Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет прикладной математики и информатики
Кафедра вычислительной математики и программирования

Курсовой проект
По дисциплине
«Практикум на ЭВМ»
2 семестр

Задание IX
«Сортировка и поиск»

Студент:	Сикорский А. А.
Группа:	М8О-108Б-20
Преподаватель:	Трубченко Н. М.
Подпись:	
Оценка:	
Дата:	20.05.2021

Москва
2021

Содержание

Задание

Составить программу на языке Си с использованием процедур и функций для сортировки таблицы заданным методом и двоичного поиска по ключу в таблице. Программа должна вводить значения элементов таблицы и проверять работу процедуры сортировки. Тестовые данные необходимо заранее поместить в текстовые файлы. В моем варианте используется комлексный ключ размером 16 байт. Получается, что действительной и мнимой частью должны быть числа типа long long по 8 байт каждое. Сортировка организуется методом простой вставки.

1 Программа

1.1 Структура данных

Таблица выполнена в виде двунаправленного списока. Это такая структура данных, написанная на указателях, в которой нам гарантируется вставка и удаление за $O(1)$. Еще к преимуществам можно отнести потенциально больший размер списка по сравнению с массивом, ведь список не требует последовательного выделения памяти, но за это стоит платить невозможностью прямого доступа к элементу, как мы это делаем в массиве по индексу. Двунаправленность значит, что в каждом элементе списка содержится указатель не только на следующий, но и на предыдущий элемент. Таким образом доступна навигация как влево, так и вправо. Сама навигация по списку и его функции работают с помощью итераторов. Так нам не особо важна реализация списка. Используя итераторы, мы можем поменять внутреннее устройство списка и изменить только реализацию итераторов. При этом остальные функции по типу удаления, поиска и сортировки не сломаются. У списка есть следующие функции:

1. **Create** выделяет память под *head* и инициализирует пустой список.
2. **Insert** вставляет элемент перед переданным как аргумент итератором.
3. **Delete** удаляет элемент на месте, куда указывает итератор.
4. **Size** возвращает размер списка.
5. **isEmpty** проверяет список на пустоту.
6. **Write** записывает данные по итератору.
7. **First и Last** возврашают итераторы на первый и терминирующий элементы соответственно.

8. **Read** считывает элемент по итератору.

9. **nPos** возвращает итератор, указывающий на n элемент списка.

10. **Next и Prev** двигают переданный итератор вперед и назад по списку.

11. **Equals** проверяет равенство двух итераторов.

12. **move** двигает итератор на n элементов вправо.

13. **Destroy** очищает список.

1.2 Интерфейс и и выполнение задания

Это лабораторная с интерфейсом. Пользователь может с клавиатуры запросить печать списка или отсортировать его, найти нужный элемент или добавить новый. Добавление нового элемента осуществляется в конец таблицы. При этом таблица перестает считаться отсортированной. Есть два вида поиска: бинарный и линейный. Очевидно, что на неупорядоченном списке бинарный поиск работать не может. При вызове поиска программа обращается к флагу isSorted и запускает поиск. Если для неупорядоченного списка пользователь решил запустить бинарный поиск, программа посоветует ему сперва отсортировать таблицу. Сортировка и поиск в программе осуществляются по ключу. Как было сказано раннее, ключ в таблице является комплексным числом. Значит нужно написать компаратор, сравнивающий комплексные числа. Будем сравнивать ключи по модулю, но не будем извлекать квадрат из посчитанного значения. Ведь понятно, что из $a^2 > b^2$ следует $a > b$. Сам модуль в любом случае число неотрицательное и знак мнимой части при сравнении не учитывается.

Листинг 1: Компаратор для комплексных чисел

```
    bool cmp(complex* lhs, complex* rhs) {
    long long lval = lhs->real * lhs->real + lhs->img * lhs->img;
    long long rval = rhs->real * rhs->real + rhs->img * rhs->img;
    return lval > rval;
}
```

Листинг 2: Бинарный поиск

```
iterator binSearch(struct List* l, iterator lhs,
                        iterator rhs, complex value) {
    iterator first = First(l);
    while (distance(lhs, rhs) > 1) {
        int m = (distance(first, lhs) + distance(first, rhs)) / 2;
        iterator mid = nPos(l, m);
        complex key = Read(&mid).key;
        if (cmp2(&value, &key)) {
            lhs = mid;
        }
        else rhs = mid;
    }
    return lhs;
}
```

Это код бинарного поиска. Нам нужно найти какой-то элемент в массиве, пусть это будет элемент *value*. Сначала сортируем этот список. Затем находим его центральный элемент l[mid]. Если искомый элемент value > l[mid], нужно искать его в отрезке правее от центра. Иначе в отрезке левее центра, включая его. После перехода в эти меньшие отрезки, проделываем там то же самое, пока длина отрезка не станет равной единице. То есть, пока не придём к одному элементу. Если этот элемент равен искомому, то мы нашли ответ. Если нет, то искомый элемент не присутствовал в списке. Так как мы каждый раз делим массив на две части, сложность бинпоиска будет $O(logn)$, n - длина списка. Эта функция возвращает итератор и остается только где-то снаружи проверить, нашел ли бинпоиск нужный элемент.

2 Вывод

В ходе работы я составил программу на языке Си с использованием процедур и функций для сортировки таблицы методом простой вставки и двоичного поиска по ключу в таблице. Программа вводит значения элементов таблицы и проверять работу процедуры сортировки. Тестовые данные заранее помещены в текстовый файл, который передается программе как аргумент при запуске. В моем варианте используется комлексный ключ размером 16 байт. Получается, что действительной и мнимой частью должны быть числа типа long long по 8 байт каждое. Оценим сложности: в лучшем случае сортировка сработает за $O(n)$. Она пройдет по списку за линейную сложность и не найдет неотсортированных элементов, соответственно будет 0 перестановок в списке. В среднем же сортировка работает за квадратичную сложность. Бинарный поиск работает за логарифм. Каждый раз мы сравниваем значения и отбрасываем ровно половину оставшейся части списк. Линейный поиск на то и линейный, что его сложность составляет $O(n)$. Но понятно, что в лучшем случае он может сработать и за $O(1)$.