*Федеральное государственное бюджетное образовательное учреждение* *высшего образования*

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

ФАКУЛЬТЕТ \_\_\_\_\_ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА \_\_\_\_\_\_\_ Компьютерные Системы и сети (ИУ6)\_\_\_\_\_\_\_\_\_\_\_

**Отчет**

**по лабораторной работе № 1**

**Вариант 24**

**Дисциплина: технология разработки программных систем**

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

Студент гр. ИУ6-44  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шумаков А.А.**

(Подпись, дата) (И.О. Фамилия)

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

(Подпись, дата) (И.О. Фамилия)

Москва, 2018

**Цель работы**: изучение форматов машинных команд, команд целочисленной арифметики ассемблера и программирование целочисленных вычислений.

**1. Цель лабораторной̆ работы**

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

**2. Описание задания**

Даны элементы: 130, 50, 120, 185, 27, 43, 913, 210, 5, 17, 245.

Основным вариантом реализации структуры данных, поиска, упорядочивания и корректировки являются:

* Структура данных — линейный односвязный список
* Поиск — гнездовой
* Упорядочение — любой метод
* Корректировка — удаление записи

**3. Выполнение**

Данной в задании структурой данных является линейный односвязный список. Список был сформирован по типу стека. Стек *–* этопоследовательность элементов, добавление и удаление элементов в которую осуществляется с одного конца.

Каждый элемент стека содержит следующие поля:

* Целое число
* Указатель на следующий элемент стека

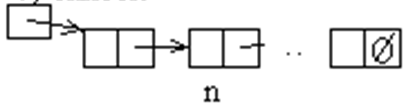


Рисунок 1– Графическая схема списка

На рисунке 2 представлена структура элемента списка chislo.

|  |  |
| --- | --- |
| chislo | |
| num | Число |
| \*next | Указатель на следующий элемент списка |

Рисунок 2 – Структура элемента списка chislo

**Реализация структуры на языке C++**

Следующий код представляет реализацию структуры списка и создание его первого элемента.

struct chislo

{

short int num;

chislo \*next;

};

int main()

{

chislo \*number = new chislo; //Память под элемент списка

chislo \*h = number; //Указатель на начало списка

…

Тип short int для поля num был выбран, исходя из условия задачи.

**Оценка объема занимаемой памяти списком**

Так как на тип short int отводится 2 байта памяти, а на указатель следующего элемента списка 4 байта, то размер одного элемента списка будет равен:

объем памяти отводимый под поле short int num;

объем памяти отводимый под указатель на следующий элемент списка chislo \*next

Но, для хранения элемента списка компьютер отводит память, равную 8 байтам, что доказывает следующий выполненный фрагмент программы (см. рис.3).

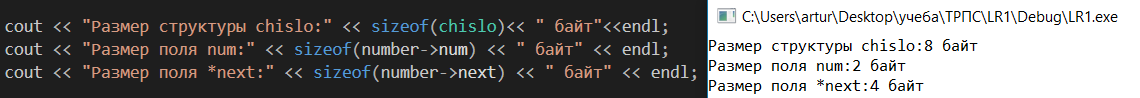


Рисунок 3 – Определение размера списка

Так как по условию задачи всего дано 11 элементов, то на хранение всего списка будет выделяться память

объем памяти отводимый под указатель на начало списка.

Следовательно, прямо зависит от количества элементов. Отметим, что при создании нового элемента списка будет постоянно отводиться память под указатель на следующий элемент списка, а это значит, что, если нам известно количество элементов, то мы будем выделять больше памяти для хранения списка по сравнению с памятью отводимой под хранение массива.

**Описание метода поиска**

Задан гнездовой метод поиска.

**Описание:** Способ заключается в том, что множество элементов разбивается на группы по определенному принципу. Затем эти группы объединяют в отдельные гнезда с одинаковой структурой. Далее строят дополнительную структуру, в которой все элементы связываются. Каждый элемент содержит ключевой признак и адрес только одного гнезда.

Гнездовой способ был реализован с помощью списка гнезд. Для данной задачи ключевым признаком является количество цифр в числе. Все числа исходного списка делятся на группы по принципу количества цифр в числе. Для доступа к определенному гнезду организован список гнезд, в котором каждый элемент содержит три поля: ключевой признак гнезда (количество цифр в числе), адрес гнезда и адрес следующего элемента списка гнезд. Для хранения самих гнезд используются массивы.

**Реализация метода поиска на языке C++**

void search(chislo \*c,short int s,short const int size)

{

//Для работы с гнездами

gnezdo \*heast = new gnezdo;

gnezdo \*begin = heast;

gnezdo \*hcurrent = heast;

short const int kheast = 3;

cout << "Размер структуры gnezdo:" << sizeof(gnezdo) << " байт" << endl;

cout << "Размер поля kolnum:" << sizeof(heast->kolnum) << " байт" << endl;

cout << "Размер поля \*point:" << sizeof(heast->point) << " байт" << endl;

cout << "Размер поля \*next:" << sizeof(heast->next) << " байт" << endl;

//Для определения количества элементов

//удовлетворяющих признаку = кол-во цифр в числе

short int kone = 0;

short int ktwo = 0;

short int kthree = 0;

short int kol = 0;

//Для работы со списком

chislo \*head = c;

chislo \*current = c;

//Цикл для того чтобы определить размерность массивов

for (int i = 0; i < size; i++)

{

if (current->num / 10 == 0)

{

kone++;

}

else

{

if (current->num / 100 == 0)

{

ktwo++;

}

else kthree++;

}

current = current->next;

}

//Объявляем открытые массивы

short int \*A = new short int[kone];

short int \*B = new short int[ktwo];

short int \*C = new short int[kthree];

//Соотвествующие индексы массивов

short int x = 0; //A[x]

short int y = 0; //B[y]

short int z = 0; //C[z]

//Заполнение массивов по признаку

current = head;

for (int i = 0; i < size; i++)

{

if (current->num / 10 == 0)

{

A[x] = current->num;

x++;

}

else

{

if (current->num / 100 == 0)

{

B[y] = current->num;

y++;

}

else

{

C[z] = current->num;

z++;

}

}

current = current->next;

}

//Создание гнезд

current = head;

for (int i = 0; i < kheast; i++)

{

heast->kolnum = i+1;

if (i != kheast-1)

{

heast = new gnezdo;

heast->next = NULL;

hcurrent->next = heast;

hcurrent = heast;

}

}

//Выделение признака в числе

short int priznak = 0;

short int searcher = s;

while (searcher != 0)

{

searcher /= 10;

priznak++;

}

heast = begin;

if (priznak == 1)

{

heast = begin;

heast->point = A;

for (short int i = 0; i < x; i++)

{

if (heast->point[i] == s)

{

cout << "Найден заданный элемент в гнезде " << heast->kolnum << " с индексом " << i << endl;

delete[] A;

delete[] B;

delete[] C;

return;

}

}

}

if (priznak == 2)

{

heast = heast->next;

heast->point = B;

for (short int i = 0; i < y; i++)

{

if (heast->point[i] == s)

{

cout << "Найден заданный элемент в гнезде " << heast->kolnum << " с индексом " << i << endl;

delete[] A;

delete[] B;

delete[] C;

return;

}

}

}

if (priznak == 3)

{

heast = heast->next;

heast = heast->next;

heast->point = C;

for (short int i = 0; i < z; i++)

{

if (heast->point[i] == s)

{

cout << "Найден заданный элемент в гнезде " << heast->kolnum << " с индексом " << i << endl;

delete[] A;

delete[] B;

delete[] C;

return;

}

}

}

cout << "Заданного элемента нет в исходной структуре данных" << endl;

return;

}

Так как количество цифр заданного числа известно, то мы можем сразу же определить с помощью условия в каком гнезде стоит осуществлять поиск. Так как гнезда представляют собой массивы с N числом записей, а поиск в гнезде осуществляется последовательно, то среднее число сравнений для реализации данного метода будет вычисляться следующим образом:.

**Оценка по времени:**

Оценку по времени начнем производить не с момента создания списка гнезд и его заполнения, а с момента самого начала поиска, т.е. с определения принадлежности элемента к конкретному гнезду.

время проверки первого условия if (priznak ==1).

время проверки второго условия if (priznak ==2).

время проверки третьего условия if (priznak ==3).

время выполнения первого условия.

время выполнения второго условия.

время выполнения третьего условия.

Определим время выполнения каждого условия:

Для данной задачи то есть у нас есть всего одно число, состоящее из одной цифры. Тогда

Аналогично рассчитываются и . Для берем , а для .

В формуле для мы использовали слагаемое 6, так как в данном случаи выполняется на одну команду присваивания больше, чем в предыдущих случаях.

Таким образом,

Отметим также, что при гнездовом поиске будет выделяться дополнительная память для хранения списка гнезд и самих гнезд. Определим данный объем памяти.

Для начала приведем описание структуры списка гнезд.

struct gnezdo {

short int kolnum;

short int \*point;

gnezdo \*next;

};

Выполним показанный на рисунке 4 фрагмент программы, чтобы понять сколько памяти тратится на хранение полей элемента списка гнезд.

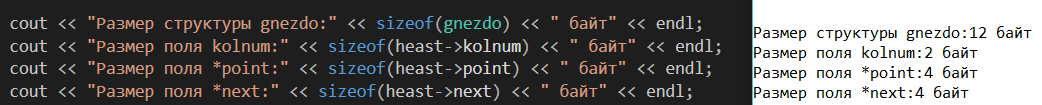


Рисунок 4-Определение размера списка гнезд

Плюс необходимо отвести память под сами гнезда, представленные массивами типа short int.

Итого получаем

**Описание метода упорядочивания**

Метод упорядочивания – метод обмена.

**Описание:** Метод, базирующийся на поиске минимального (максимального) элемента, заключается в следующем. Определяется минимальный (максимальный) элемент всего набора и осуществляется обмен с первым элементом. Затем определяется наименьший (наибольший) элемент из оставшегося набора и меняется местами со вторым и т. д. Количество сравнений по данному методу определяется по формуле: C = N(N-1)=11\*10=110.

**Реализация метода упорядочивания на языке C++**

void sort(chislo \*c)

{

bool IsSorted = false;

short int help = 0;

chislo \*current = c;

while (IsSorted == false) {

IsSorted = true;

while (c->next != NULL)

{

c = c->next;

if ((current->num) < (c->num))

{

IsSorted = false;

help = current->num;

current->num = c->num;

c->num = help;

}

}

current = current->next;

c = current;

}

}

**Оценка по времени**

Количество итераций во внешнем цикле while

Количество итераций во внутреннем цикле while

Таким образом, при увеличении количества элементов такты будут значительно увеличиваться, так как мы использовали вложенный цикл в цикле.

**Удаление записи**

Реализуем удаление элемента из списка и вывод сообщения, если все элементы удалены.

**Реализация удаления записи на языке C++**

void del(chislo \*c)

{

if (c != NULL)

{

chislo \*current = c;

cout << "Элемент " << c->num << " Удален" << endl;

c = c->next;

delete current;

}

else

cout << "Все элементы удалены!" << endl;

}

**Оценка по времени**

**4.Альтернативный вариант**

Заменим списковую структуру данных на массив, состоящий из 11 элементов.

**Реализация структуры на языке C++**

const short int size = 11; //Количество элементов в структуре

short int massive[size] = {130, 50, 120, 185, 27, 43, 913, 210, 5, 17, 245};

**Оценка объема занимаемой памяти массивом**

Так как на тип short int отводится 2 байта памяти, то массив, состоящий из 11 элементов, будет занимать:

объем памяти отводимый под поле short int num;

количество элементов массива.

Как видим массив занимает меньше места в памяти по сравнению со списком. Отметим, что если нам известно изначально количество элементов, то массив будет более выигрышным вариантом по памяти по сравнению со списком, так как теперь не отводится память под указатель на следующий элемент списка. Но если количество элементов изначально неизвестно, то использовать список более рациональный подход.

**Описание альтернативного варианта поиска**

Выполним последовательный поиск нужной записи в массиве. Количество сравнений при последовательном поиске С = (N+1)/2 = (11+1)/2=6

**Реализация последовательного поиска в массиве на языке C++**

void Alt\_search(short int \*a, short int s, short const int size) {

short int i = 0;

while (i < size) {

if (a[i] == s) {

cout << "Элемент " << a[i] << " найден, его индекс в массиве " << i << endl;

return;

}

i++;

}

cout << "Заданного элемента нет в массиве" << endl;

return;

}

**Оценка по времени**

Как видим получили улучшение по времени по сравнению с гнездовым поиском. Заметим, что мы получили значительное время, так как поиск осуществлялся среди малого количество элементов массива. При больших объемах данных гнездовой способ реализации будет выглядеть предпочтительней по времени.

**Описание альтернативного варианта упорядочивания**

Метод упорядочивания – метод вставки.

**Описание:** Элемент массива ai (начиная со второго) сравнивается последовательно с предшествующими aj, где j = i-1, i-2, … до тех пока не будет найден элемент с меньшим значением, чем ai. Пусть этот элемент с номером j, где j < i. Тогда все элементы с номерами j+1, …i-1 сдвигаются на одну позицию, а i-й элемент ставится на место (j+1)-го элемента. Если все впереди стоящие элементы больше *i*-го, то они сдвигаются на одну позицию, а *i*-й элемент ставится на первое место. Количество сравнений определяется по формуле: C = N(N-1)/4 = 11\*(11-1)/4 =27.5 .

**Реализация метода упорядочивания на языке C++**

void Alt\_sort(short int \*a,short int size) {

short int help,prev = 0;

for (int i = 1; i < size; i++) {

help = a[i];

prev = i - 1;

while (prev >= 0 && a[prev] > help) {

a[prev + 1] = a[prev];

a[prev] = help;

prev--;

}

}

}

**Оценка по времени**

*n=11*

Получили улучшение по сравнению с методом обмена, используемого в основном варианте. Выигрыш по времени объясняется тем, что количество сравнений при выборе метода вставками в четыре раза меньше, чем сравнений в методе обмена.

**Альтернативный вариант удаления**

Метод удаления – удаление сдвигами.

**Реализация метода удаления сдвигами на языке C++**

void Alt\_del(short int \*a, short int size) {

int Index\_del = 0;

cout << "Введите индекс от 0 до "<<size-1<< endl;

cin >> Index\_del;

for (int i = Index\_del; i < size - 1; i++)

a[i] = a[i + 1];

a[size - 1] = 0;

cout << "Альтернативное удаление" << endl;

cout << "Был удален элемент " << a[Index\_del] << " с индексом " << Index\_del << endl;

}

**Оценка по времени**

Проигрыш по времени по сравнению с удалением записи в списке.

**5.Вывод**

В таблице 1 обобщим все полученные результаты

Таблица 1

|  |  |  |
| --- | --- | --- |
|  | ***Основной вариант*** | ***Альтернативный вариант*** |
| **Структура** | ***Список***  Объем памяти = 92 байта | ***Массив***  Объем памяти = 22 байта. |
| **Метод поиска** | Гнездовой  С=6  Т=144  Vдоп.=62 байта | Последовательный  С=6  Т=9…79 |
| **Метод сортировки** | Обмена  С=110  Т= 12…1502 | Вставками  С=27,5  Т=205…1195 |
| **Метод корректировки** | Удаление записи  Т=7.5 | Удаление записи сдвигами  Т=40 |

Таким образом, на небольших объемах данных (11 элементов) структура с массивом оказывается выгоднее по большинству параметров. Это и значительно меньшая память, отводимая под хранение структуры, и более короткое время поиска и сортировки за счет группировки элементов и возможности обращения к ним по индексу. Но при значительном увеличении числа элементов список выиграет по скорости удаления элементов. Удаление в структуре со списком будет иметь большую скорость работы на больших объемах, так как предполагает в основном переадресацию указателей, а не перенос данных. Этот аспект может оказаться важным, если хранить придется не числа, а иные, более объемные структуры.