|  |  |
| --- | --- |
|  | /storage/emulated/0/.polarisOffice5/polarisTemp/image1.png |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждениевысшего профессионального образования «Московский технологический университет»  МИРЭА | | |

Институт Информационных Технологий

Кафедра Корпоративных Информационных Систем

**ОТЧЕТ**

по Лабораторной Работе № 6

на тему

«Структуры данных – связный список, стек, очередь»

по дисциплине

«Процедурное Программирование»

Выполнил студент группы ИКБО-07-17 Акжигитов Р. Р.

Принял асс. каф. КИС Алдобаева В. Н.

Выполнено «21» ноября 2017 г.

Зачтено «\_\_\_» \_\_\_\_\_\_\_\_\_ 2017 г.

Москва, 2017

**Задание**

Создать приложение, реализующее основную функциональность списков, стеков, очередей на основе связных списков и сравнение производительности с теми же функциональными элементами, основанными на динамических массивах, в соответствии с индивидуальным заданием.

**Вариант 1**

1. Очередь на основе двусвязного списка. Операции помещения и удаления элемента.
2. Стек на основе двусвязного списка. Операции протолкнуть и вытолкнуть.
3. Двусвязный список. Операции поиска элемента, добавления элемента в конец.

**Теоретическая часть**

Список – одна из составных структур данных. Список – динамический последовательный набор однотипных элементов, появились именно для хранения динамических наборов, это понятие тесно связано с динамической памятью и указателями.

**Различия между списками и динамическими массивами:**

1. Массивы записаны в памяти последовательно. Элементы списка могут располагаться в произвольном порядке, со свободными ячейками памяти между ними.
2. В массиве можно легко обратиться к любому элементу по индексу. Список – структура строго последовательного доступа.
3. В массиве отсутствуют ссылки элементов друга на друга, в списке они есть и занимают заметное количество памяти (накладные расходы менеджера памяти).
4. При добавлении или удалении элемента массив надо переформировать целиком. В списке надо изменить всего лишь пару указателей.

Массивы рационально применять для сравнительно небольших, не часто изменяющихся наборов данных малого размера. Списки – для больших элементов, большого набора и подверженного частым изменениям. Списки могут применяться при работе с данными строго последовательного доступа.

Для получения времени работы функции можно использовать функцию **GetTickCount,** она возвращает кол-во миллисекунд с момента загрузки системы, удобна для измерения небольших непрерывных интервалов времени. Для очень маленьких и очень больших она неприменима. Точность составляет – от 16 мс, до 49.7 суток.

Для реализации структур на основе массивов и структур на основе связных списков будем использовать стандартные шаблоны из **System.Collections.Generic**.

List<SmallData> SmallList = new List<SmallData>();

LinkedList<LargeData> LargeLinkedList = new LinkedList<LargeData>();

**О́чередь** — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, First In — First Out). Добавление элемента (принято обозначать словом **enqueue** — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом **dequeue** — убрать из очереди), при этом выбранный элемент из очереди удаляется.

**Стек** (англ. stack — стопка; читается стэк) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

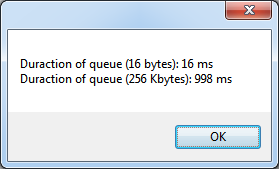
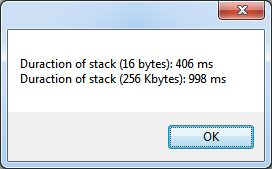
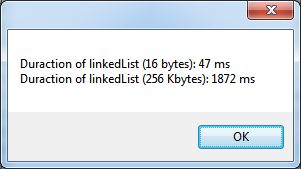
Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

В цифровом вычислительном комплексе стек называется магазином — по аналогии с магазином в огнестрельном оружии (стрельба начнётся с патрона, заряженного последним).

**Алгоритм решения задачи**

1. Создадим простейшую форму с одной кнопкой для исполнения всех измерений.
2. Все измерения будем проводить в цикле 25 000 раз и с максимальным объектом в 256 кб данных, чтобы не возникало конфликтов с системой и запускалось на любой компьютере.
3. Создаем в небезопасном коде структуру данных, хранящую либо 16 бит информации, либо 256 кб.
4. Используя встроенные типы Стек, Очередь и Связный список и их методы для добавления поиска и удаления элементов, провожу замеры времени.
5. Вывод на экран времени, затраченного на обработку большой структуры и маленькой.

**Тестирование**

**Заключение**

В заключении, хочется сказать, что данные знания очень полезны в практическом ключе. Так как часто приходится выбирать, в какой структуре данных хранить свои однотипные элементы (например, в списке или в массиве).

**Исходный код**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace laba7

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

unsafe struct SmallData

{

public fixed byte a[16];

}

unsafe struct LargeData

{

public fixed byte a[25600];

}

private void button1\_Click(object sender, EventArgs e)

{

int msTest1, msTest2;

Queue<SmallData> SmallQueue = new Queue<SmallData>();

int time = Environment.TickCount;

for (var i = 0; i < 25000; i++)

{

SmallData data = new SmallData();

SmallQueue.Enqueue(data);

}

for (var i = 0; i < 25000; i++)

{

SmallQueue.Dequeue();

}

msTest1 = Environment.TickCount - time;

//MessageBox.Show("Time = " + ((Environment.TickCount - time)).ToString());

Queue<LargeData> LargeQueue = new Queue<LargeData>();

time = Environment.TickCount;

for (var i = 0; i < 25000; i++)

{

LargeData data = new LargeData();

LargeQueue.Enqueue(data);

}

for (var i = 0; i < 25000; i++)

{

LargeQueue.Dequeue();

}

msTest2 = Environment.TickCount - time;

MessageBox.Show(String.Format("Duraction of queue (16 bytes): {0} ms\nDuraction of queue (256 Kbytes): {1} ms", msTest1, msTest2));

Stack<SmallData> SmallStack = new Stack<SmallData>();

time = Environment.TickCount;

for (var i = 0; i < 25000; i++)

{

SmallData data = new SmallData();

SmallStack.Push(data);

SmallStack.Pop();

}

msTest1 = Environment.TickCount - time;

Stack<LargeData> LargeStack = new Stack<LargeData>();

time = Environment.TickCount;

for (var i = 0; i < 25000; i++)

{

LargeData data = new LargeData();

LargeStack.Push(data);

LargeStack.Pop();

}

msTest1 = Environment.TickCount - time;

MessageBox.Show(String.Format("Duraction of stack (16 bytes): {0} ms\nDuraction of stack (256 Kbytes): {1} ms", msTest1, msTest2));

LinkedList<SmallData> SmallList = new LinkedList<SmallData>();

time = Environment.TickCount;

for (var i = 0; i < 25000; i++)

{

SmallData data = new SmallData();

SmallList.Find(data);

SmallList.AddLast(data);

}

msTest1 = Environment.TickCount - time;

LinkedList<LargeData> LargeList = new LinkedList<LargeData>();

time = Environment.TickCount;

for (var i = 0; i < 25000; i++)

{

LargeData data = new LargeData();

LargeList.Find(data);

LargeList.AddLast(data);

}

msTest2 = Environment.TickCount - time;

MessageBox.Show(String.Format("Duraction of linkedList (16 bytes): {0} ms\nDuraction of linkedList (256 Kbytes): {1} ms", msTest1, msTest2));

}

}

}