**Федеральное государственное бюджетное образовательное учреждение высшего образования**

|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | ***«Московский государственный технический университет***  ***имени Н.Э. Баумана***  ***(национальный технический университет)»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ ИУ

КАФЕДРА Компьютерные системы и сети

**Отчет  
по лабораторной работе № 1**

**Дисциплина: Технология разработки программных систем**

**Название лабораторной работы: Исследование структур и методов обработки данных**

**Вариант 22**

Студент гр. ИУ6-42 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Кузнецов И.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Хорунжина К. С.

Москва, 2018

**1. Цель лабораторной работы**

Целью данной работы является исследование структур данных, методы их обработки и оценки.

**2. Описание условий задачи**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Вариант | Задача | Структура | Поиск | Упорядочение | Корректировка |
| 22 | 5 | Кольц. список | последовательный | любой | удаление записи |

*Задача 5.* Даны элементы: 230, 150, 100, 85, 77, 33, 93, 200, 57, 137, 205.

**2.1 Требования к реализации**

**3. Описание основного варианта**

**3.1. Описание структуры данных**

Структура данных - кольцевой двунаправленный список.

Описание структуры данных на языке C++:

**struct** node {

**int** value;

node \*next, \*prev;

};

Целые числа хранятся в поле value, в полях next и prev хранятся указатели на следующий и предыдущий элементы списка соответственно.

**3.1.1. Расчет памяти, занимаемой списком**

Объем памяти, занимаемой списком *V=(lэл+lук)\*11+2\*lук.* Соответственно, найдем *lэл: lэл = 2 байта.*

*lук = 4 байта.*

Тогда, V = (2+2\*4)\*11 + 2\*4 = 118 *байт.*

**3.2.1. Описание и реализация метода поиска**

Согласно заданию используется метод последовательного поиска. В основе метода - последовательный просмотр записей списка с целью отыскания записей, для которых значение ключевого признака value совпадает со значением признака поиска key.

Реализация функции:

node \***search**(**int** key) {

node \*q = first;

**for** (**int** i = **0**; i < node\_count; ++i) {

**if** (q->value == key) {

**return** q;

}

q = q->next;

}

**return** nullptr;

}

**3.2.2.** **Оценка времени последовательного поиска по кольцевому списку**

Лучший случай (искомый элемент - первый в списке):

*T=t=+t=+tпр+1+tпр=2+2+2+1+2=9 (тактов)*

Худший случай (искомый элемент - последний в списке):

*T=t=+t=+tпр+1+11\*(tпр+1+tпр+2)=2+2+2+1+11\*(2+1+2+2)=84 (такта)*

*Среднее время поиска: T=(84+9)/2=47 (тактов)*

**3.3.1. Описание и реализация метода удаления записи**

Удаление записи по ключу key производится при помощи нахождения нужного элемента, а далее изменения указателей и высвобождения памяти.

Реализация метода:

**void** **delete\_node**(**int** key) {

**if** (node \*node\_to\_delete = search(key)) {

**if** (node\_to\_delete == first)

first = node\_to\_delete->next;

**if** (node\_to\_delete == last)

last = node\_to\_delete->prev;

node\_to\_delete->next->prev = node\_to\_delete->prev;

node\_to\_delete->prev->next = node\_to\_delete->next;

free(node\_to\_delete);

node\_count--;

}

}

**3.3.2.** **Оценка времени удаления элемента из кольцевого списка**

При удалении элемента изменяются значения указателей предыдущего и следующего элементов, указывающих на удаляемый. Также учитываются варианты, при который удаляемый элемент - крайний в списке. Для учета времени поиска воспользуемся ранее найденным средним временем поиска.

*T=Tпоиска+tпр+2\*(tпр+1/11\*t^)+t^+t^+1=47+2+2\*(2+1/11\*2)+2+2+1=58 (тактов)*

**3.4.1. Описание и реализация метода упорядочивания**

Для упорядочивания списка использовался метод упорядочивания обменом. Упорядочивание производилось по возрастанию числового поля.

Реализация метода:

**void** **sort\_list**(**void**) {

node \*i = first, \*j = nullptr;

**int** tmp = **0**;

**do** {

j = last;

**do** {

**if** (j->prev->value > j->value) {

tmp = j->value;

j->value = j->prev->value;

j->prev->value = tmp;

}

j = j->prev;

} **while** (j != i);

i = i->next;

} **while** (i != last);

}

**3.4.2. Асимптотическая оценка алгоритма**

Сложность сортировки оценивается как O(n2). Данный алгоритм следует применять к небольшим наборам данных. В данной задаче набор из 11 элементов - небольшой - благодаря чему использование данного метода упорядочивания оправдано простотой реализации.

**3.4.3. Оценка объёма буферных данных**

Для работы данного метода требуется одна буферная ячейка памяти для обмена числовых полей.

*V=2 байта.*

**3.4.4. Оценка времени упорядочивания**

*T=3\*t=+11\*(t=+65\*(tпр+1/2\*3\*t=+t^+tпр)+tпр+t^)=6+11\*(2+65\*(2+3+2+2)+2+2)=6507 (тактов)*

**4. Описание альтернативного варианта**

**4.1. Описание структуры данных**

В качестве структуры данных для альтернативного варианта выбран массив.

**4.1.2. Расчет памяти, занимаемой массивом**

V=lэл\*11=22 байта

**4.2.1. Описание и реализация метода поиска**

Метод поиска в массиве по ключу - последовательный.

Реализация метода:

**int** **search**(**int** \*array, **int** n, **int** key) {

**for** (**int** i = **0**; i < n; ++i)

**if** (array[i] == key)

**return** i;

**return** -**1**;}

**4.2.2. Оценка времени последовательного поиска по массиву**

Лучший случай (искомый элемент - первый в массиве): *T=t=+tпр+1+tпр=2+2+1+2=7 (тактов)*

Худший случай (искомый элемент - в конце массива): *T=t=+tпр+1+11\*(tпр+tпр+2)=2+2+1+11\*(2+2+2)=71 (такт)*

Среднее время поиска: *Tпоиска ср.=(71+7)/2=39 (тактов)*

**4.3.1. Описание и реализация метода удаления элемента массива**

При удалении элемента массива все последующие сдвигаются влево, а количество элементов массива уменьшается на 1. В методе также используется функция поиска для определения существования элемента.

Реализация метода:

**int** **delete\_elem**(**int** \*array, **int** n, **int** key) {

**int** i = search(array, n, key);

**if** (i != -**1**) {

**for** (**int** j = i; j < n - **1**; j++)

array[j] = array[j + **1**];

**return** n - **1**;

} **else**

**return** n;

}

**4.3.2.** **Оценка времени удаления элемента из массива**

Среднее время работы метода удаления:

*T=Tпоиска ср.+5+3,5\*11=83 (такта)*

**4.4.1. Описание и реализация метода упорядочивания**

В качестве метода упорядочивания массива используется метод вставки. Упорядочивание производилось по возрастанию числового поля.

Реализация метода:

**void** **sort**(**int** \*array, **int** n) {

**int** i = **0**, j = **0**, key = **0**;

**for** (i = **1**; i < n; ++i) {

key = array[i];

**for** (j = i; (j >= **0**) && (array[j - **1**] > key); --j)

array[j] = array[j - **1**];

array[j] = key;

}

}

**4.4.2. Асимптотическая оценка алгоритма**

Сложность сортировки оценивается как O(n2).

**4.4.3. Оценка объёма буферных данных**

Алгоритм требует одну буферную ячейку памяти для хранения числового значения текущего сравниваемого элемента.

*V=2 байта.*

**4.4.4. Оценка времени упорядочивания**

Худший случай: *T=3\*t=+tуст+tпр+1+11\*(t=+tуст+tпр+1+65\*(t=+tпр+2)+t=+tпр+2)=11+11\*(7+65\*6+6)=4444 (такта)*

**5. Заключение**

**5.1. Таблица сравнения**

В Таблице 1 приведено сравнение методов поиска, упорядочивания и удаления при использовании основного и альтернативного вариантов.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Вариант** | **Структура** | **Поиск** | **Упорядочивание** | **Удаление элемента** |
| **Основной** | Двусвязный кольцевой список  *V=118 байт* | Последовательный по ключу  *T=47 тактов* | Обменом  *V=2 байта*  *T=6507 тактов* | *T=58 тактов* |
| **Альтернативный** | Массив  *V=22 байта* | Последовательный по ключу  *T=39 тактов* | Метод вставки  *V=2 байта*  *T=4444 тактов* | *T=83 такта* |

*Таблица 1. Сравнение основного и альтернативного способов*

**5.2. Вывод**

Согласно заданию разработана структура данных “кольцевой список”, организован последовательный поиск, и сортировка методом обмена. Для усовершенствования алгоритма разработан альтернативный вариант. Структура данных изменена на массив, упорядочивание - на метод вставки. Альтернативный вариант позволяет использовать меньше памяти для хранения данных, а также быстрее производится сортировка и последовательный поиск элемента по ключу.