|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-3 «Управление и моделирование систем»

**ОТЧЕТ   
о выполнении лабораторной работы №1**

**«Реализация сортировки линейных структур данных»**

**по дисциплине   
«Программная реализация нелинейных структур»**

**Вариант № \_17\_\_**

Выполнил: студент 1 курса

группы БСБО-01-22

шифр 22Б0617

Буторин Кирилл Алексеевич  
*(фио студента)*

Проверил:

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2020 г.

**Задание на лабораторную работу № 1.**

В рамках лабораторной работы №1 требуется программно реализовать (с помощью указателей (однонаправленных/двунаправленный динамический линенйый связанный список, массива или используя стандартный контейнер библиотеки STL “stack” или «queue» - по варианту) абстрактный тип данных (АТД) в соответствии с заданием (стек, дек, очередь с одной головой, очередь с головой и хвостом).

Абстрактный тип данных должен позволять осуществлять только операции, присущие типу линейного связанного списка:

* получить значение первого элемента (на выходе),
* добавить элемент (на вход),
* удалить элемент из списка (на выходе),
* проверить – список пуст,
* обнулить (проинициализировать) список (конструктур, при необходимости).
* деструктор (при необходимости)

Используя разработанный АТД и указанный набор операций, необходимо реализовать заданный алгоритм сортировки последовательности элементов заданного типа, при этом следует учитывать, что разрешен доступ (чтение/извлечение) только к элементу на выходе.

На основе исходного текста программы получить аналитическую оценку трудоемкости работы алгоритма сортировки, используя О-символику для каждого реализованного метода АТД и сортировки в целом.

**Вариант № 17.**

**Реализация связи элементов линейного списка: Указатели**

**Способ организации линейного связанный список: Дэк**

**Алгоритм сортировки: Бинарная вставка**

**Теория о сортировках.**

**Сортировка бинарными вставками.**

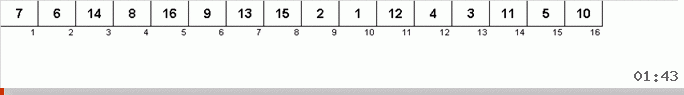
Сортировка по двоичной вставке — это алгоритм сортировки, который похож на сортировку по вставке, но вместо использования линейного поиска для нахождения местоположения, в которое должен быть вставлен элемент, мы используем двоичный поиск.

В режиме сортировки с бинарной вставкой мы разделяем одни и те же элементы на два подмассива – фильтрованный и нефильтрованный. Первый элемент из тех же элементов находится в организованном подмассиве, а все остальные элементы являются незапланированными.

Затем мы выполняем итерацию от второго элемента к последнему. При повторении i-го мы делаем текущий объект нашим “ключом”. Этот ключ является функцией, которую мы должны добавить в наш существующий список ниже.

Чтобы сделать это, мы сначала используем бинарный поиск в отсортированном подмассиве ниже, чтобы найти местоположение элемента, размер которого больше нашего ключа. Давайте назовем эту позицию “pos”. Затем мы сдвигаем все элементы вправо с pos на 1 и создаем массив [pos] = key.

Мы можем отметить, что при каждом i-м умножении левая часть массива till (i – 1) уже отсортирована.

й

**Листинг программы с расчетами.**

namespace LinearStructures.Structures;

// Линейная структура "Дэк"

public class Deque

{

public Item? head;

public Item? tail;

long len;

public bool isEmpty() //Провера на пустоту (1)

{

Program.N\_OP++;

return head == null; //1

}

public void Print() //Вывод элементов дэка

{

Item? current = head;

while (current != null)

{

Console.Write($"{current.value.ToString()} "); //Проходим до конца дэка то есть до нулевого элемена

current = current.next;

}

Console.WriteLine();

}

public void PushBack(Item newitem) //Добавить в конец =1 + (2+2) + (1+2) = 8

{

Program.N\_OP++;

if (head == null) //1

{

Program.N\_OP++;

head = newitem; //1

}

else

{ //Если дэк пустой то добавляем элемент который будет являться и головой и хвостом

Program.N\_OP += 4;

tail.next = newitem; //2 //Иначе новый элемент теперь следующий для прежнего хвоста а хвост теперь предыдущий для нового элемента

newitem.previous = tail; //2

}

Program.N\_OP += 3;

tail = newitem; //1

len++; //2

}

public void PushFront(Item newitem) //Добавить в начало =(2+2+1+1) + 2 + 2 = 10

{

Program.N\_OP += 6;

Item? temp = head; //2

newitem.next = temp; //2 //Создаем промежуточную переменную temp которой присваиваем значение головы для обмена значениями между головой и следующего для головы элемента

head = newitem; //1

if (head == null) //1 //Если дэк пустой то добавляем элемент который теперь является и головой и хвостом

{

Program.N\_OP++;

tail = head; //1

}

else

{ //Иначе новый элемент это предыдущий для temp которая является бывшей головой

Program.N\_OP += 2;

temp.previous = newitem; //2

}

Program.N\_OP+=2;

len++; //2

}

public Item PopBack() //Удаление с конца =1 + 2 + 1 + (2+2) + 2 = 10

{

Program.N\_OP++;

if (head == null) //1

throw new InvalidOperationException("Deque is empty"); //Ошибка если дэк пустой

Program.N\_OP+=3;

Item? output = tail; //2

if(len == 1) //1 //Если в дэке один элемент то присваиваем output значение этого элемента и указываем что голова и хвост теперь null

{

Program.N\_OP += 2;

head = tail = null; //2

}

else //Иначе хвостом теперь будет предыдущий для хвоста элемент

{

Program.N\_OP += 4;

tail = tail.previous; //2

tail.next = null; //2

}

Program.N\_OP+=2;

len--; //2

return output;

}

public Item PopFront() //Удаление с начала =3 + 1 + 2 + 2 + 2 = 10

{

Program.N\_OP += 3;

if (this.isEmpty()) //3

throw new InvalidOperationException("Deque is empty"); //Ошибка если дэк пустой

Program.N\_OP += 7;

Item? output = head; //1

head = head.next; //2 //Присваиваем для головы новое значение(следующий в дэке элемент)

output.next = null; //2

len--;//2

return output;

}

public int Get(int index) // (1+2+1+3)+4+24\*(n-1)+2+4+22\*(n-1) = 46n - 29

//Чтение элемента по индексу

{

Program.N\_OP += 3;

int result = 0; //1

if (isEmpty()) //2

throw new InvalidOperationException("Deque is empty"); //Ошибка если дэк пустой

Program.N\_OP++;

if (index > len) //1

{

throw new Exception("Out of range"); //Ошибка при выходе за границы дэка

}

Program.N\_OP += 3;

if (index > (len / 2) - 1) //3 //Если индекс находится правее середины то быстрее будет начать перебор дэка с конца

{

Program.N\_OP += 4;

for (int i = 0; i < (len - index - 1); i++)//4

{

PushFront(PopBack());//2 // Чтобы дойти до нужного элемента с конца мы удаляем элемент с конца и добавляем его в начало

Program.N\_OP += 6;

}

result = tail.value;//2 //Дойдя до нужного элемента запоминаем его

Program.N\_OP += 2;

Program.N\_OP += 4;

for (int i = 0; i < (len - index - 1); i++)//4

{

PushBack(PopFront()); //2 //Теперь возвращаем дэк в исходное состояние удаляя элементы из начала и добавляя их в конец пока не придем к исходному состоянию дэка

Program.N\_OP += 6;

}

return result;

}

else //Если же элемент левее середины то делаем все наоборот. Сначала удаляем элементы из начала и добавляем их в конец пока не дойдем до нужного элемента

{ //После чего возвращаем дэк в исходное состояние удаляя элементы с конца и добавляя их в начало

Program.N\_OP += 2;

for (int i = 0; i < index; i++) // 2

{

PushBack(PopFront()); //2

Program.N\_OP += 4;

}

result = head.value;

Program.N\_OP += 2;

Program.N\_OP += 2;

for (int i = 0; i < index; i++)//2

{

PushFront(PopBack());//2

Program.N\_OP += 4;

}

return result;

}

}

public void Set(int index, int value) //(2+1+3)+4+24\*(n-1)+2+4+22\*(n-1) = 46n - 30

// Изменение элемента по индексу

{ //Аналогично Get только с изменением значения

Program.N\_OP += 2;

if (isEmpty()) //2

throw new InvalidOperationException("Deque is empty");

Program.N\_OP++;

if (index > len) //1

{

throw new Exception("Out of range");

}

Program.N\_OP += 3;

if (index > (len / 2) - 1) //3

{

Program.N\_OP += 4;

for (int i = 0; i < (len - index - 1); i++) //4

{

PushFront(PopBack());//2

Program.N\_OP += 6;

}

tail.value = value;//2

Program.N\_OP += 2;

Program.N\_OP += 4;

for (int i = 0; i < (len - index - 1); i++) //4

{

PushBack(PopFront());//2

Program.N\_OP += 6;

}

return;

}

else

{

Program.N\_OP += 2;

for (int i = 0; i < index; i++) //2

{

PushBack(PopFront()); //2

Program.N\_OP += 4;

}

head.value = value;//2

Program.N\_OP += 2;

Program.N\_OP += 2;

for (int i = 0; i < index; i++) //2

{

PushFront(PopBack()); //2

Program.N\_OP += 4;

}

}

}

public int this[int index]

//Перегрузка оператора индексирования в []

{

get//46n-27

{

Program.N\_OP += 2;

return Get(index); //2 + 46n - 29

}

Set //46n-27

{

Program.N\_OP += 3;

Set(index, value); //3 + 46n - 30

}

}

public void BinaryInsertSort() //92\*n\*n\*n – 11\*n\*n + 46\*n\*n\*log2(n) – 22\*n\*log2(n) – 20\*n + 2

//Бинарная вставка

Program.N\_OP += 2;

for (int i = 1; i < len; i++) //2

{

Program.N\_OP += 2;

int key = this[i]; //2 //1+46n-29=46n-28

// Ключ (сортируемый элемент);

Program.N\_OP += 1;

int left = 0; //1

// Левая граница отсортированной части;

Program.N\_OP += 2;

int right = i - 1; //2

// Правая граница отсортированной части.

// Бинарный поиск в отсортированной части дэка.

// 2

Program.N\_OP += 1;

while (left <= right)

{

Program.N\_OP += 3;

int middle = (left + right) / 2; //3

//2

Program.N\_OP += 2;

if (this[middle] > key) //2 + 2 + 46n – 29 = 46n - 25

{

Program.N\_OP += 2;

right = middle - 1; //2

}

else

{

Program.N\_OP += 2;

left = middle + 1; //2

}

}

// Сдвиг необходимых элементов вправо на 1.

Program.N\_OP += 3;

for (int j = i - 1; j >= left; j--) //3

{

Program.N\_OP += 6;

this[j + 1] = this[j]; //4

}

Program.N\_OP += 2;

this[left] = key; //2 //46n-30 + 1 = 46n - 29

// Вставка ключа на его место в отсортированной части.

}

}

}

namespace LinearStructures.Structures;

public class Item

{

public int value; // Значение элемента;

public Item? next; // Указатель на следующий элемент.

public Item? previous; // Указатель на предыдущий элемент.

public Item(int value = 0, Item? next = null, Item? previous = null)

{

this.value = value;

this.next = next;

this.previous = previous;

}

}

using LinearStructures.Structures;

using System.Diagnostics;

internal class Program

{

public static long N\_OP = 0;

static void Main(string[] args)

{

var x = new Stopwatch();

Deque deque = new Deque();

Random rnd = new Random();

for (int n = 1; n <= 10; n++)

{

for (int i = 0; i < n\*100; i++)

{

deque.PushBack(new Item(value: rnd.Next(0, 100))); //Заполнение дэка

}

N\_OP = 0;

x.Restart();

deque.BinaryInsertSort();

x.Stop();

Console.WriteLine($"Номер сортировки: {n} Количество отсортированных элементов: {n \* 100} Время сортировки (ms): {x.ElapsedMilliseconds} Количество операций (N\_Op): {N\_OP}");

}

}

}

F(n) = 92\*n\*n\*n – 11\*n\*n + 46\*n\*n\*log2(n) – 22\*n\*log2(n) – 20\*n + 2

O(F(n)) = n\*n\*n

**Таблица результата экспериментов и графики зависимостей**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 100 | 94929559 | 1000000 | 0,014 | 6790502 |
| 200 | 749587064 | 8000000 | 0,185 | 168508797 |
| 300 | 2517017001 | 27000000 | 1,189 | 1153196252 |
| 400 | 5949774717 | 64000000 | 3,599 | 4529086588 |
| 500 | 11600247897 | 125000000 | 10,019 | 12877529099 |
| 600 | 20020735419 | 216000000 | 23,120 | 29899702857 |
| 700 | 31763480751 | 343000000 | 52,376 | 64157715977 |
| 800 | 47380689396 | 512000000 | 97,339 | 121462996951 |
| 900 | 67424539176 | 729000000 | 173,942 | 214645615943 |
| 1000 | 92447186831 | 1000000000 | 285,585 | 356236168692 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 6780682811,691 | 71428571,428 | 13,979 | 0,147 |
| 4051821969,848 | 43243243,243 | 4,448 | 0,047 |
| 2116919260,870 | 22708158,116 | 2,182 | 0,023 |
| 1653174414,454 | 17782717,421 | 1,313 | 0,014 |
| 1157824922,412 | 12476295,039 | 0,900 | 0,009 |
| 865948763,802 | 9342560,553 | 0,669 | 0,007 |
| 606451060,634 | 6548800,977 | 0,495 | 0,005 |
| 486759566,015 | 5259967,741 | 0,390 | 0,004 |
| 387626560,441 | 4191052,189 | 0,314 | 0,003 |
| 323711633,425 | 3501584,466 | 0,259 | 0,002 |

**Скриншот работы программы:**

**Изображение выглядит как текст, снимок экрана, Шрифт, черно-белый

Автоматически созданное описание**

**Выводы.**

По результатам экспериментов было установлено, что графики C1, C2, C3, C4 от N имеют линейную зависимость от количества элементов.

**Литература:**

1. Структуры данных и алгоритмы. Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. – М.: Издательский дом «Вильямс», 2000

2. Д. Кнут. Искусство программирования для ЭВМ.