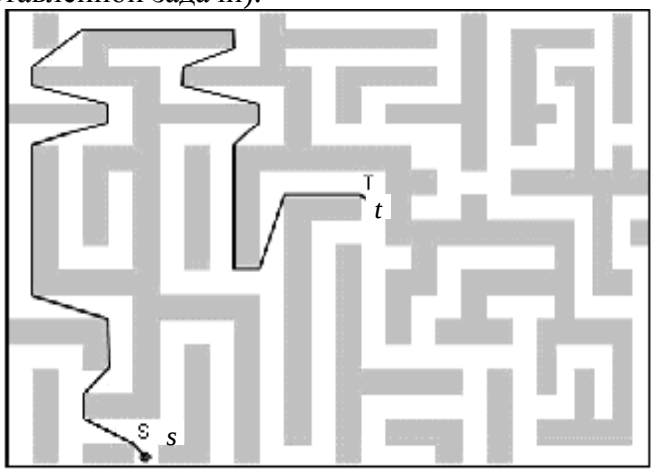


Рис.14. Путь в лабиринте

Алгоритм составления расписания

Предположим, что имеется множество n одинаковых процессоров, обозначенных $P_1, P_2, ..., P_n$, и m независимых заданий $J_1, J_2, ..., J_m$, которые нужно выполнить. Процессоры могут работать одновременно, и любое задание можно выполнять на любом процессоре. Если задание загружено в процессор, оно остается там до конца обработки. Время обработки задания J_i известно и равно $t_i, i = 1,2,...,m.$ Необходимо организовать обработку заданий таким образом, чтобы выполнение всего набора заданий было завершено 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 имеются три процессора и шесть заданий, время выполнения каждого из которых равно:

$t_1 = 2;\ t_2 = 5;\ t_3 = 8;$

$t_4 = 1;\ t_5 = 5;\ t_6 = 1.$

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 своими первоначальными заданиями. При T = 3 процессор P_3 опять 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. Рассмотренное расписание проиллюстрировано на рис.15 временной диаграммой, известной как схема Ганта. Очевидно, что расписание неоптимально. Можно, например, 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.16).

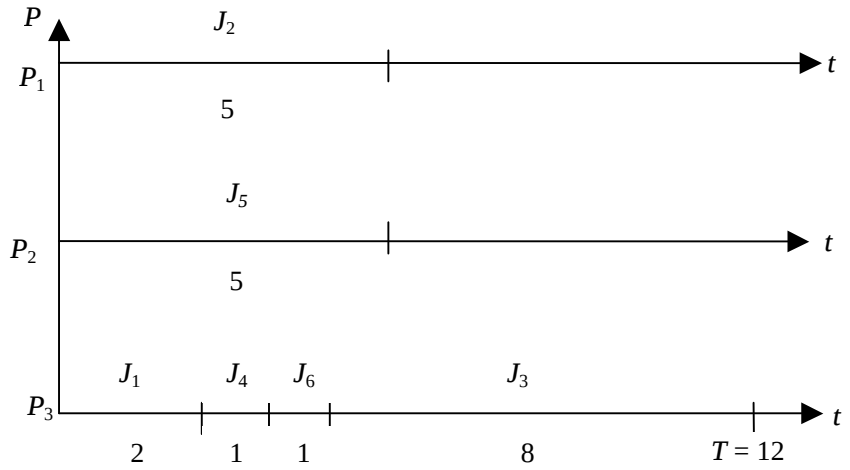


Рис.15. Схема Ганта для расписания L

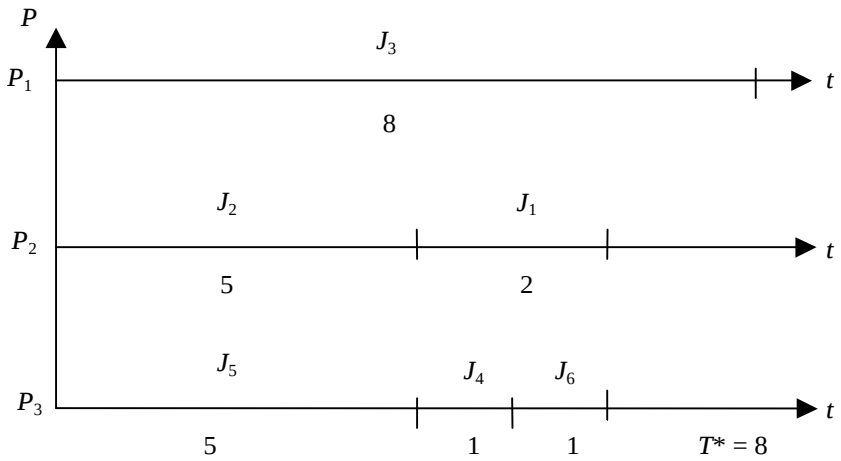


Рис.16. Схема Ганта для оптимального расписания L^*

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 $i = 1,2,...,n$. 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 предметы нельзя дробить на части.

Лабораторное задание

Алгоритм выполнения лабораторной работы следующий:

1) ознакомиться с эвристическими алгоритмами;

2) осуществить трассировку элементов интегральных схем размером 10x10, 20x20, 30x30. Зафиксировать параметры трассировок всеми рассмотренными методами.

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;

3) составить оптимальное расписание работы четырех процессоров, для которых известно $t_1, \ldots, t_{11}$;

4) составить алгоритм оптимальной упаковки 12 предметов размером от 1 до 4 в ящики размером 6;

5) составить программу моделирования эвристического алгоритма (по заданию преподавателя).

6) решить задачи (Приложение 1).

Требования к отчету

Отчет должен содержать:

1) конспект лабораторной работы;

2) схемы волнового и лучевых алгоритмов;

3) результаты выполнения работы;

4) выводы по работе.

Контрольные вопросы

1. Какова теоретическая сложность алгоритмов, рассмотренных в данной работе?

2. Каковы особенности работы волнового, лучевых и маршрутного алгоритмов?

3. Каковы принципы составления оптимального расписания работы параллельных процессоров?

4. В чем заключаются особенности задачи упаковки?

Приложение 1

Решить задачи

Задача 1. Найти выход из произвольной точки лабиринта в саду Хемптон Корт. Отождествив коридоры лабиринта с ребрами, а перекрестки, тупики, входы и выходы - с вершинами, перейти к связному графу, представляющему схему лабиринта (рис.П1.1).

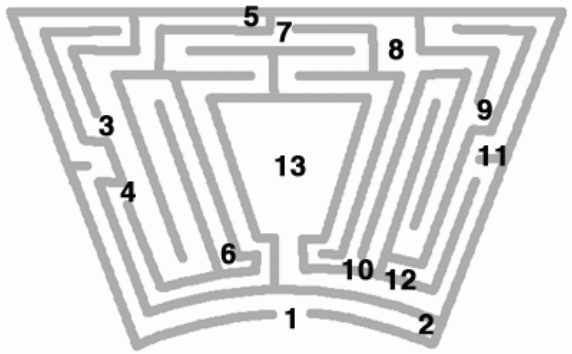
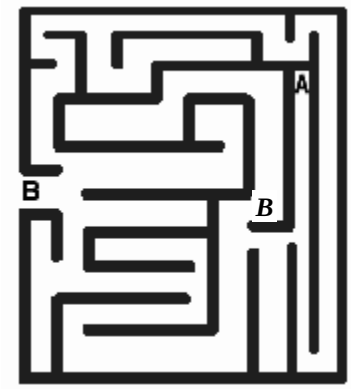


Рис.П1.1. Схема сада Хемптон Корт

Задача 2. Нарисуйте граф, соответствующий лабиринту (рис.П1.2). Найдите путь, по которому можно пройти от пункта *A* до *B* лабиринта, используя предложенные выше алгоритмы.



A

Рис.П1.2. Схема лабиринта

Задача 3 (задача о джипе). Пусть необходимо пересечь на джипе 1000-километровую пустыню, израсходовав при этом минимум горючего. Объем топливного бака джипа 500 литров, 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.

Итак, идея задачи ясна: 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 хватило на возвращение. В точке старта вновь производится полная заправка и делается попытка второй склад продвинуть в пустыню дальше. Но где обустраивать эти склады и сколько горючего оставлять в каждом из них?

Задача 4 (задача о кодовом замке). Пусть кодовый замок состоит из набора N переключателей, каждый из которых может быть в положении "вкл" или "выкл". Замок открывается только при одном наборе положений переключателей, из которых не менее $[N/2]$ (целая часть от $N/2$), 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 $N = 4$.

Литература

1. ***Колдаев В.Д.*** Основы алгоритмизации и программирования: Учеб. пос. / ***Под ред. Л.Г. Гагариной***. - М.: Форум - Инфра - М, 2006.

2. ***Колдаев В.Д., Поддубная Л.М., Полосухин Б.М.*** Методы сортировки. - М.: МИЭТ, 1985.

3. ***Колдаев В.Д., Поддубная Л.М., Полосухин Б.М.*** Лабораторный практикум по курсу "Теория алгоритмов и вычислительные методы". - М.: МИЭТ, 1988.

4. ***Полосухин Б.М., Поддубная Л.М.*** Рекурсивные функции и алгоритмы. - М.: МИЭТ, 1985.

5. ***Кнут Д.*** Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. - М.: Мир, 2000.

6. ***Кормен Т., Лейзерсон Ч., Ривест Р.*** Алгоритмы: построение и анализ. - М.: МЦНМО, 2000.

7. ***Вирт Н.*** Алгоритмы и структуры данных / Пер. с англ. - М.: Мир, 2001.

8. ***Хусаинов Б.С.*** Структуры и алгоритмы обработки данных. Примеры на языке Си: Учеб. пос. - М.: Финансы и статистика, 2004.

9. ***Ахо А., Хопкрофт Дж., Ульман Дж.*** Структуры данных и алгоритмы. - М: СПб: Киев: Вильямс, 2001.

10. ***Шень А.*** Программирование. Теоремы и задачи. - М.: МЦНМО, 2004.

11. ***Мейн М., Савитч У.*** Структуры данных и другие объекты в C++ / Пер. с англ. - М.: Издательский дом "Вильямс", 2002.

ПРИЛОЖЕНИЕ. Работа с динамическими структурами данных

В языках программирования (Pascal, C, C++, др.) существует способ выделения памяти под данные, который называется динамическим. В этом случае память под величины отводится во время выполнения программы. Использование динамических величин предоставляет программисту ряд дополнительных возможностей:

- подключение динамической памяти позволяет увеличить объем обрабатываемых данных;
- если потребность в каких-то данных отпала до окончания программы, то занятую ими память можно освободить для другой информации;
- использование динамической памяти позволяет создавать структуры данных переменного размера.

Работа с динамическими величинами связана с использованием еще одного типа данных - ссылочного типа. Величины, имеющие ссылочный тип, называют указателями.

Указатель содержит адрес поля в динамической памяти, хранящего величину определенного типа. Сам указатель располагается в статической памяти.

Адрес величины - это номер первого байта поля памяти, в котором располагается величина. Размер поля однозначно определяется типом. Величина ссылочного типа (указатель) описывается в разделе описания переменных следующим образом:

Var <идентификатор> : ^<имя типа>;

Примеры описания указателей:

```
Type Mas1 = Array[1..100] Of Integer;
Var      P1 : ^Integer;
         P2 : ^String;
         Pm : ^Mas1;
```

Здесь P1 - указатель на динамическую величину целого типа; P2 - указатель на динамическую величину строкового типа; Pm - указатель на динамический массив, тип которого задан в разделе Type. Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции память выделяется только под статические величины. Указатели - это статические величины, поэтому они требуют описания. Память под динамическую величину, связанную с указателем, выделяется в результате выполнения стандартной процедуры **NEW**. Формат обращения к этой процедуре:

NEW(<указатель>);

Считается, что после выполнения этого оператора создана динамическая величина, имя которой имеет следующий вид:

<имя динамической величины> := <указатель>^

Пусть в программе, в которой имеется приведенное выше описание, присутствуют следующие операторы:

NEW(P1); NEW(P2); NEW(Pm);

После их выполнения в динамической памяти оказывается выделенным место под три величины (две скалярные и один массив), которые имеют идентификаторы:

P1^, P2^, Pm^

Например, обозначение P1^ можно расшифровать так: динамическая переменная, на которую ссылается указатель P1.

Дальнейшая работа с динамическими переменными происходит точно так же, как со статическими переменными соответствующих типов. Им можно присваивать значения, их можно использовать в качестве операндов в выражениях, параметров подпрограмм и пр. Например, если нужно переменной P1^ присвоить число 25, переменной P2^ присвоить значение символа "Write", а массив Pm^ заполнить по порядку целыми числами от 1 до 100, то это делается так:

```
P1^ := 25;
```

```
P2^ := 'Write';
For I := 1 To 100 Do Pm^[I] := I;
```

Кроме процедуры NEW значение указателя может определяться оператором присваивания:

<указатель> := <ссылочное выражение>;

В качестве ссылочного выражения можно использовать:

- указатель;
- ссылочную функцию (т.е. функцию, значением которой является указатель);
- константу Nil.

Nil - это зарезервированная константа, обозначающая пустую ссылку, т.е. ссылку, которая ни на что не указывает. При присваивании базовые типы указателя и ссылочного выражения должны быть одинаковы. Константу Nil можно присваивать указателю с любым базовым типом.

До присваивания значения ссылочной переменной (с помощью оператора присваивания или процедуры NEW) она является неопределенной. Ввод и вывод указателей не допускается.

Пример. Пусть в программе описаны следующие указатели:

```
Var     D, P : ^Integer;
        K : ^Boolean;
```

Тогда допустимыми являются операторы присваивания

D := P; K := Nil;

поскольку соблюдается принцип соответствия типов. Оператор K : = D ошибочен, так как базовые типы у правой и левой части разные.

Если динамическая величина теряет свой указатель, то она становится "мусором". В программировании под этим словом понимают информацию, которая занимает память, но уже не нужна.

Например, в программе, в которой присутствуют описанные выше указатели, в разделе операторов записано следующее:

```
NEW(D);  NEW(P);
{Выделено место в динамической памяти под две целые переменные. Указатели получили соответствующие значения}
D^ := 3; P^ := 5;
{Динамическим переменным присвоены значения}
P := D;
{Указатели P и D стали ссылаться на одну и ту же величину, равную 3}
        WriteLn(P^, D^);   {Дважды напечатается число 3}
```

Таким образом, динамическая величина, равная 5, потеряла свой указатель и стала недоступной. Однако место в памяти она занимает. Это и есть пример возникновения "мусора". На схеме показано, что произошло в результате выполнения оператора P : = D.

В языке Паскаль имеется стандартная процедура, позволяющая освобождать память от данных, потребность в которых отпала. Ее формат:

DISPOSE(<указатель>);

Например, если динамическая переменная P^ больше не нужна, то оператор DISPOSE(P) удалит ее из памяти. После этого значение указателя P становится неопределенным. Особенно существенным становится эффект экономии памяти при удалении больших массивов. Использование динамической памяти позволяет существенно увеличить объем обрабатываемой информации.

Следует четко понимать, что работа с динамическими данными замедляет выполнение программы, поскольку доступ к величине происходит в два шага: сначала ищется указатель, затем по нему - величина. Как это часто бывает, действует "закон сохранения неприятностей": выигрыш в памяти компенсируется проигрышем во времени.

Пример. 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 10 000, заполнить его случайными целыми числами в диапазоне от –100 до 100 и вычислить его среднее значение.

{Язык Turbo Pascal}

```
Program Srednee;
Const NMax = 10000;
Type Diapazon = 1..NMax;
MasInt = Array[Diapazon] Of  Integer;
MasReal = Array[Diapazon] Of  Real;
Var PInt : ^MasInt; PReal : ^MasReal;
I, MidInt : longInt; MidReal : Real; T : Text; S : string;
Begin
      Write('Введите имя файла: '); ReadLn(S);
      Assign(T, S);  Reset(T);  MidReal := 0;  MidInt := 0;
      Randomize;
      NEW(PReal); {Выделение памяти под вещественный массив}
      {Ввод и суммирование вещественного массива}
      While Not Eof (T) Do
Begin ReadLn (T, PReal^[I]);  MidReal := MidReal + PReal^[I]
 End;
   DISPOSE(PReal); {Удаление вещественного массива}
      NEW(PInt); {Выделение памяти под целый массив}
      {Вычисление и суммирование целого массива}
      For I := 1 To NMax Do
      Begin PInt^[I] := -100 + Random(201);
               MidInt := MidInt + PInt^[I] End;
            {Вывод средних значений}
           WriteLn('среднее целое равно: ', MidInt Div NMax);
   WriteLn('среднее вещественное равно: ', (MidReal / NMax) : 10 : 6)
   End.
```

// Язык 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); // преобр.  строки в вещ. 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);
}
```

Списки

Пусть в процессе физического эксперимента многократно снимаются показания прибора и записываются в компьютерную память для дальнейшей обработки. Заранее неизвестно, сколько будет произведено измерений. Если для обработки таких данных не использовать внешнюю память (файлы), то разумно расположить их в динамической памяти. Во-первых, динамическая память позволяет хранить больший объем информации, чем статическая. А во-вторых, в динамической памяти эти числа можно организовать в связанный список, который не требует предварительного указания количества чисел, подобно массиву. Что же такое "связанный список"? Схематически он показан на рис.П2.1.

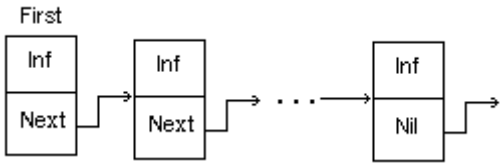


Рис.П2.1. Схематическое изображение связного списка

Здесь Inf - информационная часть звена списка (величина любого простого или структурированного типа, кроме файлового), Next - указатель на следующее звено списка; First - указатель на заглавное звено списка.

Согласно определению, список располагается в динамически распределяемой памяти; в статической памяти хранится лишь указатель на заглавное звено. Структура, в отличие от массива, является действительно динамической: звенья создаются и удаляются по мере необходимости в процессе выполнения программы.

Для объявления списка сделано исключение: указатель на звено списка объявляется раньше, чем само звено. В общем виде объявление выглядит так:

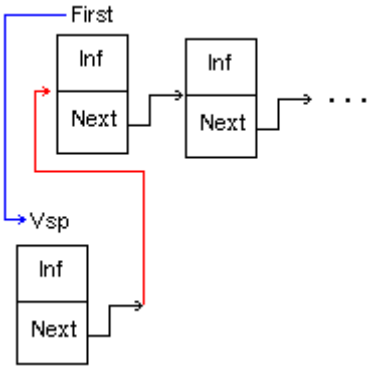
```
Type      U = ^Zveno;
          Zveno = Record Inf : BT; Next: U End;
```

Здесь BT - некоторый базовый тип элементов списка.

Указатели подразделяются на однонаправленные**,** двунаправленные, кольцевые.

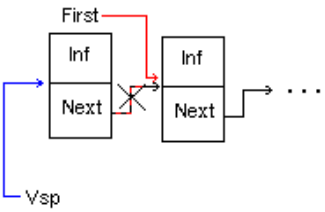
Реализуем набор операций над списками в виде модуля. Подключив этот модуль, можно решить большинство типовых задач на обработку списка.

1. Добавление звена в начало списка.



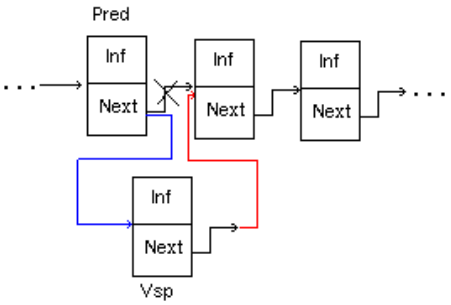

```
{Процедура добавления звена в начало списка; в x содержится добавляемая информация}
Procedure V_Nachalo(Var First : U;  X : BT);
    Var Vsp : U;
    Begin
        New(Vsp);
        Vsp^.Inf := X;
        Vsp^.Next := First;
{То звено, что было заглавным, становится вторым по счету}
First := Vsp;
{Новое звено становится заглавным}
    End;
```

2. Удаление звена из начала списка.




```
{Процедура удаления звена из начала списка; в x содержится информация из удаленного звена}
Procedure Iz_Nachala(Var First: U; Var X: BT);
    Var Vsp : U;
    Begin
        Vsp := First;
{Забираем ссылку на текущее заглавное звено}
        First := First^.Next;
{То звено, что было вторым по счету, становится заглавным}
    X := Vsp^.Inf;
{Забираем информацию из удаляемого звена}
    Dispose(Vsp);    {Уничтожаем звено}
    End;
```

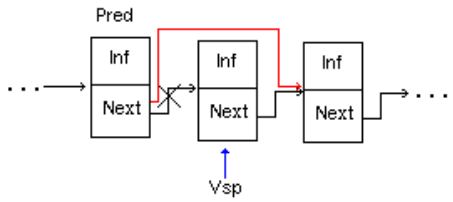
3. Добавление звена в произвольное место списка, отличное от начала.




```
{Процедура добавления звена в список после звена, на которое ссылается указатель Pred; в x содержится информация для добавления}
    Procedure V_Spisok(Pred : U; X : BT);
    Var Vsp : U;
    Begin
      New(Vsp);  {Созда-
```

```
ем пустое звено}
      Vsp^.Inf := X; {Заносим информацию}
      Vsp^.Next := Pred^.Next; {Теперь это звено ссылается на то, что было следом за звеном Pred}
      Pred^.Next := Vsp; {Теперь новое звено встало вслед за звеном Pred}
    End;
```

4. Удаление звена из произвольного места списка, отличного от начала.




```
{Процедура удаления звена из списка после звена, на которое ссылается указатель Pred; в x содержится информация из удаленного звена} Procedure Iz_Spiska (Pred : U; Var X : BT);
    Var Vsp : U;
    Begin
 Vsp := Pred^.Next; {Забираем ссылку на удаляемое звено}{Удаляем звено из списка, перенаправив ссылку на следующее
  за ним звено}
      Pred^.Next := Pred^.Next^.Next;
      X := Vsp^.Inf; {Забираем информацию из удаляемого звена}
   Dispose(Vsp); {Уничтожаем звено}
    End;
```

Пример 1. Составить программу, которая на основе заданного списка формирует два других, помещая в первый из них положительные, а во второй - отрицательные элементы исходного списка.

При реализации алгоритма будем использовать подпрограммы разработанного модуля.

{Программа на Turbo Pascal}

```
Program Ex_sp_1;
Uses Spisok;
Var S1, S2, S3, V1, V2, V3 : U; A : BT; I, N : Byte;
Begin
  Randomize;
  N := 1 + Random(20);
  S1 := Nil; A := -100 + Random(201);
  V_Nachalo(S1, A); V1 := S1;
```

```
   For I := 2 To N Do
   Begin A := -100 + Random(201); V_Spisok(V1, A);
            V1 := V1^.Next End;
   WriteLn('Исходный список: '); Print(S1);
   V1 := s1;  S2 := Nil; S3 := Nil;
   While V1 <> Nil Do
   Begin
      If V1^.Inf > 0
      Then If S2 = Nil
        Then Begin V_Nachalo(S2, V1^.Inf); V2 := S2 End
        Else Begin V_Spisok(V2, V1^.Inf); V2 := V2^.Next End;
      If V1^.Inf < 0
      Then If S3 = Nil
          Then Begin V_Nachalo(s3, V1^.Inf); V3 := S3 End
          Else Begin V_Spisok(V3, V1^.Inf); V3 := V3^.Next End;
      V1:= V1^.Next
   End;
   WriteLn('Результирующий список из полож. элементов: '); Print(S2);
   WriteLn('Результирующий список из отриц. элементов: '); Print(S3);
   Ochistka(S1); Ochistka(S2); Ochistka(S3);
End.
```

// Программа на 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 полож.  элементов: \n";
 Print(S2);
 cout << "Результирующий список из отриц.  элементов: \n";
 Print(S3);
 S1=Ochistka(S1); S2=Ochistka(S2); S3=Ochistka(S3);
}
```

Пример 2. 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.

{ Turbo Pascal, файл ST2.PAS }

```
 Program St2;
 Uses Spisok, Stack;
 Const  Znak  =  ['+',  '-',
'*', '/'];
 Var S, S1 : String;
     T : Text;
     I, N : Byte;
     X,  Y  :  BT;  Code  :
Integer;
     NS : U;
 Begin
   Write('Введите   имя
файла:');
ReadLn(S1);
   Assign(T,S1);      Re-
Set(T);
   NS := Nil;
   While Not Eof(T) Do
    Begin
 ReadLn(T, S);  I := 1;
  While I <= Length(S)
Do
       Begin
         If S[I] In ['0'..'9']
         Then
         Begin
          N := I;
          While   S[I]   In
['0'..'9'] Do
           I := I + 1;
 Val(Copy(S, N, I - N),
X, Code);
         V_Stack(NS, X);
         End
         Else
```

/* C++, файл ST2.CPP */

```
#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;
```

```
      If S[I] In Znak
      Then
       Begin
      Iz_Stack(NS, X);
      Iz_Stack(NS, Y);
     Case S[I] Of
      '+' : X := X + Y;
      '-' : X := Y - X;
      '*' : X := X * Y;
      '/' : X := Y Div X
         End;
     V_Stack(NS, X)
       End;
      I := I + 1
     End;
     Iz_Stack(NS, X);
   WriteLn(S, ' = ', X);
    End
  End.
```

```
                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 3. 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 X2, X3, X5, 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.

{Язык Pascal}

```
Program Example;
Uses Spisok2;
Var X2, X3, X5 : U; X : BT; I, N : Word;
Procedure PrintAndAdd(T : BT);
Begin
  If T <> 1 Then Write(T : 6);
  V_Och(X2, T * 2); V_Och(X3, T * 3); V_Och(X5, T * 5);
End;
Function Min(A, B, C : BT) : BT;
Var Vsp : BT;
Begin
  If A < B Then Vsp := A Else Vsp := B;
  If C < Vsp Then Vsp := C;
  Min := Vsp
End;
Begin
  X2 := Nil; X3 := Nil; X5 := Nil;
  Write('Сколько чисел напечатать? '); ReadLn(N);
  PrintAndAdd(1);
  For I := 1 To N Do
  Begin
        X := Min(X2^.Inf, X3^.Inf, X5^.Inf);
        PrintAndAdd(X);
```

```
            If X = X2^.Inf Then Iz_Och(X2, X);
            If X = X3^.Inf Then Iz_Och(X3, X);
            If X = X5^.Inf Then Iz_Och(X5, X);
    End;
    Ochistka(X2); Ochistka(X3); Ochistka(X5);
    WriteLn
End.
```

// 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;
}
```