Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»

Факультет ИВТ
Кафедра вычислительных систем

Курсовая работа

Разработка библиотеки сортировок

Выполнила:
студент гр. ИВ-822
Бобров Данил
Проверил:
Старший преподаватель Кафедры ВС
Перышкова Е.Н.

Новосибирск, 2019

Содержание

Тема курсового проекта:

Разработка библиотеки сортировок. Сортировка вставками, пирамидальная сортировка.

Цель работы:

Целью курсового проекта является разработка динамической библиотеки сортировок

Задание:

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

Анализ задачи:

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 2 ^ 8 – 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.

Создание динамической библиотеки

Для начала стоит сказать, что объектный файл создаваемый нашим проверенным способом вовсе не подходит для динамических библиотек. Связано это с тем, что все объектные файлы создаваемые обычным образом не имеют представления о том в какие адреса памяти будет загружена использующая их программа. Несколько различных программ могут использовать одну библиотеку, и каждая из них располагается в различном адресном пространстве. Поэтому требуется, чтобы переходы в функциях библиотеки (операции **goto**

на ассемблере) использовали не абсолютную адресацию, а относительную. То есть генерируемый компилятором код должен быть независимым от адресов, такая технология получила название **PIC - Position Independent Code**. В компиляторе **gcc** данная возможность включается ключом **-fPIC**.

Теперь компилирование наших файлов будет иметь вид:

```
dron:~# gcc -fPIC -c src/heap.c
dron:~# gcc -fPIC -c src/insertion.c
```

Динамическая библиотека это уже не архивный файл, а настоящая загружаемая программа, поэтому созданием динамических библиотек занимается сам компилятор **gcc**. Для того, чтобы создать динамическую библиотеку надо использовать ключ **-shared**:

```
dron:~# gcc -shared heap.o insertion.o -o libmyfunc.so
```

В результате получим динамическую библиотеку **libfsdyn.so.** Теперь, чтобы компилировать результирующий файл с использованием динамической библиотеки нам надо собрать файл командой:

```
dron:~# gcc -c src/main.c
dron:~# gcc main.o -L. -lmyfunc -o dynamic
```

Если сейчас попробуем запустить файл **dynamic**, то получим ошибку:

```
dron:~# ./dynamic
./dynamic: error in loading shared libraries: libfsdyn.so: cannot open
shared object file: No such file or directory
dron:~#
```

Это сообщение выдает динамический линковщик. Он просто не может найти файл нашей динамической библиотеки. Дело в том, что загрузчик ищет файлы динамических библиотек в известных ему директориях, а наша директория ему не известна. Для настройки динамического линковщика существует ряд программ.

Первая программа называется **ldd**. Она выдает на экран список динамических библиотек используемых в программе и их местоположение. В качестве параметра ей сообщается название обследуемой программы. Давайте попробуем использовать ее для нашей программы **dynamic**:

```
dron:~# ldd dynamic
```

```
        libfsdyn.so => not found
        libc.so.6 => /lib/libc.so.6 (0x40016000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
dron:~#
```

Как видите все правильно. Программа использует три библиотеки:

- **libc.so.6** - стандартную библиотеку функций языка **C++**.
- **ld-linux.so.2** - библиотеку динамической линковки программ **ELF** формата.
- **libfsdyn.so** - нашу динамическую библиотеку функций.

Нашу библиотеку она найти не может. И правильно! Динамический линковщик ищет библиотеки только в известных ему каталогах, а каталог нашей программы ему явно не известен.

Для того, чтобы добавить нашу директорию с библиотекой в список известных директорий надо подредактировать файл **/etc/ld.so.conf**. Например, у меня этот файл состоит из таких строк:

```
dron:~# cat /etc/ld.so.conf
/usr/X11R6/lib
/usr/i386-slackware-linux/lib
/usr/i386-slackware-linux-gnulibc1/lib
/usr/i386-slackware-linux-gnuaout/lib
dron:~#
```

Во всех этих директории хранятся всеми используемые библиотеки. В этом списке нет лишь одной директории - **/lib**, которая сама по себе не нуждается в описании, так как она является главной. Получается, что наша библиотека станет "заметной", если поместить ее в один их этих каталогов. Это использование специальной переменной среды **LD_LIBRARY_PATH**, в которой перечисляются все каталоги содержащие пользовательские динамические библиотеки. Для того, чтобы установить эту переменную в командной среде **bash** надо набрать всего несколько команд. Для начала посмотрим есть ли у нас такая переменная среды:

```
dron:~# echo $LD_LIBRARY_PATH
```

У меня в ответ выводится пустая строка, означающая, что такой переменной среды нет. Устанавливается она следующим образом:

```
dron:~# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`pwd`
```

После этого программа **dynamic** будет прекрасно работать.

Если Вы обнулите эту переменную, то снова библиотека перестанет работать:

```
dron:~# LD_LIBRARY_PATH=""
dron:~# export LD_LIBRARY_PATH
dron:~# ./rezultdyn
./rezultdyn: error in loading shared libraries: libfsdyn.so: cannot open
shared object file: No such file or directory
dron:~#
```

Описание алгоритмов сортировок.

Пирамидальная сортировка (Heap Sort, «Сортировка кучей») — алгоритм сортировки, работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за $\Theta(n \log n)$ операций при сортировке nэлементов. Количество применяемой служебной памяти не зависит от размера массива (то есть, O(1)).

Алгоритм Heapsort предполагает подготовку списка, сначала превратив его в максимальную кучу. Затем алгоритм многократно заменяет первое значение списка последним значением, уменьшая диапазон значений, рассматриваемых в операции кучи, на единицу и просеивая новое первое значение в его позицию в куче. Это повторяется до тех пор, пока диапазон рассматриваемых значений не станет одним значением в длине.

Шаги:

1.Вызовите функцию max () из списка. Он создает кучу из списка в O (n) операциях.

2.Поменяйте местами первый элемент списка с последним элементом. Уменьшите рассматриваемый диапазон списка на один.

3.Вызовите функцию swap () в списке, чтобы переместить новый первый элемент в соответствующий индекс в куче.

4.Переходите к шагу (2), если рассматриваемый диапазон списка не является одним элементом.

Операция max () запускается один раз и имеет производительность O (n). Функция swap () имеет значение O (log n) и вызывается n раз. Следовательно, производительность этого алгоритма составляет O (n + n og n) = O (n log n).

Вставка сортировки (Insertion Sort) - это простой алгоритм сортировки, который создает окончательный отсортированный массив (или список) по одному элементу за раз. Он гораздо менее эффективен в больших списках, чем более продвинутые алгоритмы, такие как быстрая сортировка или сортировка слиянием. Тем не менее, вставка сортировки обеспечивает несколько преимуществ:

- Простая реализация

•Эффективен для небольших наборов данных, так же, как и другие алгоритмы квадратичной сортировки

•Более эффективен на практике, чем большинство других простых квадратичных (т. Е. O (*n* ^2)) алгоритмов, таких как выборочная сортировка или пузырьковая сортировка

•Адаптивный, т. Е. Эффективный для наборов данных, которые уже существенно отсортированы: сложность по времени равна O (*kn*), когда каждый элемент на входе находится не более чем на *k* мест от своего отсортированного положения

•Стабильный; т.е. не меняет относительный порядок элементов с равными ключами

•На месте; т.е. требуется только постоянное количество O (1) дополнительного пространства памяти

•Онлайн; т.е. может сортировать список по мере его получения

Листинг:

Пирамидальная сортировка.

heap.c

```
#include <time.h>
#include <sys/time.h>
double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}
int max (int *a, int n, int i, int j, int k) {
    int m = i;
    if (j < n && a[j] > a[m]) {
        m = j;
    }
    if (k < n && a[k] > a[m]) {
        m = k;
    }
    return m;
}
void swap (int *a, int n, int i) {
    while (1) {
        int j = max(a, n, i, 2 * i + 1, 2 * i + 2);
```

```
        if (j == i) {
            break;
        }
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
        i = j;
    }
}
 void heapSort (int *a, int n) {
    int i;
    for (i = (n - 2) / 2; i >= 0; i--) {
        swap(a, n, i);
    }
     for (i = 0; i < n; i++) {
        int t = a[n - i - 1];
        a[n - i - 1] = a[0];
        a[0] = t;
        swap(a, n - i - 1, 0);
    }
}
```

Сортировка вставками.

insertion.c

```
#include <time.h>
#include <sys/time.h>


double witime()
{
   struct timeval t;
   gettimeofday(&t, NULL);
   return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}
void InsertionSort(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;
   }
}
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
extern double wtime();
extern int max (int *a, int n, int i, int j, int k);
extern void swap (int *a, int n, int i);
extern void heapSort (int *a, int n);
extern double witime();
extern void InsertionSort(int *a, int n);
int main()
{
	FILE *sort = fopen("sort.dat", "w");
	int n;
	for (int i = 1; i <= 20; i++){
		n = i * 5000;
		int *arr = malloc(n * sizeof(int));
		for(int i = 0; i < n; i++)
		{
			arr[i] = rand() % 1500000;
		}
		double htime = wtime();
		heapSort(arr, n);
		htime = wtime() - htime;
		double itime = witime();
```

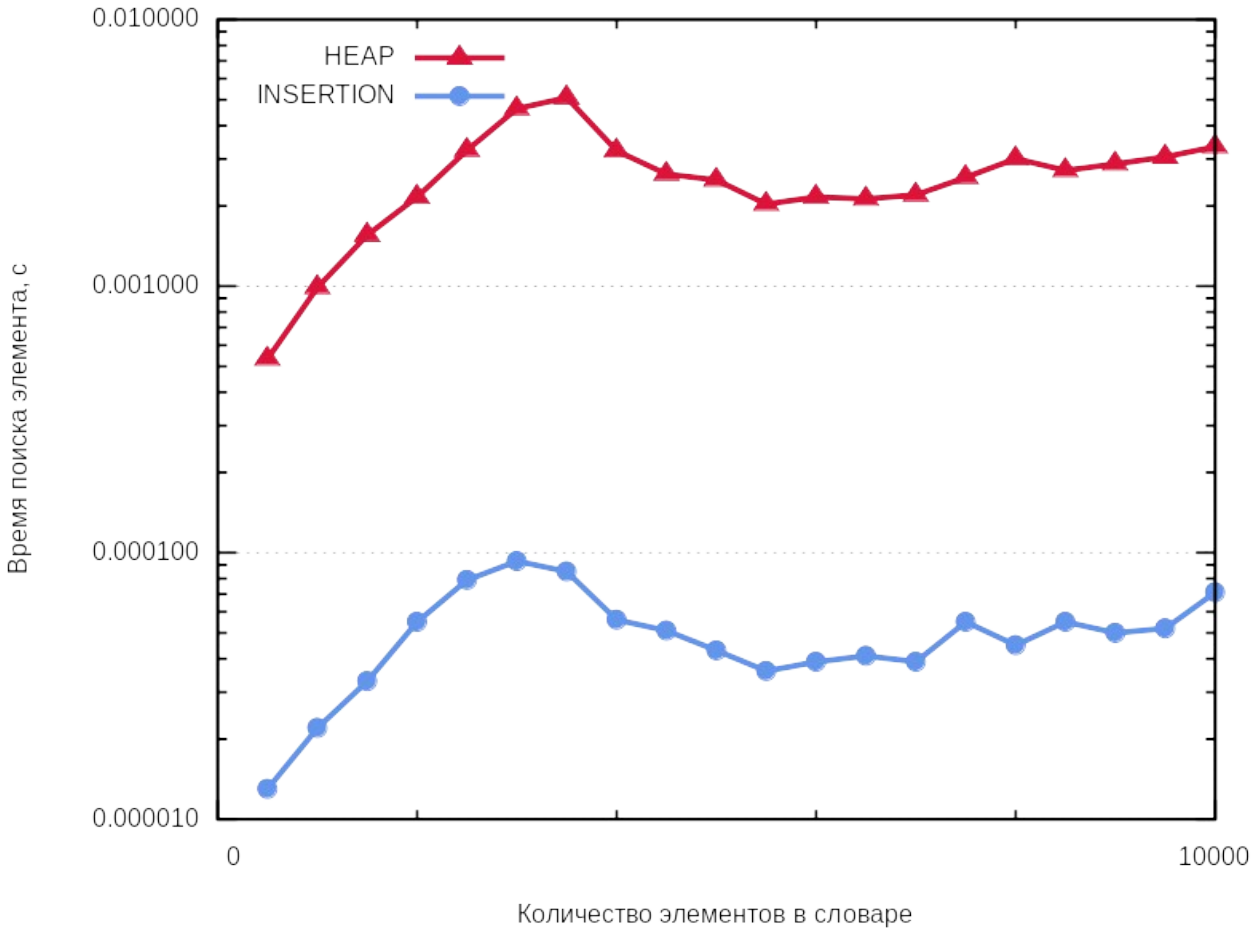
```
            InsertionSort(arr,n);

            itime = witime() - itime;

            fprintf(sort, " %d %f %f\n", n, htime, itime);

        }

        fclose(sort);

        return 0;

}
```

Makefile

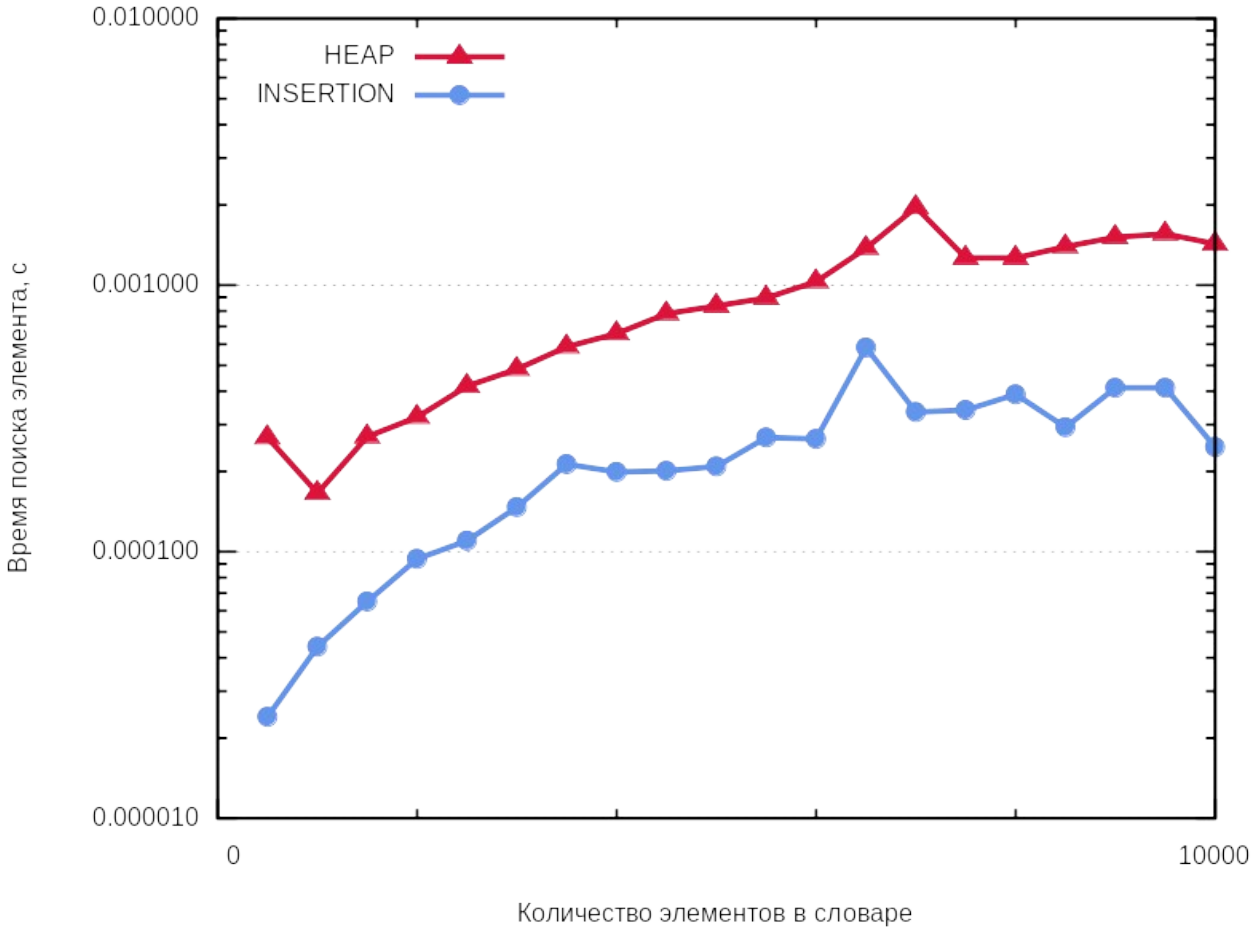
```
dynamic: src/main.c src/heap.c src/insertion.c

        gcc -fPIC -c src/heap.c

        gcc -fPIC -c src/insertion.c

        gcc -shared heap.o insertion.o -o libmyfunc.so

        gcc -c src/main.c

        gcc main.o -L. -lmyfunc -o dynamic

clean:

        rm -f main.o heap.o insertion.o
```

Графики:

Случайная последовательность.



В упорядоченной последовательности (убывание).



В упорядоченной последовательности (возрастание).