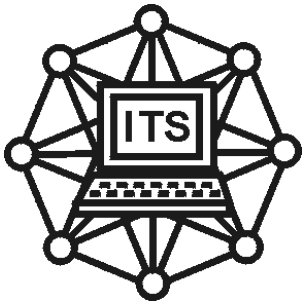
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНА МЕТАЛУРГІЙНА АКАДЕМІЯ УКРАЇНИ



МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт з дисципліни

«ТЕОРІЯ АЛГОРИТМІВ»

для студентів спеціальності 122 - «Комп'ютерні науки»
денної форми навчання

Дніпро НМетАУ 2019

УДК 681.3.07

Методичні вказівки до виконання лабораторних робіт з дисципліни «Теорія алгоритмів» для студентів спеціальності 122 – «Комп'ютерні науки» денної форми навчання/Укл. А.О. Журба. - Дніпро: НМетАУ, 2019 – 51 с.

Методичні вказівки містять навчально-методичні матеріали з дисципліни «Теорія алгоритмів»; містять теоретичні положення з теорії алгоритмів, приклади програм мовою С++, завдання з питань використання базових алгоритмів для самостійного виконання.

Призначені для студентів спеціальності 122 денної форми навчання, а також для слухачів курсів підвищення кваліфікації, студентів і аспірантів інших спеціальностей.

Укладачі: А.О. Журба, канд. техн. наук, доцент,

Друкується за авторською редакцією.

Затверджено на засіданні кафедри інформаційних технологій і систем, протокол № 9 від 06.03.2019 р.

Відповідальний за випуск О.І. Михальов, д-р техн. наук, проф.

Рецензент : О.І. Дерев'янко, канд. техн. наук, доцент (ДНУ)

Національна металургійна академія України.
49600, Дніпро, пр. Гагаріна, 4

СОДЕРЖАНИЕ

Лабораторная работа №1

АНАЛИЗ ВРЕМЕННОЙ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ

Цель работы: изучение средств анализа временной эффективности алгоритмов.

Теоретические сведения

Алгоритм - набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий.

Так как одним из основных показателей качества алгоритма является время его выполнения, то целесообразно проводить анализ временной эффективности алгоритмов.

Время выполнения большинства алгоритмов зависит от размера входных данных. Поэтому принято описывать эффективность алгоритма в виде функции от параметра n, связанного с размером входных данных.

Единицы измерения времени выполнения алгоритма

1. Общепринятые единицы измерения времени (с, мс и.т.д.) такой подход имеет недостатки, т.к. результаты измерений будут зависеть от посторонних факторов.

2. Подсчет количества раз, сколько выполняется каждая операция алгоритма – данный подход сложен.

3. Вычисление количества раз, сколько выполняются базовые операции алгоритма (наиболее важные операции в алгоритме, которые вносят наибольший вклад в общее время выполнения алгоритма). В качестве базовых выбирают наиболее длительные по времени операции, выполняемые во внутреннем цикле алгоритма.

Эффективность алгоритма в разных случаях

Существует большое количество алгоритмов, время выполнения которых зависит не только от размера входных данных, но и от особенностей входных данных.

Эффективность алгоритма в ***наихудшем случае*** – эффективность для наихудшей совокупности входных данных размером n; для такой совокупности входных данных время работы алгоритма будет максимально возможным.

Эффективность алгоритма в ***наилучшем случае*** - эффективность для наилучшей совокупности входных данных размером n; для такой совокупности входных данных время работы алгоритма будет минимально возможным.

Эффективность алгоритма в ***среднем случае*** – эффективность для такой совокупности входных данных размером n, для которой время работы алгоритма будет средним. Эффективность алгоритма в среднем случае нельзя

получить усреднив его эффективности для наихудшего и наилучшего случае. В алгоритме проверяется приблизительно половина входных данных.

Пример.

Дан массив из семи целых чисел: 8 4 7 2 10 5 6. Необходимо найти заданный ключ по следующему алгоритму: последовательно каждый элемент массива сравнивать с ключом. Как только будет найден искомый ключ, алгоритм прекращает работу.

Рассмотрим следующие варианты:

8	4	7	2	10	5	6

1. Поиск ключа 6 - ХУДШИЙ СЛУЧАЙ
Необходимо выполнить 7 сравнений:
6 и 8 - ключ 6 не найден, продолжаем поиск
6 и 4 - ключ 6 не найден, продолжаем поиск
6 и 7 - ключ 6 не найден, продолжаем поиск
6 и 2 - ключ 6 не найден, продолжаем поиск
6 и 10 - ключ 6 не найден, продолжаем поиск
6 и 5 - ключ 6 не найден, продолжаем поиск
6 и 6 - ключ 6 найден. Поиск завершен.

2. Поиск ключа 8 - ЛУЧШИЙ СЛУЧАЙ
Небходимо выполнить 1 сравнение:
8 и 8 - ключ 8 найден. Поиск завершен.

3. Поиск ключа 2 - СРЕДНИЙ СЛУЧАЙ
Необходимо выполнить 4 сравнения:
2 и 8 - ключ 2 не найден, продолжаем поиск
2 и 4 - ключ 2 не найден, продолжаем поиск
2 и 7 - ключ 2 не найден, продолжаем поиск
2 и 2 - ключ 2 найден. Поиск завершен.

Асимптотические классы эффективности алгоритмов

Константный $O(1)$ - устойчивое время работы, которое не зависит от размера задачи. При увеличении размера входных данных время выполнения алгоритма либо изменяется мало либо не изменяется вовсе.

Логарифмический $O(\log n)$ - удвоение размера задачи увеличивает время работы на постоянную величину. Присуще алгоритмам, которые сводят большую задачу к набору меньших задач, уменьшая на каждом шаге размер задачи на некоторый постоянный фактор.

Линейный $O(n)$ - удвоение размера задачи удвоит и необходимое время. Каждый входной элемент подвергается небольшой обработке.

Линеаритмичный $O(n \log n)$ - удвоение размера задачи увеличит необходимое время чуть более чем вдвое. Алгоритмы декомпозиции.

Квадратичный $O(n^2)$ - удвоение размера задачи увеличивает время выполнения в 4 раза. Полезен для практического использования небольших задач.

Кубический $O(n^3)$ - удвоение размера задачи увеличивает время выполнения в 8 раз. Сложные алгоритмы линейной алгебры.

Экспоненциальный $O(c^n)$ - увеличение размера задачи на 1 приводит к с-кратному увеличению времени выполнения. Удвоение размера входных данных увеличивает временя выполнения в квадрат. Алгоритмы, выполняющие обработку всех подмножеств некоторого множества, состоящего из n элементов.

Факториальный $O(n!)$ - алгоритмы, выполняющие обработку всех перестановок некоторого множества, состоящего из n элементов.

Эмпирический анализ эффективности алгоритмов

План проведения эмпирического анализа

1. Определение цели.
2. Определение единиц измерения (количество операций или время работы).
3. Определение характеристик входных данных (их диапазон, размер).
4. Создание программы, реализующей алгоритм.
5. Генерация образца входных данных.
6. Выполнение алгоритма над образцом входных данных и запись наблюдаемых результатов.
7. Анализ полученных данных.

Измерение эффективности алгоритма можно проводить двумя способами:

1. Вставка в программу, реализующую алгоритм, счетчика, который будет подсчитывать количество выполнений алгоритмом базовых операций. При этом, счетчиков может быть несколько и они могут находиться в разных местах программы.

2. Определение времени работы программы, реализующей исследуемый алгоритм. Время работы фрагмента кода определяется путем запроса системного времени непосредственно перед началом выполнения фрагмента и сразу после его завершения и вычислением разности их значений.

$$t = t_{end} - t_{start}$$

t - время выполнения алгоритма;

t_{start} - время перед началом выполнения алгоритма;

t_{end} - время после выполнения алгоритма.

При этом системное время возвращается в «тиках». Поэтому полученное время t нужно разделить на константу, определяющую количество «тиков» в единице времени.

Пример проведения эмпирического анализа для C/C++.

1. Подключить библиотеку

#include <time.h>

2. Объявить переменные, которые будут хранить системное время:

clock_t start, end;

3. Объявить и инициализировать переменную, которая будет хранить время выполнения алгоритма:

float t = 0;

4. Перед началом алгоритма засечь системное время с помощью функции clock():

start = clock();

5. После выполнения алгоритма засечь системное время:

end = clock();

6. Вычислить время выполнения алгоритма и перевести в секунды, разделив на системную переменную:

t = (end - start)/CLK_TCK;

Замечания:

1. Т.к. системное время не очень точное, то при повторных запусках одной и той же программы с одними и теми же входными данными могут получаться различные результаты с незначительными расхождениями.

$\Rightarrow$ Многократный запуск программы и усреднение результатов.

2. Высокая скорость работы компьютера приводит к тому, что время выполнения алгоритма будет слишком малым и будет давать нулевые значения.

$\Rightarrow$ Запуск программы в цикле много раз, поделить время на количество итераций цикла.

$\Rightarrow$ Умножить время выполнения на 10^n при выводе значения на экран.

3. Многозадачные ОС (например, Unix) во время t могут включать и время, затраченное на работу с другими программами.

$\Rightarrow$ Запрашивать у ОС «пользовательское время» - время, затраченное на выполнение конкретного.

Выбор входных данных:

а) размер входных данных – начинать с данных относительно небольшого размера и постепенно увеличивать;

б) диапазон входных данных - выбирать не слишком малые и не слишком большие.

При использовании нескольких экземпляров входных данных одного размера, значения времени усредняют.

Эмпирический анализ эффективности требует генерации случайных чисел. В C++ для этого используют генератор случайных чисел:

Библиотека ***include <stdlib.h>***

int i = rand( ); - генерация любого положительного числа от 0 до RAND_MAX (одинаковое число)

srand (time(NULL)); i = rand ( ); - каждый запуск программы выдает разное число

начальное_значение + rand( ) % конечное_значение; - генерация числа из заданного диапазона

Анализ эмпирических данных

Эмпирические данные для анализа представляют в таблице или графически.

Пример: Есть информация о количестве входных данных и соответствующем времени работы данного алгоритма, представленная в виде таблицы.

N	**10**	**20**	**40**	**80**
$t(n)$	5	11	23	45
$\frac{t(2n)}{t(n)}$	11/5 = 2.2	23/11 = 2.1	45/23 = 1.96	

Т.е. при удвоении размера входных данных, время выполнения алгоритма растет в 2 раза. $\Rightarrow$ **Линейная зависимость**.

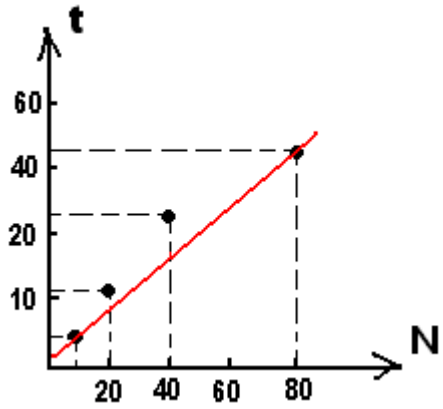


Рисунок 1 – График зависимости времени выполнения алгоритма от размера входных данных

Математический анализ эффективности алгоритмов

Математический анализ производительности алгоритмов заключается в выполнении следующих шагов:

1. Выбрать параметры, по которым будет оцениваться размер входных данных алгоритма (например, размер массива, строки).
2. Определить основную операцию алгоритма (базовую операцию).
3. Проверить, зависит ли число базовых операций только от размера входных данных.

Если число базовых операций зависит только от размера входных данных, то эффективность алгоритма можно оценить лишь для среднего случая.

Если число базовых операций зависит и от других факторов, то необходимо рассмотреть эффективность алгоритма для наихудшего, среднего и наилучшего случаев.

4. Записать сумму, выражающую количество выполняемых основных операций алгоритма.

5. Используя стандартные формулы и правила суммирования, упростить полученную формулу для количества основных операций алгоритма. Если это невозможно, то определить лишь порядок роста.

Правила суммирования	***Формулы суммирования***
$1.\sum_{i=L}^{u} ca_i = c\sum_{i=L}^{u} a_i;$	$1.\sum_{i=L}^{u} 1 = u - L + 1;$
$2.\sum_{i=L}^{u} (a_i \pm b_i) = \sum_{i=L}^{u} a_i \pm \sum_{i=L}^{u} b_i;$	$2.\sum_{i=1}^{u} i^k \approx \frac{1}{k+1} n^{k+1};$
$3.\sum_{i=L}^{u} (ca_i \pm b_i) = c\sum_{i=L}^{u} a_i \pm \sum_{i=L}^{u} b_i;$	$3.\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}, при \quad a \neq 1.$
$4.\sum_{i=L}^{u} (a_i - a_{i-1}) = a_u - a_{L-1};$	

Пример 1. Провести математический анализ эффективности алгоритма, находящего сумму элементов одномерного массива.

1. Параметр, по которому оценивается размер входных данных: массив размера n.

2. Основная операция алгоритма: сложение элементов массива.

3. Зависимость числа выполняемых операций: только от количества входных данных.

4. Сумма основных операций алгоритма: $C(n) = \sum_{i=1}^{n-1} 1$.

За один цикл в алгоритме выполняется одна операция сложения. Этот процесс повторяется для каждого значения переменной цикла, которое изменяется от 1 до n.

$\Rightarrow$ Линейная зависимость.

Пример 2. Провести математический анализ эффективности алгоритма, который заполняет главную диагональ квадратной матрицы путем

перестановки элементов: максимальный элемент матрицы ставится в позицию (1, 1), следующий – в позицию (2, 2) и т.д.

1. Параметр, по которому оценивается размер входных данных: матрица размером n x n.

2. Основная операция алгоритма: перестановка элементов матрицы.

3. Зависимость числа выполняемых операций: от количества входных данных и их расположения.

4. Сумма основных операций алгоритма:

$$C(n) = \sum_{a=0}^{n-1}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{a=0}^{n-1}\sum_{i=0}^{n-1} n = \sum_{a=0}^{n-1} n^2 = n^3 .$$

$\Rightarrow$ Кубическая зависимость.

Пример 3. Провести математический анализ эффективности алгоритма, который определяет максимум среди сумм элементов диагоналей квадратной матрицы, параллельных главной диагонали матрицы.

1. Параметр, по которому оценивается размер входных данных: матрица размером n x n.

2. Основная операция алгоритма: сложение и сравнение элементов матрицы.

3. Зависимость числа выполняемых операций: только от количества входных данных.

4. Сумма основных операций алгоритма:

$$C(n) = \sum_{i=1}^{n}\sum_{j=1}^{n-1} 1 + \sum_{k=1}^{n-1} 1 = n(n-1) + n - 1 = n^2 - n + n - 1 = n^2 - 1.$$

Для диагоналей, включая главную и параллельные, найти суммы всех элементов (n элементов, n-1 операций сложения). Затем сравнение полученных сумм (n элементов, n-1 операций сложения).

$\Rightarrow$ Квадратичная зависимость.

Задание

1. Разработать алгоритм и программу по индивидуальному заданию.
2. Провести математический анализ эффективности алгоритма и определить класс эффективности.
3. Провести эмпирический анализ производительности алгоритма и определить класс эффективности.

Индивидуальное задание

1. Преобразовать одномерный массив, состоящий из n вещественных элементов, таким образом, чтобы сначала располагались все элементы,

отличающиеся от максимального не более чем на 20%, а потом – все остальные.

2. Дана целочисленная квадратная матрица. Определить произведение элементов в тех строках, которые не содержат отрицательных элементов.
3. В одномерном массиве, состоящем из n вещественных элементов, вычислить произведение элементов массива, расположенных между максимальным и минимальным элементами.
4. Дана целочисленная квадратная матрица. Определить максимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.
5. Преобразовать одномерный массив, состоящий из n целых элементов, таким образом, чтобы сначала располагались все положительные элементы, а потом – все отрицательные (элементы, равные 0, считать положительными).
6. Дана целочисленная квадратная матрица. Найти сумму модулей элементов, расположенных выше главной диагонали.
7. Сжать одномерный массив, состоящий из n вещественных элементов, удалив из него все элементы, модуль которых не превышает X. Освободившиеся в конце массива элементы заполнить нулями.
8. Дана целочисленная квадратная матрица. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.
9. Преобразовать одномерный массив, состоящий из n вещественных элементов, таким образом, чтобы сначала располагались все элементы, равные нулю, а потом – все остальные.
10. Путем перестановки элементов квадратной целочисленной добиться того, чтобы ее максимальный элемент находился в левом верхнем углу (в позиции (1,1)), следующий по величине – в позиции (2,2), следующий по величине – в позиции (3,3) и т.д., заполнив таким образом всю диагональ.
11. Преобразовать одномерный массив, состоящий из n целых элементов, таким образом, чтобы в первой его половине располагались элементы, стоявшие в нечетных позициях, а во второй половине – элементы, стоявшие в четных позициях.
12. Дана целочисленная квадратная матрица. Найти номер первой из строк, не содержащих ни одного положительного элемента.
13. Преобразовать одномерный массив, состоящий из n вещественных элементов, таким образом, чтобы сначала располагались все элементы, модуль которых не превышает X, а потом – все остальные.
14. Дана целочисленная квадратная матрица. Определить количество строк, содержащих хотя бы один нулевой элемент.

15. В одномерном массиве, состоящем из n вещественных элементов, вычислить сумму модулей элементов массива, расположенных после первого отрицательного элемента.
16. Дана целочисленная квадратная матрица. Определить номер столбца, в котором находится самая длинная серия одинаковых элементов.
17. Преобразовать одномерный массив, состоящий из n вещественных элементов таким образом, чтобы сначала располагались все элементы, целая часть которых лежит в интервале [a, b], а потом – все остальные.
18. Дана целочисленная квадратная матрица. Определить количество отрицательных элементов в тех строках, которые содержат хотя бы один нулевой элемент.
19. В одномерном массиве, состоящем из n вещественных элементов, вычислить количество элементов массива, больших X.
20. В одномерном массиве, состоящем из n целых элементов, вычислить сумму элементов массива, расположенных после последнего элемента, равного нулю.
21. В одномерном массиве, состоящем из n вещественных элементов, вычислить произведение отрицательных элементов массива.
22. В одномерном массиве, состоящем из n вещественных элементов, вычислить сумму положительных элементов массива, расположенных до максимального элемента.
23. В одномерном массиве, состоящем из n вещественных элементов, вычислить сумму отрицательных элементов массива.
24. В одномерном массиве, состоящем из n вещественных элементов, вычислить количество отрицательных элементов массива.
25. В одномерном массиве, состоящем из n вещественных элементов, вычислить сумму целых частей элементов массива, расположенных после последнего отрицательного элемента.
26. В одномерном массиве, состоящем из n вещественных элементов, вычислить сумму модулей элементов массива, расположенных после минимального по модулю элемента.

Контрольные вопросы

1. В чем заключается математический анализ производительности алгоритмов?
2. В чем заключается эмпирический анализ производительности алгоритмов?
3. Что понимается под эффективностью в худшем, среднем и лучшем случае?
4. Какие асимптотические классы эффективности алгоритмов Вам известны?

Задание для достаточного уровня

Написать программу, реализующую алгоритм нахождения суммы элементов квадратной целочисленной матрицы. Провести эмпирический и математический анализ эффективности данного алгоритма.

```
#include <iostream>
#include <ctime>
#include <time.h>
using namespace std;

int main()
{
   srand(time(NULL));
   int n = 0, sum = 0;
   clock_t start, end;
   float t = 0;

   cin>>n; //Считываем с клавиатуры размер массива
   int **a = new int* [n]; //Создаем массив указателей

   for (int i = 0; i<n; i++)   {      a[i] = new int [n]; }//Создаем элементы

   //Генерируем случайным образом элементы матрицы и выводим на экран
   for(int i=0; i<n; i++)
   {
      for(int j=0; j<n; j++)
      {  a[i][j] = rand()%99;      cout<<a[i][j]<<" ";   }
      cout<<endl;
   }

   //Сложение элементов матрицы
   start = clock();
   for(int i=0; i<n; i++)
   {
      for(int j=0; j<n; j++)   {  sum += a[i][j];   }
   }
   end = clock();
   t = (end - start)/CLK_TCK;

   //Вывод результата на экран
cout << "sum = " << sum <<endl;
cout << "time = " << t <<endl;

   //Удаление массива
 for(int i=0; i<n; i++)  {  delete[]a[i];   } //Удаляем каждый элемент

 delete [] a;    //Удаляем массив
 return 0;
}
```

Лабораторная работа №2

АЛГОРИТМЫ СОРТИРОВКИ ДАННЫХ

Цель работы: изучение алгоритмов сортировки и проведение их эмпирического анализа производительности.

Теоретические сведения

Сортировка – упорядочивание данных в определенном порядке.

Цель сортировки – переупорядочивание элементов таким образом, чтобы их ключи следовали в соответствии с четко определенными правилами (цифровой или алфавитный порядок).

Будем рассматривать методы сортировки файлов элементов, обладающих ключами.

Ключ – это небольшая часть элементов, которая используется для управления сортировкой.

Основной характеристикой алгоритма сортировки является время, затрачиваемое на его выполнение.

В общем случае, время выполнения алгоритма сортировки пропорционально:

- количеству операций сравнения, выполняемых этим алгоритмом;
- количеству перемещений или перемен местами элементов;
- различий соответствующих постоянных коэффициентов пропорциональности в зависимости от числа выполняемых операций во внутренних циклах.

Второй по важности характеристикой алгоритмов сортировки является дополнительный объем оперативной памяти, используемый данным алгоритмом.

По данной характеристике методы сортировки делятся на три категории:

- те, которые выполняют сортировку на месте и не нуждаются в дополнительной памяти;
- те, которые получают доступ к данным, используя для этого указатели или индексы массивов, в связи с чем необходима дополнительная память для размещения N указателей или индексов;
- те, которые требуют дополнительную память для размещения еще одной копии массива, подвергаемого сортировке.

Часто применяются методы сортировки элементов с несколькими ключами, поэтому важно знать, обладает ли выбранный метод сортировки свойством устойчивости.

Метод сортировки устойчив, если он сохраняет относительный порядок размещения элементов в файле, который содержит дублированные ключи.

Сортировка выбором – работает по принципу выбора наименьшего элемента из числа неотсортированных.

Алгоритм:

- Отыскать наименьший элемент массива. Поменять его местами с элементом, стоящим первым в сортируемом массиве.
- Найти второй наименьший элемент в массиве. Поменять его местами со вторым элементом, стоящим в исходном массиве.
- Этот процесс продолжается до тех пор, пока весь массив не будет отсортирован.

Пример.

16 7 30 18 2 64 9 39

2 7 30 18 16 64 9 39

2 7 30 18 16 64 9 39

2 7 9 18 16 64 30 39

2 7 9 16 18 64 30 39

2 7 9 16 18 64 30 39

2 7 9 16 18 30 64 39

2 7 9 16 18 30 39 64

2 7 9 16 18 64 30 39

Реализация

```
template <class Item>
void selection (Item a[], int l, int r)
{
   for(int i=l; i<r; i++){
      int min=i;
      for(int j = i+1; j <=r; j++)
         if (a[j] < a[min])min=j;
     exch(a[i],a[min]);
   }
}
void exch (Item &A, Item &B)
{
   Item t=A;
   A=B;
   B=t;
}
```

Недостаток: время выполнения мало зависит от степени упорядоченности исходного массива.

Применение: когда записи файла огромны, а ключи занимают незначительное пространство.

Сортировка выбором производит в среднем $\approx \frac{N^2}{2}$ операций сравнения и N операций обмена элементов местами.

Сортировка вставками – заключается в том, что отдельно анализируется каждый конкретный элемент, который затем помещается в надлежащее место среди других, уже отсортированных элементов.

При компьютерной реализации нужно освобождать место для вставляемого элемента путем смещения больших элементов на одну позицию вправо, после чего на освободившееся место помещается вставленный элемент.

Элементы, находящиеся слева от текущего индекса, отсортированы в соответствующем порядке, но они еще не занимают свои окончательные позиции, так как могут быть передвинуты для освобождения места под элементы, которые находятся позже.

Массив будет полностью отсортирован, когда индекс достигнет правой границы массива.

Пример.

16	**7**	**30**	**18**	**2**	**64**	**9**	**39**
7	**16**	**30**	**18**	**2**	**64**	**9**	**39**
7	**16**	**30**	**18**	**2**	**64**	**9**	**39**
7	**16**	**18**	**30**	**2**	**64**	**9**	**39**
2	**7**	**16**	**18**	**30**	**64**	**9**	**39**
2	**7**	**16**	**18**	**30**	**64**	**9**	**39**
2	**7**	**9**	**16**	**18**	**30**	**64**	**39**
2	**7**	**9**	**16**	**18**	**30**	**39**	**64**

Реализация

```
template <class Item>
void insertion(Item a[], int l, int r)
{
        int i;
        for(i=r; i<l; i--) compexch(a[i-1],a[i]);
        for(i=l+2; i<=r; i++)
        {
                int j = i;
                Item v = a[i];
                while(v < a[j-1]) {a[j] = a[j-1]; j--;}
                a[j] = v;
        }
}
void compexch (Item &A, Item &B) {  if(B < A) ecxh(A,B); }
```

Сортировка вставками производит в среднем $\approx \frac{N^2}{4}$ операций сравнения и $\approx \frac{N^2}{4}$ операций перемещения и в 2 раза больше операций в худшем случае.

Сортировка обменным (двухпутевым) слиянием – заключается в объединении двух отсортированных файлов в один файл большего размера.

Имея два упорядоченных входных файла, сортировка обменным слиянием позволяет их объединить в один упорядоченный выходной, отслеживая наименьший элемент в каждом файле и входя в цикл, в котором меньший из двух элементов, наименьших в своих файлах, переносится в выходной файл. Процесс продолжается до тех пор, пока оба входных файла не будут исчерпаны.

Время выполнения линейно зависит от количества элементов в выходном файле, если на каждую операцию следующего наименьшего элемента в файле уходит одно и то же время.

Пример.

Даны два отсортированных массива:

A = [3 8 15 ~~44~~]; B = [9 10 17 28 30].

Слить в один отсортированный массив C:

A[1] = 3 B[1] = 9 -> 3
A[2] = 8 B[1] = 9 -> 8
A[3] = 15 B[1] = 9 -> 9
A[3] = 15 B[2] = 10 -> 10
A[3] = 15 B[3] = 17 -> 15
A[4] = ~~44~~ B[3] = 17 -> 17
A[4] = ~~44~~ B[4] = 28 -> 28
A[4] = ~~44~~ B[5] = 30 -> 30
A[4] = ~~44~~ -> ~~44~~

C = [3 8 9 10 15 17 28 30 ~~44~~]

Реализация

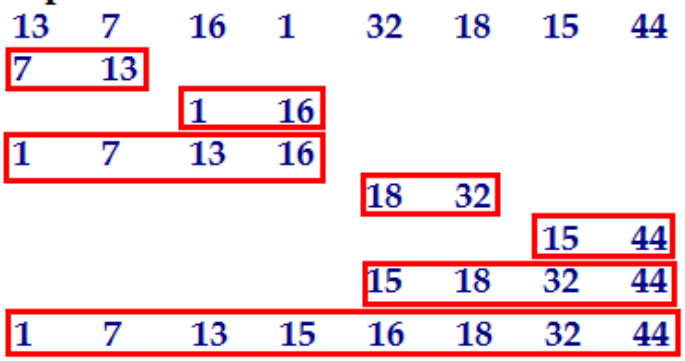
```
template <class Item>
void mergeAB (Item c[], Item a[], int N, Item b[], int M)
{
   for(int i=0, j=0, k=0; k<N+M; k++)
   {
      if (j==N)   {c[k]=b[j++];continue;}
      if (j==M)   {c[k]=a[i++];continue;}
      c[k] = (a[i]<b[j]) ? a[i++] : b[j++];
   }
}
```

Нисходящая сортировка слиянием – аналогична принципу управления сверху вниз, в рамках которого руководитель организует работу таким образом, что получив большую задачу он разбивает ее на подзадачи, которые должны независимо решать подчиненные. Затем решения объединяются.

Сортировка слиянием играет важную роль, так как является простой и оптимальной (время ее выполнения пропорционально $N \log N$). Допускает возможность реализации, обладающей устойчивостью.

Чтобы отсортировать заданный файл, он делится на 2 части, выполняется рекурсивная сортировка обеих частей, затем производится их слияние.

Пример.

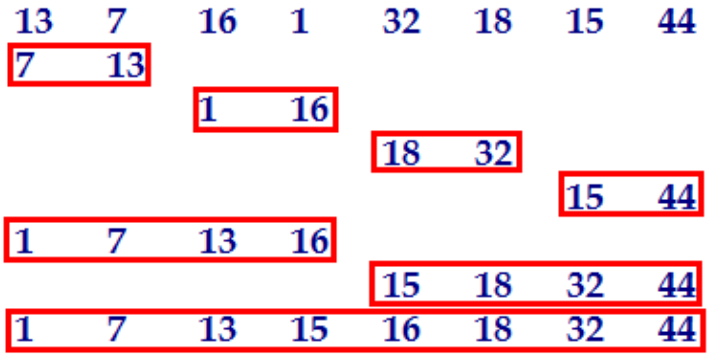


Реализация

```
template <class Item>
void mergesort (Item a[], int l, int r)
{
        if(r <= l) return;
        int m = (r + l)/2;
        mergesort (a, l,  m);
        mergesort (a, m+1, r);
        merge (a, l, m, r);
}
void merge (Item a[], int l, int m, int r)
{
        int i,j;
        static Item aux[maxN];
        for(i = m+1; i<l; i--) aux[i-1] = a[i-1];
        for(j = m; j<r; j++) aux[r+m-j] = a[j+1];
        for(int k=l; k<=r; k++)
                if(aux[j] < aux[i]) a[k] = aux[j--];
                else a[k] = aux[i++];
}
```

Восходящая сортировка слиянием – состоит из последовательности проходов по всему файлу с выполнением слияния, при этом m на каждом шаге удваивается.

Пример.



Реализация

```
inline int min(int A, int B) { return (A < B) ? A : B; }
template <class Item>
void mergesortBU (Item a[], int l, int r)
{
        for(int m=1; m<=r-1; m=m+m)
                for(int i=1; i<=r-m; i+=m+m)
                        merge(a,i,i+m-1, min(i+m+m-1,r));
}
```

Быстрая сортировка – наиболее популярный алгоритм сортировки, так как его легко реализовывать, хорошо работает на различных видах входных данных.

Алгоритм быстрой сортировки принадлежит к категории обменных сортировок и требует небольшого вспомогательного стека.

На выполнение сортировки N элементов в среднем затрачивается время, пропорциональное $N \log N$.

Недостаток: неустойчив, для его выполнения в худшем случае требуется N^2 операций. Простая ошибка в реализации может пройти незамеченной и вызвать ошибки в работе алгоритма на некоторых видах файлов.

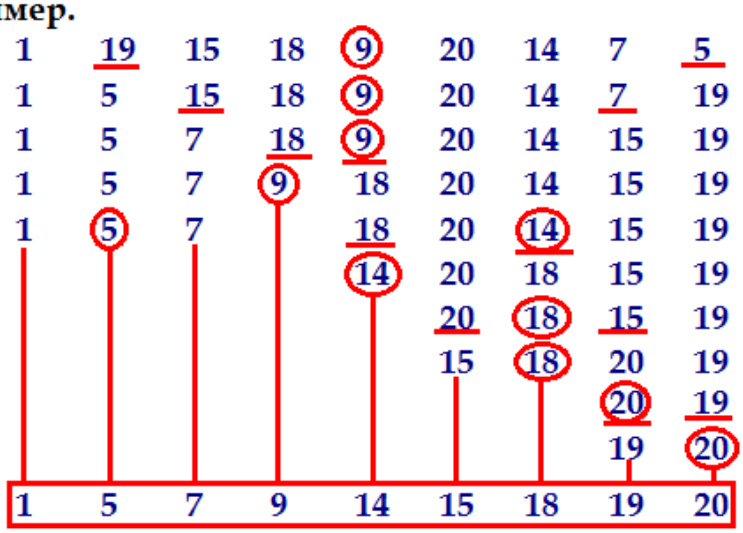
Быстрый алгоритм сортировки функционирует по принципу «разделяй и властвуй». Он делит массив на две части, затем сортирует их независимо друг от друга.

Алгоритм:

1. Произвольно выбрать разделяющий элемент a[r].
2. Осуществить просмотр с левого конца массива пока не будет найден элемент больше разделяющего.
3. Осуществить просмотр с правого конца массива пока не будет найден элемент меньше разделяющего.
4. Элементы, на которых был прерван просмотр, меняются местами.

Такая процедура выполняется до тех пор, пока слева от левого указателя нет элементов больше разделяющего, а справа – меньше разделяющего. Затем рекурсивно применяем эту процедуру к частям массива.

Пример.



Реализация

```
template <class Item>
void quicksort (Item a[], int l, int r)
{       if (r <= l) return;
        int i = partition (a,l,r);
        quicksort (a,l,i-1);
        quicksort (a,i+1,r);
}
int partition (Item a[], int l, int r)
{       int i = l-1, j = r;
        Item v = a[r];
        for (; ;) {
                while ( a[++i] < v );
                while ( v < a[--j] ) if (j == l) break;
                if(i >= j) break;
                exch(a[i], a[j]);   }
        exch( a[i], a[r] );
        return i;
}
```

Пирамидальная сортировка – неразрывно связана с понятием сортирующего дерева.

Сортирующее дерево – структура данных, в которой записи хранятся в виде массива таким образом, что каждый ключ обязательно принимает значение большее, чем значение двух других ключей, занимающих относительно него строго определенные положения.

Дерево называется пирамидально упорядоченным, если ключ в каждом его узле больше или равен ключам всех потомков этого узла. Ни один из узлов пирамидально упорядоченного дерева не может иметь ключа, большего, чем ключ корня дерева.

Производительность пирамидальной сортировки $O(N \log N)$.

Алгоритм:

1. Строим бинарное дерево: сверху вниз, слева направо, заполняя каждый уровень дерева.

2. Перемещаем элементы дерева так, чтобы дерево было пирамидально упорядочено: потомки должны быть меньше или равны своему родителю.

Для построения пирамидально упорядоченного дерева, движемся справа налево снизу вверх, меняя местами потомка с родителем (в случае, если он не удовлетворяет условию пирамидально упорядоченного дерева).

3. В вершине пирамидально упорядоченного дерева находится максимальный элемент. Меняем элемент, находящийся в вершине с элементом, находящимся в нижней правой позиции.

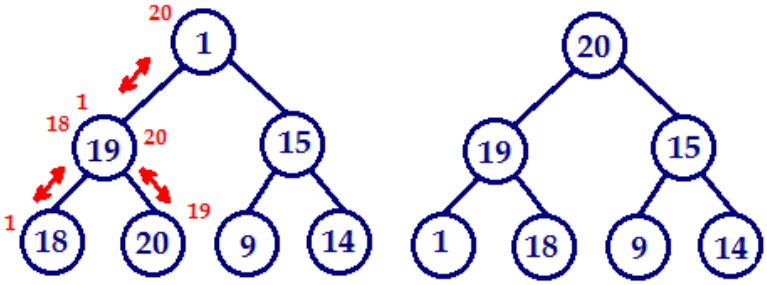
4. Вершина, которая была помещена в нижнюю правую позицию дерева, исключается из рассмотрения. Элемент, который был помещен в вершину дерева просеивается вниз по дереву так, чтобы дерево снова стало пирамидально упорядоченным.

5. Алгоритм завершается, когда больше нет элементов, которые необходимо помещать в вершину дерева. В этом случае все элементы записываем сверху вниз слева направо. Получим отсортированный массив.

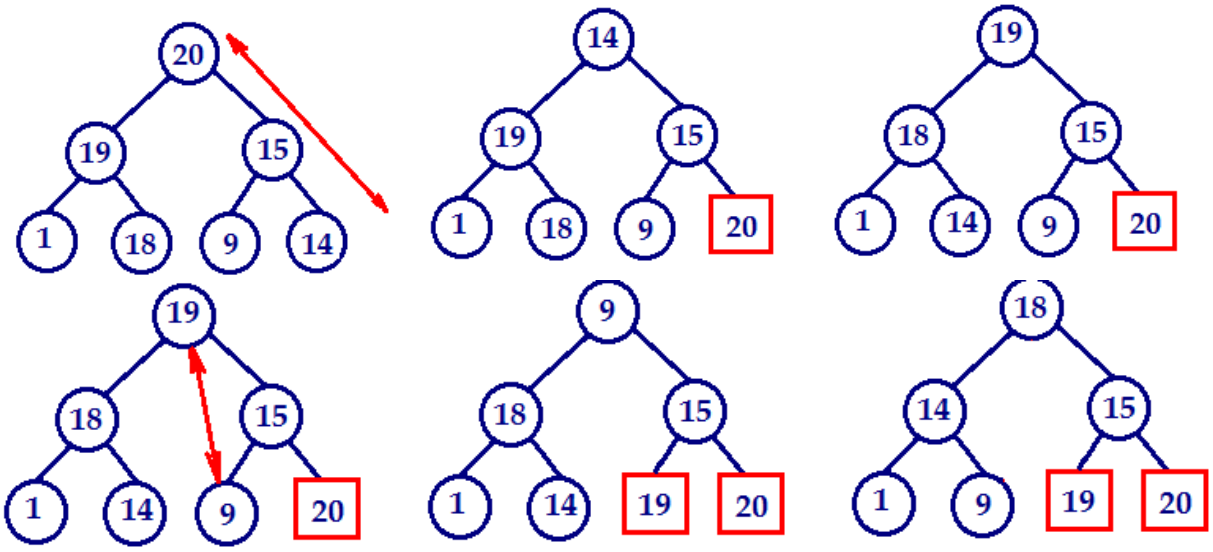
Пример.

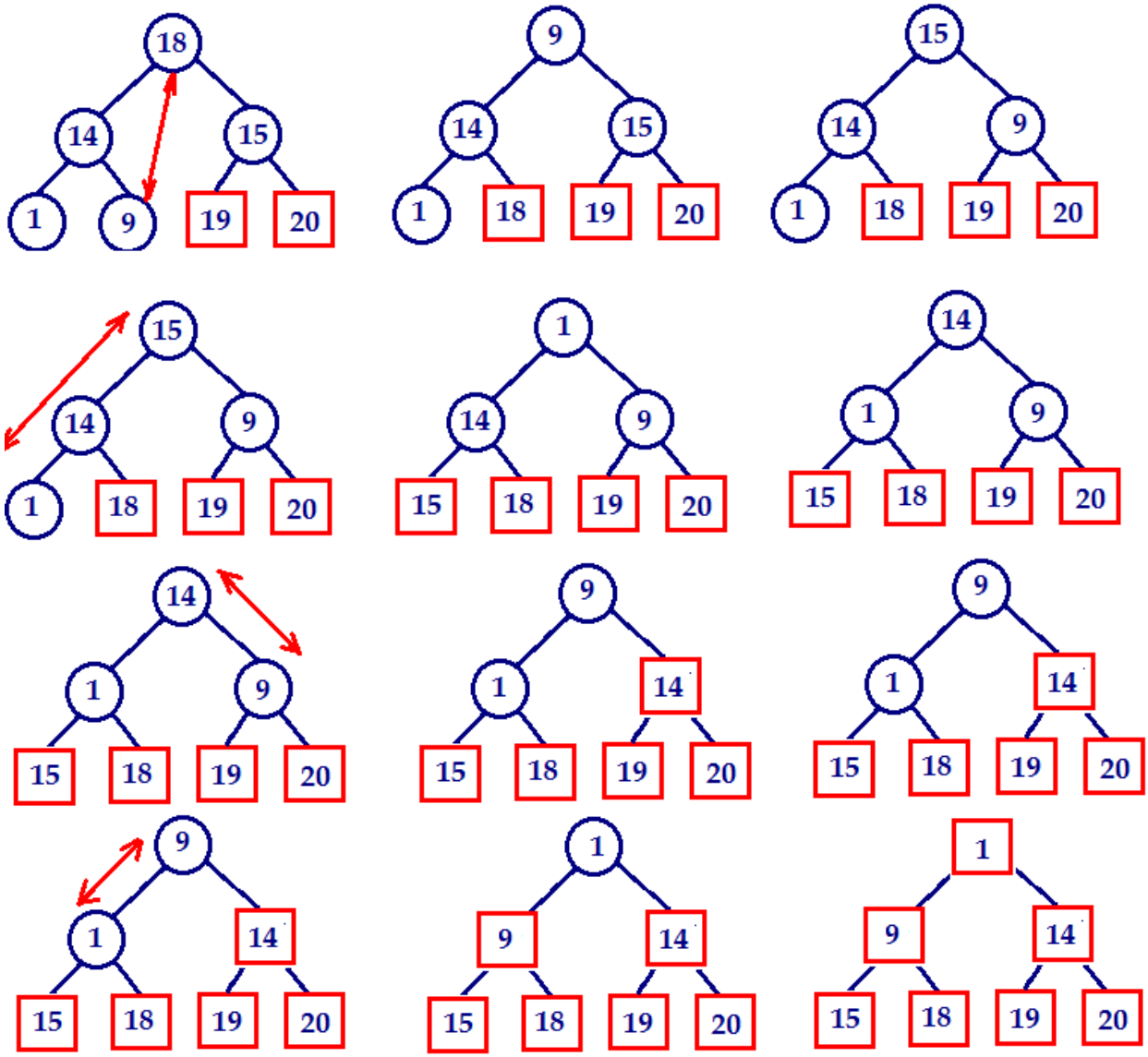
1 19 15 18 20 9 14

1. Строим пирамидально упорядоченное дерево.



2. Пирамидальная сортировка





Результат: 1 9 14 15 18 19 20

Реализация

```
template <class Item>
void heapsort (Item a[], int l, int r)
{
        int k, N = r-l+1;
        Item *pq = a+l-1;
        for(k=N/2; k >= l; k--)  fixDown(pq, k, N);
        while (N > 1)
        {       exch(pq[l], pq[N]);
                fixDown(pq, l, --N);
        }
}
void fixDown (Item a[], int k, int N)
{
        while(2*k <= N)
```

```
        {
                int j = 2*k;
                if (j < N && a[j] < a[j+1]) j++;
                if (!(a[k] < a[j])) break;
                exch(a[k], a[j]);
                k = j;
        }
}
void fixUp (Item a[], int k)
{       while (k > 1 && a[k/2] < a[k])
        {       exch(a[k], a[k/2]);
                k = k/2;
        }
}
```

Поразрядная сортировка – построена на обработке чисел по одной порции за раз.

Алгоритмы поразрядной сортировки рассматривают ключи как числа, представленные в системе счисления с основанием R при различных значениях R, и работают с отдельными цифрами чисел.

В зависимости от контекста, ключом в поразрядной сортировке может быть слово или строка.

В качестве числа разрядов в слове используются константы:

```
const int bitword = 32;
const int bitsbyte = 8;
const int bytesword = bitsword/bitsbyte;
const int R = 1 << bitsbyte;
```

Поразрядная сортировка MSD – поразрядная сортировка сначала по старшей цифре. Алгоритм, который анализирует значение цифр в ключах в направлении слева направо, при этом первыми обрабатываются наиболее значащие цифры.

Поразрядная сортировка MSD выполняется за счет разделения сортируемого файла в соответствии со старшими цифрами ключей, после чего тот же метод применяется к подфайлам в режиме рекурсии.

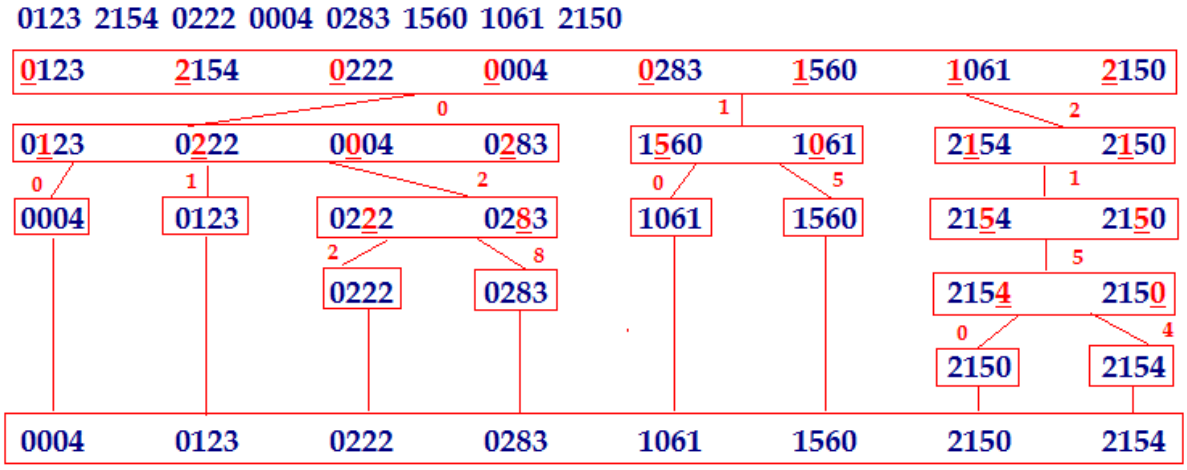
Производительность MSD-сортировки $O(N \log_R N)$. При больших R величина $\log_R N$ становится малой и производительность становится линейной $O(N)$.

Алгоритм:

1. Исходный массив делится на R частей («корзин»), где R – основание, выбранное для представления ключей как чисел. В первую корзину попадают ключи, у которых старшая цифра в позиции d = 0 имеет значение 0. Во вторую корзину попадают ключи, у которых старшая цифра в позиции d = 0 имеет значение 1 и т.д.

2. Ключи, попавшие в разные корзины, подвергаются рекурсивному разделению по следующей цифре с позицией d = 1. Разделение выполняется для корзин, у которых более одного ключа.

Пример.



Реализация

```
#define bin(A) l+count[A]
template <class Item>
void radixMSD (Item a[], int l, int r, int d)
{
        int i, j, count[R+1];
        static Item aux[maxN];
        if(d > bytesword) return;
        if(r-l <= M) {insertion (a,l,r); return ;}
        for (j=0; j<R; j++) count[j] = 0;
        for(i=l; i<=r; i++) count[digit(a[i],d)+1]++;
        for(j=l; j<R; j++) count[j] += count[j-1];
        for(i=l; i<=r; i++) aux[l+count[digit(a[i],d)]++] = a[i];
        for(i=l; i<=r; i++) a[i] = aux[i];
        radixMSD(a, l, bin(0)-1, d+1);
        for(j=0; j<R-1; j++) radixMSD(a, bin(j), bin(j+1)-1, d+1);
}
inline int digit(char *A, int B) { return A[B]; }
```

Поразрядная сортировка LSD – поразрядная сортировка сначала по младшей цифре. Алгоритм, который анализирует цифры ключей в направлении справа налево, работая сначала с наименее значащей цифрой.

Поразрядная сортировка LSD – выполняет разделение ключей по корзинам, выбирая цифры справа налево, от младших цифр, находящихся в позиции d = k-1 к старшей цифре в позиции d = 0.

Алгоритм LSD не рекурсивный. При сортировке по более старшим разрядам порядок ключей, полученный на предыдущих шагах не поддерживается. После очередного шага все сформированные корзины вновь объединяются в пределах всего сортируемого массива в одну группу и на следующем шаге выполняется разделение по следующей цифре всего

массива. После сортировки по последней, старшей цифре, массив упорядочен по ключам.

Производительность LSD-сортировки $O(n_x w)$, где w – число разрядов в сортируемых ключах. LSD-сортировка имеет линейную зависимость от объема сортируемых данных.

Пример.

0123 2154 0222 0004 0283 1560 1061 2150

1 шаг:

распределение по корзинам:

[1560 2150] [1061] [0222] [0123 0283] [2154 0004]

объединение корзин:

1560 2150 1061 0222 0123 0283 2154 0004

2 шаг:

распределение по корзинам:

[0004] [0222 0123] [2150 2154] [1560 1061] [0283]

объединение корзин:

0004 0222 0123 2150 2154 1560 1061 0283

3 шаг:

распределение по корзинам:

[0004 1061] [0123 2150 2154] [0222 0283] [1560]

объединение корзин:

0004 1061 0123 2150 2154 0222 0283 1560

4 шаг:

распределение по корзинам:

[0004 0123 0222 0283] [1061 1560] [2150 2154]

объединение корзин:

0004 0123 0222 0283 1061 1560 2150 2154

Реализация

```
template <class Item>
void radixLSD (Item a[], int l, int r)
{
        static Item aux[maxN];
        for(int d = bytesword-1; d>=0; d--)
        {
                int i, j, count[R+1];
                for(j=0; j<R; j++) count[j]=0;
                for(i=1; i<=r; i++) count[digit(a[i],d)+1]++;
                for(j=1; j<R; j++) count[j] += count[j-1];
                for(i=l; i<=r; i++) aux[count[digit(a[i],d)]++] = a[i];
                for(i=l; i<=r; i++) a[i] = aux[i];
        }
}
```

Контрольные вопросы

1. Что представляет собой сортировка? Ее основная цель.
2. Что представляет собой ключ?
3. Основные характеристики алгоритмов сортировки.
4. Что понимают под устойчивостью алгоритмов сортировки?
5. Назвать известные методы сортировок, их алгоритмы, недостатки и преимущества, производительность.

Задание

1. Разработать две клиентские программы, реализующие алгоритмы сортировки, указанные в варианте.

2. На основе эмпирического анализа определить классы эффективности, к которым относятся указанные алгоритмы. При этом рассмотреть «средний» и «худший» случай.

Расшифровка номера сортировок в задании:

1- сортировка выбором;
2- сортировка вставками;
3- сортировка обменным (двухпутевым) слиянием;
4- нисходящая сортировка слиянием;
5- восходящая сортировка слиянием;
6- быстрая сортировка;
7- пирамидальная сортировка;
8- поразрядная сортировка MSD;
9- поразрядная сортировка LSD.

№ студента в списке	КН-1		КН-2		КН-с	
	№1	№2	№1	№2	№1	№2
1	1	6	5	9	3	7
2	2	7	4	8	5	8
3	3	8	3	7	1	6
4	4	9	2	6	4	9
5	5	6	1	9	2	7
6	1	7	5	8	3	8
7	2	8	4	7	5	6
8	3	9	3	6	1	9
9	4	6	2	9	4	7
10	5	7	1	8	2	8
11	1	8	5	7	3	6
12	2	9	4	6	5	9
13	3	6	3	9	1	7

14	4	7	2	8	4	8
15	5	8	1	7	2	6
16	1	9	5	6	3	9
17	2	6	4	9	5	7
18	3	7	3	8	1	8
19	4	8	2	7	4	6
20	5	9	1	6	2	9

Лабораторная работа №3

АЛГОРИТМЫ ПОИСКА

Цель работы: изучение структур данных и алгоритмов, реализующих поиск.

Теоретические сведения

Поиск – операция, позволяющая получить конкретный фрагмент информации из больших томов ранее сохраненных данных.

Цель поиска – отыскание элементов с ключами, которые соответствуют заданному ключу поиска.

Назначение поиска – получение доступа к информации внутри элемента с целью ее обработки.

Таблица символов (или ***словарь***) – это структура данных элементов с ключами, которая поддерживает две базовые операции:

1. вставку нового элемента;
2. возврат элемента с заданным ключом.

Интерфейс АТД «Таблица символов»

```
template <class Item, class Key>
class ST
{
 private:
        //код, зависящий от реализации
 public:
        ST(int);//создание таблицы символов
        int count ();//подсчет количества элементов
        Item search(Key);//поиск элемента с заданным ключом
        void insert(Item);//вставка нового элемента
        void remove(Item);//удаление указанного элемента
        Item select(int);//выбор k-го по величине элемента в ТС
        void show(ostream &);//отображение элементов в порядке их ключей
};
```

Реализация АТД-элемента

```
static int maxKey=1000;
typedef int Key;
class Item
{
 private:
        Key keyval;
        float info;
 public:
        Item(){keyval=maxKey;}
        Key key(){return keyval;} //извлечение ключей из элементов
        int null(){return keyval==maxKey;} //проверка, является ли элемент нулевым
        void rand()//генерирует произвольный Item
        {
```

```
            keyval=1000*::rand()/RAND_MAX;
            info=1.0*::rand()/RAND_MAX;
        }
        int scan(istream& is=cin) //считывает Item
            {return (is>>keyval>>info)!=0;}
        void show(ostream& os=cout)//выводит Item
            {os<<keyval<<" "<<info<<endl;}
};
ostream& operator<<(ostream& os, Item& x) {x.show(os); return os;}
```

Поиск с использованием индексации по ключам

Данный алгоритм поиска предполагает хранение элементов в массиве, проиндексированном значениями ключей. Изначально массив инициализируется значениями NULL. Затем можно вставить элемент со значением K, записав его в A[k]. Найти элемент можно, обратившись в A[k]. Удалить элемент можно записав в A[k] значение NULL.

Пример.

Вставить элементы с ключами 5, 7, 6, 3, 10 в ТС с использованием индексации по ключам.

1. Индексируем массив значениями ключей и инициализируем значениями NULL

A[0] = NULL
A[1] = NULL
A[2] = NULL
A[3] = NULL
A[4] = NULL
A[5] = NULL
A[6] = NULL
A[7] = NULL
A[8] = NULL
A[9] = NULL
A[10] = NULL

2. Вставка элементов со значениями.

A[5] = "5"
A[7] = "7"
A[6] ="8"
A[3] = "3"
A[10] = "10"

3. Поиск элемента с ключом 6.

A[6] = "6"

4. Удаление элемента с ключом 7.

A[7] = NULL

Реализация ТС, основывающейся на индексированном по ключам массиве

Программа не обрабатывает дублированные ключи и в ней предполагается, что значения ключей лежат в пределах 0 … M-1.

```
template <class Item, class Key>

class ST
{
   private:
      Item nullItem, *st;
      int M;
```

```
  public:
    ST(int maxN)    {  M = nullItem.key(); st = new Item[M];  }
    int count()
    { int N = 0;
      for(int i=0; i<M; i++)   if(!st[i].null()) N++;
      return N;
    }
    void insert(Item x) { st[x.key()] = x; }
    Item search(Key v) { return st[v]; }
    void remove(Item x) { st[x.key()] = nullItem; }
    Item select(int k)
    {
       for (int i=0; i<M; i++)
        if(!st[i].null())  if(k-- == 0) return st[i];
       return nullItem;
    }
    void show(ostream& os)
    {   for(int i=0; i<M; i++)   if(!st[i].null()) st[i].show(os);  }
};
```

Последовательный поиск

Данный алгоритм поиска предполагает последовательное сохранение элементов в массиве в упорядоченном виде. Используется, когда диапазон ключей слишком велик. Для вставки элемента остальные сдвигаются. При поиске последовательно просматривается весь массив. Т.к. массив упорядочен, то при встрече ключа больше искомого можно сделать вывод о неудаче поиска. При удачном поиске используется в среднем $\frac{N}{2}$ сравнений, при неудачном – N.

Пример.

Вставить элементы с ключами 5, 7, 6, 3, 10 в ТС с использованием последовательного поиска.

1. Вставка элементов с ключами.

5
5 7
5 6 7
3 5 6 7
3 5 6 7 10

2. Поиск элемента с ключом 7.

3 != 7
5 != 7
6 != 7
7 = 7 - элемент найден

Реализация ТС с использованием массива

Поддержание упорядоченности массива обеспечивается тем, что при вставке нового элемента большие элементы смещаются с целью освобождения места. В этом случае функция поиска может выполнять в массиве поиск элемента с указанным ключом, возвращая nullItem при обнаружении элемента с большим ключом.

```
template <class Item, class Key>
```

```
class ST
{
   private:
      Item nullItem, *st;
      int N;
   public:
      ST(int maxN) { st = new Item[maxN+1]; N=0;}
      int count() { return N; }
      void insert(Item x)
      {
        int i = N++;
        Key v = x.key();
        while(i>0 && v<st[i-1].key()) { st[i] = st[i-1]; i--; }
        st[i] = x;
      }
      Item search(Key v)
      {
        for(int i=0; i<N; i++) if(!(st[i].key() < v)) break;
        if(v == st[i].key()) return st[i];
        return nullItem;
      }
      Item select(int k) { return st[k]; }
      void show(ostream& os)
      {
         int i=0;
         while (i<N) st[i++].show(os);
      }
};
```

Реализация ТС с использованием связного списка

Преимущество применения связных списков для реализации ТС состоит в том, что необязательно заранее точно определять максимальный размер таблицы. Недостаток – необходимость расхода дополнительного объема памяти под ссылки.

Реализация ТС на базе связного списка использует односвязный список, каждый узел которого содержит элемент с ключом и с ссылкой.

Функция insert помещает новый элемент в начало списка и выполняется за постоянное время. Функция search использует рекурсивную функцию для просмотра списка. Список не упорядочен.

```
#include <stdlib.h>
template <class Item, class Key>
class ST
{
  private:
    Item nullItem;
    struct node
    {
```

```
    Item item; node* next;
    node(Item x, node* t) { item = x; next = t; }
  };
  typedef node *link;
  int N;
  link head;
  Item searchR(link t, Key v)
  {
    if (t == 0) return nullItem;
    if (t->item.key() == v) return t->item;
    return searchR(t->next,v);
  }
 public:
  ST(int maxN) { head = 0; N = 0; }
  int count() { return N; }
  Item search(Key v) { return searchR(head, v); }
  void insert (Item x) { head = new node(x, head); N++; }
};
```

Бинарный поиск

Бинарный поиск – способ разделения наборов элементов на части, состоящий в поддержке элементов в отсортированном виде с последующим использованием индексов в отсортированном массиве для определения части массива, над которой будет выполняться дальнейшая работа. Используется процедура «разделяй и властвуй».

Алгоритм бинарного поиска: разделить упорядоченную ТС на 2 части, определить часть, в которой может лежать ключ и произвести с данной частью массива такую же операцию. Для определения, присутствует ли искомый ключ в заданном массиве, необходимо сравнить заданный ключ с элементом, находящимся в средней позиции массива. Если заданный ключ меньше среднего элемента массива, то дальше будет рассматриваться левая часть массива, если больше – правая часть массива.

Реализация бинарного поиска

```
private:
 Item searchR(int l, int r, Key V)
 {
  if (l > r) return nullItem;
  int m = (l + r)/2;
  if ( v == st[m].key() ) return st[m];
  if ( l == r ) return nullItem;
  if ( v < st[m].key() )  return searchR(l, m-1, v);
  else return searchR (m+1, r, v);
 }
public:
 Item search(Key v) { return searchR(0, N-1, v); }
```

При бинарном поиске используется не более, чем $\log_2 N + 1$ операций сравнения – как при удачном так и при неудачном поиске.

Пример.

Выполнить поиск элемента 25 в упорядоченном массиве с помощью алгоритма бинарного поиска.

3 7 13 19 23 25 29 34 35 56 98 99

В качестве серединного элемента выберем элемент 29.

3 7 13 19 23 25 29 34 35 56 98 99

Сравним: 25 < 29 - дальше рассматриваем левую часть массива

3 7 13 19 23 25

В качестве серединного элемента выберем элемент 19

3 7 13 19 23 25

Сравним: 25 > 19 - дальше рассматриваем правую часть массива

23 25

В качестве серединного элемента выберем элемент 25

23 25

Сравним: 25 = 25 - элемент найден. Поиск завершен

Поиск с использованием BST-дерева

BST-дерево – дерево бинарного поиска – это бинарное дерево, с каждым из внутренних узлов которого связан ключ, причем ключ в любом узле больше (или равен) ключам во всех узлах левого поддерева этого узла и меньше (или равен) ключам во всех узлах правого поддерева этого узла.

Узлы в BST-дереве определяются как содержащие элемент с ключом, левую и правую связи. Левая связь указывает на BST-дерево с меньшими (или равными) ключами, а правая – на BST-дерево с большими (или равными) ключами.

Рекурсивная программа, реализующая поиск на базе BST-дерева, принимает в качестве аргументов дерево и ключ. Процедура поиска завершается либо в случае нахождения элемента с искомым ключом, либо когда текущее поддерево становится пустым.

Реализация ТС с использованием BST-дерева

Ссылка head указывает на корень дерева.

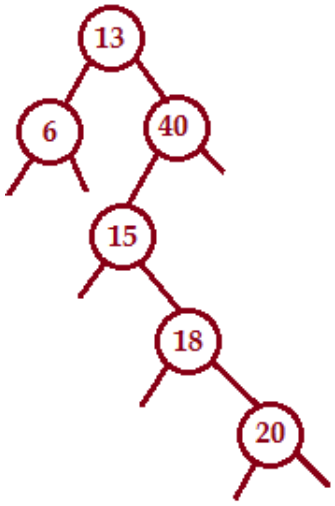
```
template <class Item, class Key>
class ST
{
  private:
    struct node
```

```
    {
      Item item; node *l, *r;
      node (Item x) { item = x; l = 0; r = 0; }
    };
    typedef node *link;
    link head;
    Item nullItem;
    Item serachR (link h, Key v)
    {
      if (h == 0) return nullItem;
      Key t = h->item.key();
      if (v == t) return h->item;
      if (v < t) return searchR (h->l, v);
        else return searchR (h->r, v);
    }
    void insertR (link& h, Item x)
    {
      if (h == 0) { h = new node(x); return; }
      if (x.key() < h -> item.key()) insertR (h->l, x);
        else insertR(h->r, x);
    }
  public:
    ST (int maxN) { head = 0; }
    Item search (Key v) { return searchR (head, v); }
    void insert (Item x) {insertR (head, x); }
};
```

Пример.

Вставить элементы 13, 40, 15, 6, 18, 20 в BST-дерево. Выполнить поиск элемента 18.

1. Вставка элементов в дерево бинарного поиска.



Вставка 13: 1-й элемент вставляем в корень, т.к. дерево пусто.

Вставка 40: 40>13 - вставка в правое поддерево

Вставка 15: 15>13 - правое поддерево
15<40 - вставка в левое поддерево

Вставка 6: 6<13 - вставка в левое поддерево

Вставка 18: 18>13 - правое поддерево
18<40 - левое поддерево
18>15 - вставка в правое поддерево

Вставка 20: 20>13 - правое поддерево
20<40 - левое поддерево
20>15 - правое поддерево
20>18 - вставка в правое поддерево

2. Поиск элемента 18.

Начинаем поиск с корня дерева.
18 > 13 - идем в правое поддерево.
18 < 40 - идем в левое поддерево.
18 > 15 - идем в правое поддерево.
18 = 18 - элемент найден. Поиск завершен.

Хеширование

Хеширование – расширенный вариант поиска с использованием индексирования по ключу, применяемый в приложениях поиска, в которых ключи расположены в широком диапазоне.

Хеширование основано на обращении к элементам в таблице за счет выполнения арифметических операций для преобразования ключей в адреса таблицы.

Алгоритмы поиска, использующие хеширование, состоят из двух частей:

1. вычисление хеш-функции, которая преобразует ключ поиска в адрес в таблице.

2. разрешение конфликтов, которые обрабатывают ключи, преобразованные в один и тот же адрес в таблице.

Преимущества: использует программный объем памяти и время выполнения при поиске.

Недостатки: а) время выполнения зависит от длины ключа; б) не обеспечивает эффективные реализации для других операций с таблицей символов (выбор, сортировка).

Вычисление хеш-функции. Хеш-функция преобразует ключ поиска в адрес в таблице. Хеш-функция зависит от типа ключа. Предлагается несколько вариантов хеш-функций.

а) Вычисление хеш-функций для ключей в диапазоне от 0 до 1 – ключи необходимо умножить на M (количество элементов в ТС) и округлить до ближайшего целого. Полученное число – адрес в ТС в диапазоне от 0 до M-1.

б) Вычисление хеш-функций для ключей в диапазоне от s до t – по формуле:

$$Хеш-значение = \left(\frac{Ключ - s}{t - s} \right) * M;$$

```
inline int hash (Key v,   int M) {return   (int)  M*(v-s)/(t-s);}
```

в) Вычисление хеш-функций для целых ключей – вычисление остатка от деления на M (количество элементов в ТС):

$$Хеш-значение = Ключ \% M;$$

г) Вычисление хеш-функции для строковых ключей – необходимо вычислить десятичное число, соответствующее коду символа строки, умножая полученное значение на 128, а затем добавляя кодовое значение следующего символа.

```
int hash (char *v, int M) {
 int h = 0; a = 127;
 for (; *v != 0; v++) h = (a*h + *v) % M;
 return h;      }
```

Еще более эффективный подход — использование случайных значений коэффициентов в вычислении и другого случайного значения для каждой цифры ключа. Такой подход дает рандомизованный алгоритм, называемый универсальным хешированием.

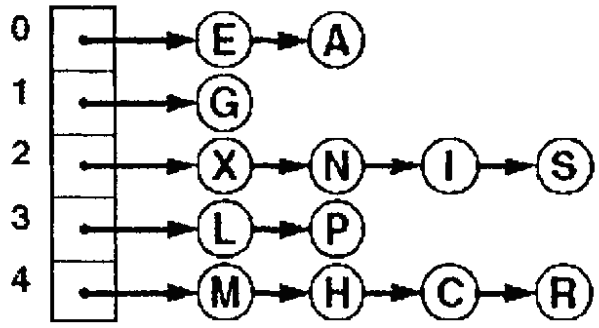
```
int hashU (char *v, int M)
{
  int h, a = 31415, b = 27183;
  for (h = 0; *v != 0; v++, a = a*b % (M-1))     h = (a*h + *v) % M;
  return (h < 0) ? (h + M) : h;
}
```

Хеширование методом раздельного связывания

- построение для каждого адреса таблицы связного списка элементов, ключи которых отображаются на этот адрес. Таким образом формируется M связных списков. То есть конфликтующие элементы объединяются в отдельные связные списки.

Пример.

A	S	E	R	C	H	I	N	G	X	M	P	L
0	2	0	4	4	4	2	2	1	2	4	3	3



Реализация хеширования методом раздельного связывания

Конструктор устанавливает M так, что каждый список должен содержать около 5 элементов.

Раздельное связывание уменьшает количество выполняемых при последовательном поиске сравнений в среднем в M раз при использовании дополнительного объема памяти для M связей.

Для раздельного связывания используются неупорядоченные списки. Для вставки требуется постоянное время, для поиска – время, пропорциональное N/M.

```
private:
  link* heads;
  int N, M;
public:
  ST(int maxN)
  {
     N = 0; M = maxN/5;
     heads = new link[M];
     for (int i = 0; i < M; i++) heads[i] = 0;
  }
  Item search(Key v)    { return searchR(heads[hash(v, M)], v); }
  void insert (Item item)
  {      int i = hash (item.key(), M);
         heads[i] = new node (item, heads[i]); N++;    }
```

Раздельное связывание уменьшает количество выполняемых при последовательном поиске сравнений в среднем в M раз при использовании дополнительного объема памяти для M связей.

Для раздельного связывания используются неупорядоченные списки. Для вставки требуется постоянное время, для поиска – время, пропорциональное N/M.

Хеширование методом линейного зондирования

- простейший метод хеширования с открытой адресацией, при которой разрешение конфликтов основывается на наличии пустых мест в таблице.

При наличии конфликта (когда хеширование выполняется в место таблицы, которое уже занято элементом с ключом, не совпадающим с ключом поиска) необходимо проверить следующую позицию в таблице.

Линейное зондирование характеризуется выявлением одного из трех исходов зондирования:

1. ели позиция таблицы содержит элемент, ключ которого совпадает с искомым, имеет место попадание при поиске.
2. если позиция таблицы содержит элемент, ключ которого не совпадает с искомым, необходимо зондировать позицию таблицы со следующим по величине индексом, продолжая этот процесс до тех пор, пока не будет найден искомый ключ или пустая позиция таблицы.
3. если содержащий искомый ключ элемент должен быть вставлен вслед за неудачным поиском, он помещается в пустую область таблицы, в которой поиск был завершен.

Для вставки нового элемента в ТС выполняется хеширование в позицию таблицы и сканирование вправо с целью нахождения незанятой позиции, используя в незанятых позициях нулевые элементы в качестве служебных.

Для поиска элемента с данным ключом мы обращаемся к позиции ключа в хеш-таблице и выполняем сканирование для отыскания совпадения, завершая процесс после нахождения незанятой позиции.

Пример. **Вставить ключи в ТС размером 13 с помощью хеширования методом линейного зондирования.**

A	S	E	R	C	H	I	N	G	X	M	P
7	3	9	9	8	4	11	7	10	12	0	8

1. Вставка элентов в таблицу:

0	1	2	3	4	5	6	7	8	9	10	11	12
G	X	M	S	H	P		A	C	E	R	I	N

Реализация хеширования методом линейного зондирования

Элементы хранятся в ТС, размер которой вдвое превышает максимально ожидаемое количество элементов и инициализируются значением nullItem. Конструктор устанавливает M так, чтобы таблица была заполнена менее, чем наполовину.

```
private:
  Item *st;
  int N, M;
  Item nullItem;
public:
  ST(int maxN)
  {
     N = 0; M = 2*maxN;
     st = new Item[M];
     for (int i = 0; i < M; i++) st[i] = nullItem;
  }
  int count() const { return N; }
  void insert (Item item)
  {
     int i = hash (item.key(), M);
     while (!st[i].null()) i = (i+1) % M;
     st[i] = item; N++;
  }
  Item search(Key v)
  {
     int i = hash(v, M);
     while (!st[i].null())
     if (v == st[i].key()) return st[i];
       else i = (i+1) % M;
     return nullItem;
  }
```

Задание

Вариант выбирается согласно номеру студента в списке группы.

Вариант 1. Разработать клиентскую программу, демонстрирующую возможности таблицы символов, реализованной на базе дерева бинарного поиска. На основании эмпирического анализа оценить временную эффективность алгоритмов создания таблицы символов, реализованной на основе рассмотренной структуры.

Вариант 2. Разработать клиентскую программу, демонстрирующую возможности таблицы символов, реализованной на базе хеширования методом раздельного связывания. На основании эмпирического анализа оценить временную эффективность алгоритмов создания таблицы символов, реализованной на основе рассмотренной структуры.

Вариант 3. Разработать клиентскую программу, демонстрирующую возможности таблицы символов, реализованной на базе хеширования методом линейного зондирования. На основании эмпирического анализа оценить временную эффективность алгоритмов создания таблицы символов, реализованной на основе рассмотренной структуры.

Вариант 1 для достаточного уровня. Разработать клиентскую программу, демонстрирующую возможности таблицы символов, реализующую последовательный поиск. Провести эмпирический анализ производительности реализованного алгоритма поиска.

Вариант 2 для достаточного уровня. Разработать клиентскую программу, демонстрирующую возможности таблицы символов, реализующую бинарный поиск. Провести эмпирический анализ производительности реализованного алгоритма поиска.

Контрольные вопросы

1. Что представляет собой поиск и какова его цель?
2. Что представляет собой таблица символов?
3. В чем заключается поиск с использованием индексации по ключам, последовательный и бинарный поиск? Какова его производительность?
4. В чем заключается поиск с использованием BST-дерева? Какова его производительность?
5. Что представляет собой хеширование? Хеш-функция? Какие задачи необходимо решить при использовании хеширования?
6. Методы хеширования и их назначение.

Лабораторная работа №4

АЛГОРИТМЫ ПОИСКА НА ГРАФАХ

Цель работы: изучение структуры данных «Граф», методов представления графов и алгоритмов поиска на графах.

Теоретические сведения

Граф **–** некоторое множество вершин и некоторое множество ребер, соединяющих пары различных вершин. Одно ребро может соединять максимум одну пару вершин.

V – число вершин в графе;

E – число ребер в графе.

Петля – ребро, замыкающееся на одну и ту же вершину.

Граф, состоящий из V вершин, содержит не более V(V-1)/2 ребер.

Общее число возможных пар вершин V^2, в т.ч. V петель.

Ребро, соединяющее две вершины - ***инцидентно*** этим вершинам.

Вершины, соединенные одним ребром – ***смежные***.

Степень вершины – число ребер, инцидентных этой вершине.

Обозначение v-w соответствует обозначению ребра, соединяющего вершины v и w.

Путь в графе – это последовательность вершин, в которой каждая следующая вершина после первой является смежной с предыдущей вершиной на этом пути.

Цикл – путь, у которого первая и последняя вершина одна и та же.

Длина пути – количество образующих этот путь ребер. Каждая отдельная вершина – путь длины 0.

Связный граф – граф, у которого существует путь из каждой вершины в любую другую вершину этого графа.

Несвязный граф – состоит из некоторого множества связных компонент, которые представляют собой связные подграфы.

Насыщенный граф – граф, средняя степень вершин которого пропорциональна V; число ребер E пропорционально V^2.

Разреженный граф – граф, дополнение которого насыщено; число ребер E не пропорционально V^2.

В зависимости от того с насыщенным или разреженным графом работаем, зависит выбор эффективного алгоритма обработки графа.

Неориентированные графы – графы, у которых ребра не имеют направления.

Ориентированные графы (орграфы) – графы, ребра которых однонаправленные, т.е. ориентированные. Первая вершина ориентированного ребра – начало, вторая вершина ориентированного ребра – конец.

Полустепень исхода вершины – число ребер, для которых вершина служит началом.

Полустепень захода вершины – число ребер, для которых вершина служит концом.

Направленный цикл в орграфе – это цикл, в котором пары смежных вершин появляются в порядке, указанном ребрами графа.

Сток – вершина с полустепенью исхода 0.

Исток – вершина с полустепенью захода 0.

Граф DAG (Directed Acyclic Graph – ориентированный ациклический граф) – орграф, который не содержит направленных циклов.

Взвешенный граф – граф, с каждым ребром которого связаны числа (веса), которые в общем случае представляют собой расстояние либо стоимость. Можно присвоить вес каждой вершине либо несколько весов каждой вершине и каждому ребру.

Интерфейс АТД «Граф»

Конструктор GRAPH принимает два аргумента: целое, задающее число вершин и булевое, которое показывает, является ли граф ориентированным или неориентированным.

Класс adjIterator позволяет проводить обработку любых вершин, смежных по отношению к заданной вершине.

```
struct Edge
{
   int v,w;
   Edge(int v=-1, int w=-1): v(v), w(w){}
};

class GRAPH
{
private:

public:
   GRAPH(int, bool);
   ~GRAPH();
   int V() const;
   int E() const;
   bool directed() const;
   int insert(Edge);
   int remove(Edge);
   bool edge(int, int);
};

class adjIterator
{
public:
   adjIterator(const GRAPH &, int);
   int beg();
```

```
    int nxt();
    bool end();
};
```

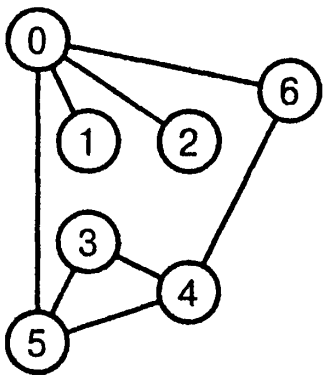
Пример клиентской функции обработки графов:

```
template <class Graph>
vector <Edge> edges (Graph &G)
{
    int E = 0;
    vector <Edge> a(G.E());
    for (int v=0; v<G.V(); v++)
    {
        typename Graph::adjIterator A(G,v);
        for(int w = A.beg(); !A.end(); w = A.nxt())
            if (G.directed() || v<w)
            a[E++] = Edge(v,w);
    }
    return a;
}
```

<u>Представление графа в виде матрицы смежности</u>

- матрица булевских значений размерности VxV, элемент которой, стоящий на пересечении v-й строки и w-го столбца принимает значение 1, если в графе есть ребро, соединяющее вершину v с вершиной w, и 0 в противном случае.

На такую реализацию затрачивается пространство памяти и время, пропорциональное V^2.



	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	0	0	1	1
1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0
3	0	0	0	0	1	1	0
4	0	0	0	1	0	1	1
5	1	0	0	1	1	0	0
6	1	0	0	0	1	0	0

На неориентированном графе необходимо, чтобы каждое ребро было представлено двумя вхождениями: ребро v-w представлено значением true как для adj[v][w], так и для adj[w][v], что соответствует ребру w-v.

Для инициализации матрицы смежности значением false необходимо время, пропорциональное V^2, независимо от того, сколько ребер в графе.

Просмотр матрицы для нахождения значений true необходимо время, пропорциональное числу вершин графа V.

Чтобы добавить в граф ребро, необходимо присвоить элементам указанной матрицы значение true. Чтобы удалить ребро – присвоить указанным элементам матрицы значение false.Включение и удаление ребер происходит за постоянное время.

Функция edge используется для проверки существования ребра.

```
class DenseGRAPH
{
   int Vcnt, Ecnt;
   bool digraph;
   vector <vector <bool>> adj;
public:
   DenseGRAPH(int v, bool digraph = false):
      adj(v), Vcnt(v), Ecnt(0), digraph(digraph)
      {
         for(int i=0; i<v; i++)
            adj[i].assign(v,false);
      }
   int V() const{return Vcnt;}
   int E() const{return Ecnt;}
   bool directed() const{return digraph;}
   void insert(Edge e)
   {
      int v=e.v; w=e.w;
      if(adj[v][w] == false) Ecnt++;
      adj[v][w] = true;
      if(!digraph) adj[w][v] = true;
   }
   void remove(Edge e)
   {
      int v=e.v; w=e.w;
      if(adj[v][w] == true) Ecnt--;
      adj[v][w] = false;
      if(!digraph) adj[w][v] = false;
   }
   bool edge (int v, int w) const
   {
      return adj[v][w];
   }
   class adjIterator;
   friend class adjIterator;
};
```

Итератор для представления матрицы смежности:

```
class DenseGRAPH::adjIterator
{
   const DenseGRAPH &G;
```

```
   int i,v;
public:
   adjIterator(const DenseGRAPH &G, int v):
      G(G), v(v), i(-1){}
      int beg(){  i=-1; return nxt();}
      int nxt()
      {
         for(i++; i<G.v(); i++)
            if(G.adj[v][i]==true) return i;
         return -1;
      }
      bool end() {return i>=G.v();}
};
```

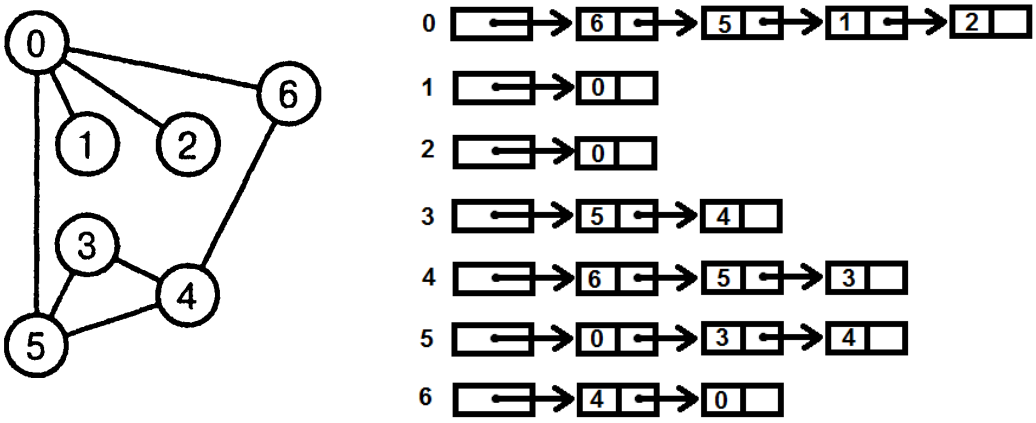
Данная реализация использует индекс i для просмотра значений строки v матрицы смежности (adj[v]), пропуская элементы, имеющие значения false.

Вызов функции beg(), за которым следует последовательность вызовов функции nxt()(проверка того, что значением функции end() является false перед каждым таким вызовом), позволяет получить последовательность вершин, смежных с вершиной v графа G в порядке возрастания индексов вершин.

<u>Представление графа в виде списка смежных вершин</u>

- отслеживание всех вершин, соединенных с каждой вершиной, включенной в связный список этой вершины.

На такую реализацию затрачивается пространство памяти, пропорциональное V+E.



Чтобы найти индексы вершин, связанных с заданной вершиной v, просматривают v-ю позицию этого массива, которая содержит указатель на связанный список, содержащий один узел для каждой вершины, соединенной с v.

Чтобы добавить в представление графа ребро, соединяющее вершину v с вершиной w, добавить w в список смежности вершины v и v в список смежности вершины w.

Функция insert включает ребро за постоянное время за счет отказа от проверки наличия дубликатов ребер; общее пространство используемой памяти пропорционально V+E.

```
class SparseMultiGRAPH
{
 int Vcnt, Ecnt;
 bool digraph;
 struct node
 {
    int v;
    node * next;
    node(int x, node * t){v=x; next=t;}
 };

 typedef node* link;
 vector <link> adj;

 public:
    SparseMultiGRAPH(int v, bool digraph=false):
       adj(v), Vcnt(v), Ecnt(0), digraph(digraph)
       { adj.assign(v,0);}
   int V() const{return Vcnt;}
   int E() const{return Ecnt;}
   bool directed() const {return digraph;}

   void insert(Edge e)
   {
      int v=e.v; w=e.w;
      adj[v]=new node(w, adj[v]);
      if(!digraph)adj[w]=new node(v,adj[w]);
      Ecnt++;
   }
   void remove(Edge e);
   bool edge(int v, int w) const;
   class adjIterator;
   friend class adjIterator;
};
```

Итератор для представления графа в виде списка смежных вершин:

```
class SparseMultiGRAPH::adjIterator
{
   const SparseMultiGRAPH &G;
   int v;
   link t;
```

```
public:
   adjIterator(const SparseMultiGRAPH &G, int v):
      G(G), v(v) {t=0;}
   int beg() {   t=G.adj[v]; return t?t->v:-1;    }
   int nxt() {if(t) t=t->next; return t?t->v:-1;  }
   bool end() {return t==0;}
};
```

В случае разреженных графов, когда V^2 огромно по сравнению с V+E, предпочтительнее использовать списки. В случае насыщенного графа, когда E и V^2 сравнимы – предпочтительнее использовать матрицу смежности.

Для взвешенных графов матрица смежности наполняется структурами, содержащими информацию о ребрах (включая данные об их наличии или отсутствии), заменяя ими соответствующие булевые значения; в представлении графа в виде списков смежных вершин эту информацию включают в элементы списков смежных вершин.

В представлениях орграфа в виде матрицы смежности и в виде списка смежных вершин каждое ребро представляется только один раз.

Алгоритмы поиска на графах

- используют перемещение по ребрам графа при переходе от вершины к вершине с целью систематического обхода каждой вершины и каждого ребра графа и изучением по мере продвижения структурных свойств графа.

BFS – Breadth First Search – поиск в ширину – основан на поддержке очереди FIFO.

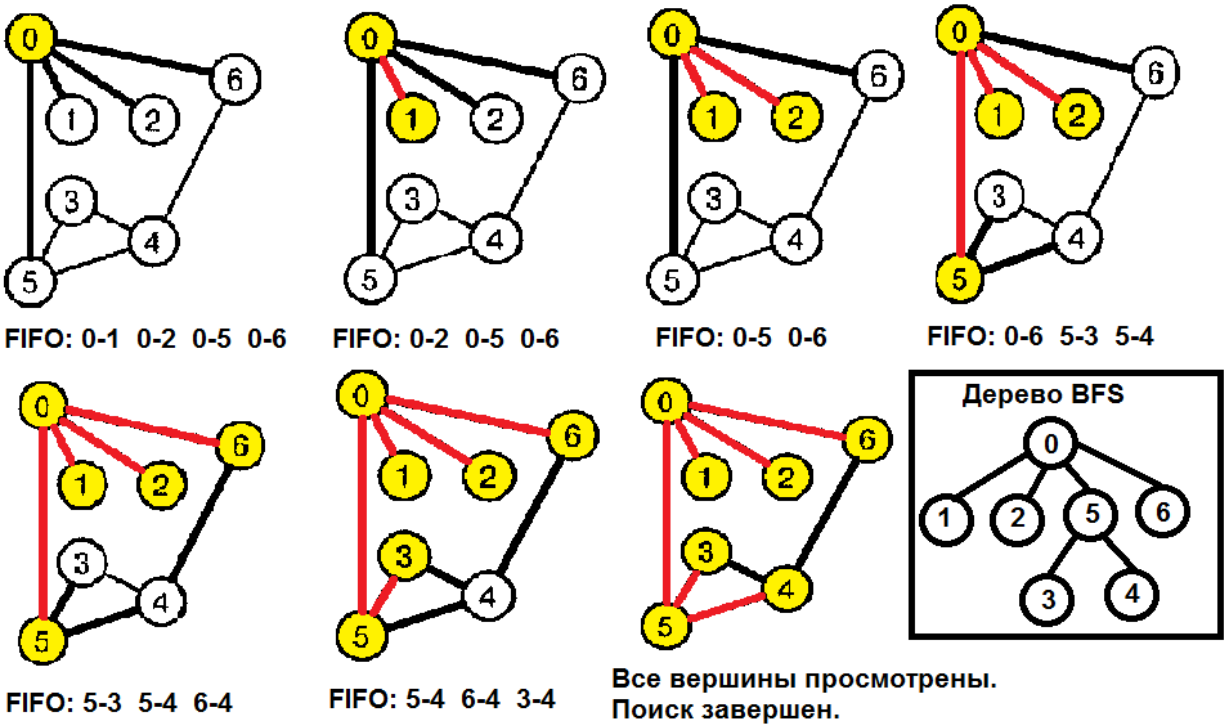
1. Поместить исходную вершину в очередь.
2. Просмотреть эту вершину, поставить в очередь все ребра, исходящие из этой вершины в еще не посещенные вершины. Не нужно помещать в очередь ребро с той вершиной назначения, что и у хотя бы одного ребра, находящегося в очереди.
3. Просмотреть следующую вершину, стоящую в очереди и проделать пункт 2.
4. Продолжать просмотр вершин из очереди до тех пор пока очередь не станет пустой.

Свойства BFS:

- в процессе поиска в ширину вершины поступают в очередь FIFO и покидают ее в порядке, определяемом их расстоянием от исходной вершины;

- для любого узла w в дереве BFS с корнем в вершине v путь из v в w соответствует кратчайшему пути из v в w в соответствующем графе.

- поиск в ширину посещает все вершины и ребра графа за время, пропорциональное V^2, в случае представления графа в виде матрицы смежности, и за время, пропорциональное V+E, в случае представления графа в виде списков смежных вершин.



Программная реализация поиска в ширину

Посещаемая вершина на основании просмотра инцидентных ей ребер, помещая все ребра, ведущие на не посещенную вершину, в очередь вершин, планируемых для просмотра. Вершины маркируются в порядке их посещения вектором ord. Функция search, вызываемая конструктором, выполняет построение явного представления дерева BFS с родительскими связями (ребра, которые приводят на каждый узел первый раз) в другом векторе st, который можно использовать для решения базовой задачи поиска кратчайшего пути.

```
#include "QUEUE.cc"
template <class Graph>
class BFS: public SEARCH <Graph>
{
   vector <int> st;
   void searchC(Edge e)
   {
      QUEUE <Edge> Q;
      Q.put(e);
      while (!Q.empty())
         if(ord[(e=Q.get()).w]==-1)
      {
         int v=e.v, w=e.w;
         ord[w]=cnt++; st[w]=v;
         typename Graph::adjIterator A(G,w);
         for(int t=A.beg(); !A.end(); t=A.nxt())
            if(ord[t]==-1) Q.put(Edge(w,t));
      }
   }
```

```
public:
   BFS(Graph &G): SEARCH<Graph> (G), st(G.v(),-1)
   {       search(); }
   int ST(int v) const {return st[v];}
};
```

DFS –Depth First Search – поиск в глубину – основан на рекурсивном обходе графа.

1. Посетить исходную вершину.
2. Посетив вершину, ее необходимо обозначить меткой, которая свидетельствует о том, что она посещена.
3. Затем в режиме рекурсии посещаются все смежные с данной вершины, которые еще не были помечены.

Применяя поиск в глубину к неориентированному графу, проверяются два представления каждого ребра. Если встречается ребро v-w, то либо выполняется рекурсивный вызов (если вершина w не помечена), или пропускается это ребро (если вершина w помечена). Когда встречается это ребро во второй раз с противоположной ориентацией w-v, то его игнорируем, т.к. вершину назначения v уже посещали.

Порядок обхода вершин зависит от графа, от его представления и реализации АТД. При представлении графа в виде матрицы смежности, исследуются ребра, инцидентные каждой вершине в цифровой последовательности. При представлении графа в виде списков смежных вершин, исследуются вершины в том порядке, в котором они выступают в списках.

Алгоритм поиска в глубину посещает все ребра и все вершины, связанные с начальной вершиной, независимо от того, в какой последовательности он проверяет ребра, инцидентные каждой вершине.

Поиск в глубину на графе, представленном матрицей смежности требует время, пропорциональное V^2.

Поиск в глубину на графе, представленном списками смежных вершин, требует времени, пропорциональное V+E.

Программная реализация поиска в глубину

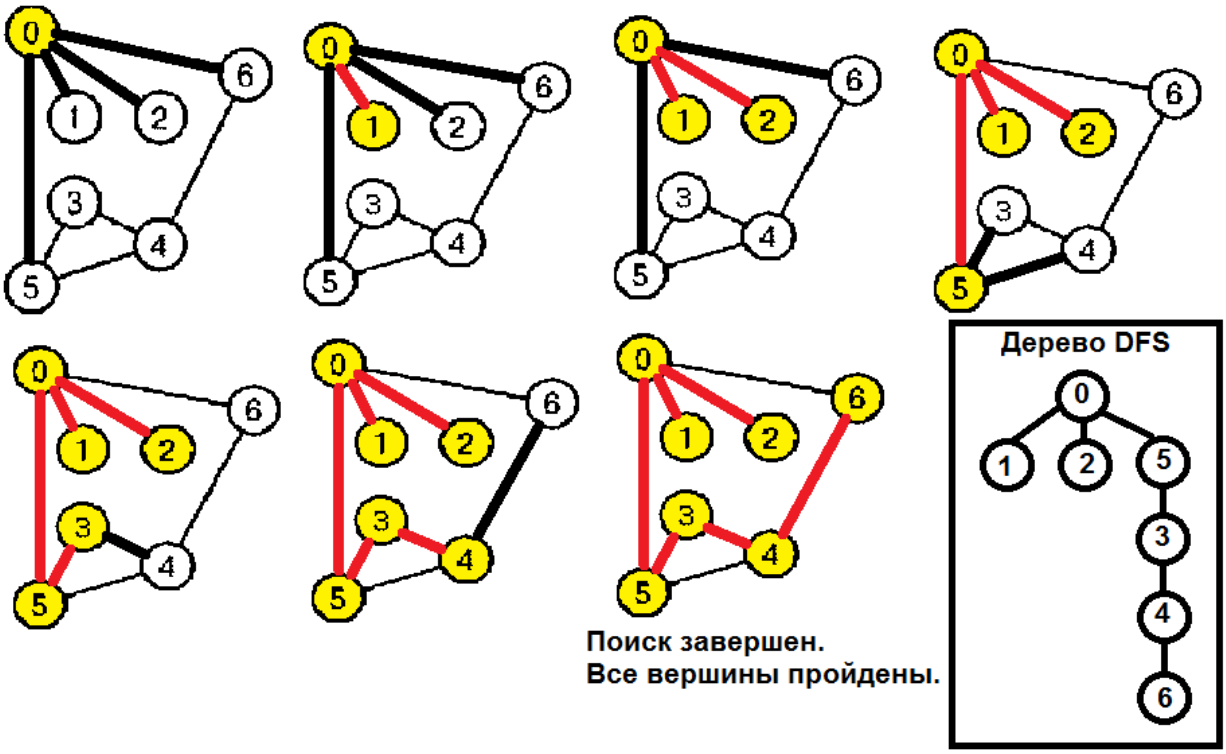
Конструктор помечает как посещенные все вершины графа через вызов рекурсивной функции searchC, которая обходит все вершины, смежные с V, подвергая их все проверке, и вызывая саму себя для каждого ребра, которое ведет из V в непомеченную вершину.

Функция count вычисляет число посещенных вершин.

Оператор [] перегружен для определения последовательности, в которой алгоритм посещал вершины.

```
#include <vector>
template <class Graph> class DFS
{
   int cnt;
```

```
    const Graph &G;
    vector <int> ord;
    void searchC(int v)
    {
       ord[v]=cnt++;
       typename Graph::adjIterator A(G,v);
       for(int t=A.beg(); !A.end(); t=A.nxt())
          if(ord[t]==-1) searchC(t);
public:
    cDFS(const Graph &G, int v=0):
       G(G), cnt(0), ord(G.v(),-1)        {          searchC(v);       }
       int count() const{return cnt;}
       int operator [] (int v)const{return ord[v];}
    }    };
```



Задание

1. Разработать клиентскую программу, реализующую алгоритмы поиска на графе согласно варианту. Граф представить в виде матрицы смежности или списка смежных вершин в зависимости от варианта.
2. Провести эмпирический анализ производительности указанного алгоритма поиска на графе.

№ студента в списке	КН-1		КН-2		КН-с	
	Поиск на графе	Представление графа	Поиск на графе	Представление графа	Поиск на графе	Представление графа
1	В ширину	Матрица смежности	В ширину	Матрица смежности	В ширину	Матрица смежности
2	В глубину	Матрица смежности	В ширину	Список смежных вершин	В ширину	Список смежных вершин
3	В ширину	Список смежных вершин	В глубину	Матрица смежности	В ширину	Матрица смежности
4	В глубину	Список смежных вершин	В глубину	Список смежных вершин	В глубину	Матрица смежности
5	В ширину	Матрица смежности	В ширину	Матрица смежности	В глубину	Список смежных вершин
6	В глубину	Матрица смежности	В ширину	Список смежных вершин	В глубину	Список смежных вершин
7	В ширину	Список смежных вершин	В глубину	Матрица смежности	В ширину	Матрица смежности
8	В глубину	Список смежных вершин	В глубину	Список смежных вершин	В ширину	Список смежных вершин
9	В ширину	Матрица смежности	В ширину	Матрица смежности	В ширину	Матрица смежности
10	В глубину	Матрица смежности	В ширину	Список смежных вершин	В глубину	Матрица смежности
11	В ширину	Список смежных вершин	В глубину	Матрица смежности	В глубину	Список смежных вершин
12	В глубину	Список смежных вершин	В глубину	Список смежных вершин	В глубину	Список смежных вершин
13	В ширину	Матрица смежности	В ширину	Матрица смежности	В ширину	Матрица смежности
14	В глубину	Матрица смежности	В ширину	Список смежных вершин	В ширину	Список смежных вершин
15	В ширину	Список смежных вершин	В глубину	Матрица смежности	В ширину	Матрица смежности
16	В глубину	Список смежных вершин	В глубину	Список смежных вершин	В глубину	Матрица смежности
17	В ширину	Матрица смежности	В ширину	Матрица смежности	В глубину	Список смежных вершин
18	В глубину	Матрица смежности	В ширину	Список смежных вершин	В глубину	Список смежных вершин
19	В ширину	Список смежных вершин	В глубину	Матрица смежности	В ширину	Матрица смежности
20	В глубину	Список смежных вершин	В глубину	Список смежных вершин	В ширину	Список смежных вершин

Контрольные вопросы

1. Что представляет собой граф? Виды графов и способы представления.
2. Какие основные алгоритмы поиска на графах? Их производительность.
3. В чем заключается алгоритм поиска в ширину?
4. В чем заключается алгоритм поиска в глубину?

ЛИТЕРАТУРА

1. Gonnet. G. H. (Gaston H.). Handbook of algorithms and data structures : in Pascal and C / G. H. Gonnet, R. Baeza-Yates. – 2nd ed. – 1991.
2. Ridgway Scott L. Numerical Analysis. - Princeton University Press, 2011. – 342 P.
3. Skiena Steven S. The Algorithm Design Manual. /Second Edition. - Springer-Verlag London Limited 2008. – 739 P.
4. Ахо А., Хопкрофт Дж., Ульман Д. Структуры данных и алгоритмы.- М.: Издательский дом «Вильямс», 2001.- 384 с.
5. Левитин А. Алгоритмы: введение в разработку и анализ.-М.: Издательский дом «Вильямс», 2006.-576 с.
6. Макконнелл Дж. Основы современных алгоритмов. 2-е дополненное издание. Москва: Техносфера, 2004. – 368 с.
7. Методичні вказівки до виконання лабораторних робіт з дисципліни “Теорія алгоритмів і математичні основи подання знань”. Для студентів з напрямку “Комп’ютерні науки” /Укл.: Н.В. Лиса, К.Ю. Новікова, В.В. Помулев, О.О. Кавац, І.В. Стовпченко. Під ред. О.І. Михальова. – Дніпропетровськ: НМетАУ, 2011.– 66 с.
8. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ /Структуры данных / Сортировка /Поиск.- К.: Издательство «ДиаСофт», 2001.- 688с.