Федеральное государственное бюджетное образовательное учреждение

высшего образования «Сибирский государственный университет телекоммуникаций и информатики»

Факультет ИВТ

Кафедра вычислительных систем

**Курсовая работа**

на тему Разработка библиотеки сортировок

Вариант 1.2 Сортировка вставками, пирамидальная сортировка.

Выполнил:студент гр. ИС-242Воркунов К. С.

Проверил: Ст. преподаватель кафедры ВС

Фульман В.О.

Новосибирск, 2023

**Тема курсовой работы**

Разработка библиотеки сортировок (Сортировка методом пузырька, быстрая сортировка)

**Задание на курсовую работу**

Задание: Реализовать динамическую библиотеку сортировок. Алгоритмы сортировок выбираются в соответствии с вариантом задания. Проанализировать эффективность алгоритмов сортировки. Разработать демонстрационную программу, использующую созданную библиотеку.

**Критерии оценки**

Оценка «удовлетворительно»: алгоритмы реализованы в виде простой программы без применения библиотек, данные поступают на вход с клавиатуры. Тесты проведены только на небольших последовательностях, нет анализа и сравнения алгоритмов. Не предусмотрено динамическое выделение памяти под входные данные.

Оценка «хорошо»: работа выполнена в полном соответствии с заданием. Обязательно динамическое выделение памяти под входные данные.

Оценка «отлично»: помимо выполнения условий задания предусмотрена сортировка произвольных данных (по аналогии с функцией qsort библиотеки GNU C Library – GLibC). Обязательно динамическое выделение памяти под входные данные.

**Указания к выполнению задания**

Алгоритмы сортировки необходимо реализовать в подпрограммах. Подпрограммы выносятся в отдельную библиотеку, которая компилируется как динамическая. Информация о создании и использовании динамических библиотек может быть найдена на ресурсе FirstSteps: http://firststeps.ru/linux/general1.html. Эффективность сортировок оценивать по времени работы алгоритмов. По полученным результатам сформулировать выводы о преимуществах и недостатках каждого алгоритма. Сравнить полученные результаты с теоретическими оценками вычислительной сложности реализованных алгоритмов. Экспериментальные измерения необходимо провести как для упорядоченных данных (по возрастанию и по убыванию), так и случайных последовательностей, размер которых составляет 28 – 2 15 элементов (с некоторым шагом). Построить графики полученных зависимостей. В случае, если время работы одного из алгоритмов превышает 15 мин., прекратить измерения по данному алгоритму и строить график не на всем интервале.

......................................................

**Анализ задачи**

1. Для реализации динамической библиотеки нам нужно создать файл с функцией сортировки, а также заголовочный файл, в котором будет эта функция объявлена. Помимо функции сортировки в библиотеке находятся функции вывода массива, заполнения массива и вычисления времени работы сортировки.
2. Функция вычисления времени работы сортировки

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **double** **wtime**()  {  **struct** timeval t;  gettimeofday(&t, NULL);  **return** (**double**)t.tv\_sec + (**double**)t.tv\_usec \* **1E-6**;  } |

1. Функции генерации псевдослучайного числа и заполнения массива псевдослучайными числами

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **int** **getrand**(**int** min, **int** max)  {  **return** (**double**)rand() / (RAND\_MAX + **1.0**) \* (max - min) + min;  }  **void** **filling\_array**(**int** \*a, **int** min, **int** max, **int** x)  {  // Заполнение массива случайными числами  **for** (**int** i = **0**; i < x; i++)  a[i] = getrand( min, max);  } |

1. Функция заполнения массива по возрастания и убыванию

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // Функция для заполнения массива порядковыми числами, идущими по убыванию  **void** **fillingArray**(**int** arr[], **int** n) {  **for** (**int** i = **0**; i < n; i++) {  arr[i] = n - i;  }  }  // Функция для заполнения массива порядковыми числами, идущими по возрастанию  **void** **fillingArray**(**int** arr[], **int** x) {  **for** (**int** i = **0**; i < x; i++) {  arr[i] = i + **1**;  }  } |

1. Функция вывода массива

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **void** **printArray**(**int** arr[], **int** x) {  **for** (**int** i = **0**; i < x; i++) {  printf("%d ", arr[i]);  }  printf("**\n**");  } |

1. Для создания динамической библиотеки нужно правильно скомпилировать файл с функциями и задать ему расширение \*.so команда для компиляции выглядит так:

gcc –Wall –g –shared –fpic –o libsomecode.so somecode.c

С помощью опции **-shared** мы говорим компилятору, что хотим получить, на выходе библиотеку. А опция **-fpic** говорит компилятору, что объектные файлы должны содержать позиционно-независимый код (position independent code), который рекомендуется использовать для динамических библиотек. Также стоит обратить внимание на префикс **lib** в файле с расширением \*.so без него создание динамической библиотеки не получится.

1. Теперь скомпилируем наш исполняемый файл подключив к нему нашу библиотеку.

gcc –Wall –g –o main main.c –lsomecode –L.

с помощью **–l** мы указываем название библиотеки, а с помощью опции **–L** указываем текущую директорию, где лежит наша библиотека.

1. При запуске программы нам выдаст ошибку, чтобы ее не было, и программа работала нужно добавить нашу библиотеку в директорию это можно сделать с помощью команды:

export LD\_LIBRARY\_PATH=”$LD\_LIBRARY\_PATH:$PWD”

1. Так как постоянно компилировать вручную, а тем более удалять исполняемый и объектные файлы — это не удобно, напишем makefile. В нем реализуем сборку двух библиотек и удаление всех созданных файлов по окончанию.

**Теоритическое сравнение сортиро вок**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Алгоритм сортировки | Время работы в худшем случае | Время работы в среднем случае | Время работы в лучшем случае | Пространственная сложность |
| Сортировка вставками | O(n^2) | O(n^2) | O(n) | 1 |
| Пирамидальная сортировка | O(n log n) | O(n log n) | O(n log n) | 1 |

Исходя из таблицы можно сделать вывод, что эти сортировки подойдут в разных ситуациях. Сортировка вставками менее эффективна для больших объемов данных, но может быть более эффективной для небольших или почти отсортированных списков, в то время как сортировка кучей эффективна для больших объемов данных.

**Тестовые данные**

**Неотсортированный массив 28-215**

Для заполнения такого большого массива нам понадобиться функция getrand. В эту функцию подается максимальное и минимальное число, а она возвращает произвольно сгенерированное из этого промежутка. Числа которые поступали на вход для работы программы **min**=0; **max**=1000

|  |  |
| --- | --- |
| 1  2  3  4 | **int** **getrand**(**int** min, **int** max)  {  **return** (**double**)rand() / (RAND\_MAX + **1.0**) \* (max - min) + min;  } |

После запуска программы получились следующие данные:

|  |  |  |
| --- | --- | --- |
| Кол-во элементов | heap | insertion |
| 256 | 0,000043 | 0,000012 |
| 512 | 0,000087 | 0,000017 |
| 1024 | 0,000141 | 0,000056 |
| 2048 | 0,000332 | 0,000054 |
| 4096 | 0,000647 | 0,000088 |
| 8192 | 0,001629 | 0,000205 |
| 16384 | 0,003560 | 0,000374 |
| 32768 | 0,006287 | 0,000729 |

Благодаря им мы видим, что сортировка heap намного дольше чем insertion.

**Отсортированный массив 28-215 (по возрастанию)**

Для этого сравнения массив был заполнен порядковыми числами, которые шли по возрастанию.

После запуска программы получились следующие данные:

|  |  |  |
| --- | --- | --- |
| Кол-во элементов | heap | insertion |
| 256 | 0,000116 | 0,000005 |
| 512 | 0,000052 | 0,000006 |
| 1024 | 0,000141 | 0,000011 |
| 2048 | 0,000283 | 0,000019 |
| 4096 | 0,000574 | 0,000023 |
| 8192 | 0,001192 | 0,000038 |
| 16384 | 0,002477 | 0,000075 |
| 32768 | 0,004550 | 0,000185 |

В этом случае мы видим, что обе сортировки отработали заметно быстрее, heap отработала быстрее всего из всех своих отработок, но insertion все равно быстрее.

**Отсортированный массив 28-215 (по убыванию)**

Для этого сравнения массив был заполнен порядковыми числами, которые шли по убыванию.

После запуска программы получились следующие данные:

|  |  |  |
| --- | --- | --- |
| Кол-во элементов | heap | insertion |
| 256 | 0,000030 | 0,000007 |
| 512 | 0,000049 | 0,000007 |
| 1024 | 0,000122 | 0,000012 |
| 2048 | 0,000274 | 0,000017 |
| 4096 | 0,000563 | 0,000022 |
| 8192 | 0,001340 | 0,000054 |
| 16384 | 0,002257 | 0,000085 |
| 32768 | 0,005403 | 0,000180 |

В данном случае нет большой разницы с прошлым.

**Листинг программы**

**Сортировка вставками**

incertionsort.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93 | #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  #include <malloc.h>  **double** **wtime**()  {  **struct** timeval t;  gettimeofday(&t, NULL);  **return** (**double**)t.tv\_sec + (**double**)t.tv\_usec \* **1E-6**;  }  **void** **printArray**(**int** arr[], **int** x) {  **for** (**int** i = **0**; i < x; i++) {  printf("%d ", arr[i]);  }  printf("**\n**");  }  **int** **getrand**(**int** min, **int** max)  {  **return** (**double**)rand() / (RAND\_MAX + **1.0**) \* (max - min) + min;  }  // int filling\_array(int \*a, int min, int max, int x)  // {  // // Заполнение массива случайными числами  // for (int i = 0; i < x; i++)  // a[i] = getrand( min, max);  // return a;  // }  // Функция для заполнения массива порядковыми числами, идущими по убыванию  **void** **fillingArray**(**int** arr[], **int** x) {  **for** (**int** i = **0**; i < x; i++) {  arr[i] = x - i;  }  }  // // Функция для заполнения массива порядковыми числами, идущими по возрастанию  // void fillingArray(int arr[], int x) {  // for (int i = 0; i < x; i++) {  // arr[i] = i + 1;  // }  // }  **void** **Sort**(**int** arr[], **int** n)  {  **int** i, key, j;  **for** (i = **1**; i < n; i++)  {  key = arr[i];  j = i - **1**;  **while** (j >= **0** && arr[j] > key)  {  arr[j + **1**] = arr[j];  j = j - **1**;  }  arr[j + **1**] = key;  }  }  **int** **insertionsort**(**int** x)  {  **int** min = **0**, max = **100000**;  **double** v = wtime();  **int** \*a = (**int**\*)malloc(x \* **sizeof**(**int**));    //заполнение массива  // filling\_array(a, min, max, x);  fillingArray(a, x);    // Вывод элементов массива до сортировки  // printArray(a, x);  **int** n = **sizeof**(a) / **sizeof**(a[**0**]);    Sort(a, n);  // Вывод элементов массива после сортировки  // printArray(a, x);    **double** m = wtime();  printf("%lf**\n**", m - v);  **return** **0**;  } |

imain.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include <stdio.h>  #include "insertionsort.h"  **int** **main**()  {  **int** x;  scanf("%d", &x);  insertionsort(x);  **return** **0**;  } |

inserionsort.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  #include <malloc.h>  **double** **wtime**();  **void** **printArray**(**int** arr[], **int** x);  **int** **getrand**(**int** min, **int** max);  **void** **filling\_array**(**int** \*a, **int** min, **int** max, **int** x);  **void** **siftDown**(**int** \*arr, **int** x, **int** bottom);  **void** **Sort**(**int** arr[], **int** n);  **int** **insertionsort**(**int** x);  **void** **fillingArray**(**int** arr[], **int** n); |

**Сортировка кучей**

heapsort.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117 | #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  #include <malloc.h>  **double** **wtime**()  {  **struct** timeval t;  gettimeofday(&t, NULL);  **return** (**double**)t.tv\_sec + (**double**)t.tv\_usec \* **1E-6**;  }  // void printArray(int arr[], int x) {  // for (int i = 0; i < x; i++) {  // printf("%d ", arr[i]);  // }  // printf("\n");  // }  **int** **getrand**(**int** min, **int** max)  {  **return** (**double**)rand() / (RAND\_MAX + **1.0**) \* (max - min) + min;  }  // void filling\_array(int \*a, int min, int max, int x)  // {  // // Заполнение массива случайными числами  // for (int i = 0; i < x; i++)  // a[i] = getrand( min, max);  // }  // // Функция для заполнения массива порядковыми числами, идущими по убыванию  // void fillingArray(int arr[], int n) {  // for (int i = 0; i < n; i++) {  // arr[i] = n - i;  // }  // }  // // Функция для заполнения массива порядковыми числами, идущими по возрастанию  // void fillingArray(int arr[], int x) {  // for (int i = 0; i < x; i++) {  // arr[i] = i + 1;  // }  // }  // Функция "просеивания" через кучу - формирование кучи  **void** **siftDown**(**int** \*arr, **int** x, **int** bottom)  {  **int** max; // индекс максимального потомка  **int** done = **0**; // флаг того, что куча сформирована  // Пока не дошли до последнего ряда  **while** ((x \* **2** <= bottom) && (!done))  {  **if** (x \* **2** == bottom) // если мы в последнем ряду,  max = x \* **2**; // запоминаем левый потомок  // иначе запоминаем больший потомок из двух  **else** **if** (arr[x \* **2**] > arr[x \* **2** + **1**])  max = x \* **2**;  **else**  max = x \* **2** + **1**;  // если элемент вершины меньше максимального потомка  **if** (arr[x] < arr[max])  {  **int** temp = arr[x]; // меняем их местами  arr[x] = arr[max];  arr[max] = temp;  x = max;  }  **else**  done = **1**; // пирамида сформирована  }  }  // Функция сортировки на куче  **void** **Sort**(**int** \*arr, **int** array\_size)  {  // Формируем нижний ряд пирамиды  **for** (**int** i = (array\_size / **2**); i >= **0**; i--)  siftDown(arr, i, array\_size - **1**);  // Просеиваем через пирамиду остальные элементы  **for** (**int** i = array\_size - **1**; i >= **1**; i--)  {  **int** temp = arr[**0**];  arr[**0**] = arr[i];  arr[i] = temp;  siftDown(arr, **0**, i - **1**);  }  }  **int** **heapsort**(**int** x)  {  **int** min = **0**, max = **100000**;  //int \*a; // указатель на массив  // int x;  // scanf("%d", &x);  **double** v = wtime();  **int** \*a = (**int**\*)malloc(x \* **sizeof**(**int**));  //заполнение массива  // filling\_array(a, min, max, x);  fillingArray(a, x);  // Вывод элементов массива до сортировки  // printArray(a, x);  Sort(a, x); // вызов функции сортировки  // Вывод элементов массива после сортировки  // printArray(a, x);  **double** m = wtime();  printf("%lf**\n**", m - v);  **return** **0**;  } |

hmain.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <stdio.h>  #include "heapsort.h"  **int** **main**()  {  **int** x;  scanf("%d", &x);  heapsort(x);  **return** **0**;  } |

heapsort.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  #include <malloc.h>  **double** **wtime**();  **void** **printArray**(**int** arr[], **int** x);  **int** **getrand**(**int** min, **int** max);  **void** **filling\_array**(**int** \*a, **int** min, **int** max, **int** x);  **void** **siftDown**(**int** \*arr, **int** x, **int** bottom);  **void** **Sort**(**int** \*arr, **int** array\_size);  **int** **heapsort**(**int** x);  **void** **fillingArray**(**int** arr[], **int** n); |

Makefile

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | TARGET1 = hmain  TARGET2 = imain  # Исходные файлы  SRCS1 = heapsort.c hmain.c  SRCS2 = insertionsort.c imain.c  # Объектные файлы  OBJS1 = $(SRCS1:.c=.o)  OBJS2 = $(SRCS2:.c=.o)  # Компилятор и флаги  CC = gcc  CFLAGS = -Wall -fPIC  # Имя динамической библиотеки  LIBRARY1 = libheapsort.so  LIBRARY2 = libinsertionsort.so  # Правило по умолчанию  **all:** $(TARGET1) $(TARGET2)  # Правило сборки heap библиотеки  $(TARGET1): $(OBJS1)  $(CC) -o $@ $^ -L. -lheapsort  # Правило сборки insertion библиотеки  $(TARGET2): $(OBJS2)  $(CC) -o $@ $^ -L. -linsertionsort  # Правило компиляции исходных файлов  %.o: %.c  $(CC) $(CFLAGS) -c $< -o $@  # Правило сборки динамической библиотеки heap  $(LIBRARY1): heapsort.o  $(CC) -shared -o $@ $^  # Правило сборки динамической библиотеки insertion  $(LIBRARY2): insertionsort.o  $(CC) -shared -o $@ $^  # Правило связывания программы с библиотекой  $(TARGET1): $(LIBRARY1)  # Правило связывания программы с библиотекой  $(TARGET2): $(LIBRARY2)  # Правило очистки файлов  **clean:**  rm -f $(OBJS1) $(TARGET1) $(LIBRARY1) $(OBJS2) $(TARGET2) $(LIBRARY2)  .PHONY: all clean |