|  |  |
| --- | --- |
|  | /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 № 7

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 «06» декабря 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»).

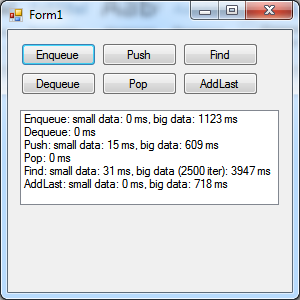
Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

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\_rightWay |
|  | { |
|  | public partial class Form1 : Form |
|  | { |
|  | public Form1() |
|  | { |
|  | InitializeComponent(); |
|  | } |
|  |  |
|  | public class Node |
|  | { |
|  | public Node next; |
|  | public Node prev; |
|  | public Object data; |
|  | } |
|  |  |
|  | public class DoubleLinkedList |
|  | { |
|  | private Node firstNode; |
|  | private Node lastNode; |
|  |  |
|  | public void AddAfter(Node node, Node newNode) |
|  | { |
|  | newNode.prev = node; |
|  | if (node.next == null) |
|  | { |
|  | newNode.next = null; |
|  | lastNode = newNode; |
|  | } |
|  | else |
|  | { |
|  | newNode.next = node.next; |
|  | node.next.prev = newNode; |
|  | } |
|  | node.next = newNode; |
|  | } |
|  |  |
|  | public void AddBefore(Node node, Node newNode) |
|  | { |
|  | newNode.next = node; |
|  | if (node.prev == null) { |
|  | newNode.prev = null; |
|  | firstNode = newNode; |
|  | } |
|  | else |
|  | { |
|  | newNode.prev = node.prev; |
|  | node.prev.next = newNode; |
|  | } |
|  | node.prev = newNode; |
|  | } |
|  |  |
|  | public void AddFirst(Node newNode) |
|  | { |
|  | if (firstNode == null) |
|  | { |
|  | firstNode = newNode; |
|  | lastNode = newNode; |
|  | newNode.prev = null; |
|  | newNode.next = null; |
|  | } |
|  | else |
|  | AddBefore(firstNode, newNode); |
|  | } |
|  |  |
|  | public void AddLast(Node newNode) |
|  | { |
|  | if (lastNode == null) |
|  | AddFirst(newNode); |
|  | else |
|  | AddAfter(lastNode, newNode); |
|  | } |
|  |  |
|  | public void Remove(Node node) |
|  | { |
|  | if (node.prev == null) |
|  | firstNode = node.next; |
|  | else |
|  | node.prev.next = node.next; |
|  | if (node.next == null) |
|  | lastNode = node.prev; |
|  | else |
|  | node.next.prev = node.prev; |
|  | } |
|  |  |
|  | public Node Find(Node node) |
|  | { |
|  | Node firstNodeCopy = firstNode; |
|  | for (; firstNodeCopy != null && !firstNodeCopy.data.Equals(node.data); firstNodeCopy = firstNodeCopy.next) ; |
|  | return firstNodeCopy; |
|  | } |
|  |  |
|  | public void Clear() |
|  | { |
|  | firstNode = null; |
|  | lastNode = null; |
|  | } |
|  |  |
|  | public Node LastElement() => lastNode; |
|  |  |
|  | public Node FirstElement() => firstNode; |
|  |  |
|  | public int Count() |
|  | { |
|  | int count = 0; |
|  | Node firstNodeCopy = firstNode; |
|  | for (; firstNodeCopy != null; firstNodeCopy = firstNodeCopy.next, count++); |
|  | return count; |
|  | } |
|  |  |
|  | public Node ElementAt(int index) |
|  | { |
|  | Node firstNodeCopy = firstNode; |
|  | for (; firstNodeCopy != null && index > 0; firstNodeCopy = firstNodeCopy.next, index--); |
|  | return firstNodeCopy; |
|  | } |
|  |  |
|  | public List<Object> ToList() |
|  | { |
|  | Node firstNodeCopy = firstNode; |
|  | List<Object> result = new List<Object>(); |
|  | while (firstNodeCopy != null) |
|  | { |
|  | result.Add(firstNodeCopy.data); |
|  | firstNodeCopy = firstNodeCopy.next; |
|  | } |
|  | return result; |
|  | } |
|  | } |
|  |  |
|  | public class LinkedStack |
|  | { |
|  | private DoubleLinkedList stack = new DoubleLinkedList(); |
|  |  |
|  | public void Push(Object item) |
|  | { |
|  | stack.AddLast(new Node() { data = item }); |
|  | } |
|  |  |
|  | public Object Pop() |
|  | { |
|  | Object item = Peek(); |
|  | if ((Node)item != null) |
|  | { |
|  | stack.Remove((Node)item); |
|  | return item; |
|  | } |
|  | else |
|  | return null; |
|  | } |
|  |  |
|  | public void Clear() => stack.Clear(); |
|  |  |
|  | public Object Peek() => stack.LastElement(); |
|  |  |
|  | public List<Object> ToList() => stack.ToList(); |
|  |  |
|  | public int Count() => stack.Count(); |
|  | } |
|  |  |
|  | public class LinkedQueue |
|  | { |
|  | private DoubleLinkedList list = new DoubleLinkedList(); |
|  |  |
|  | public void Enqueue(Object item) |
|  | { |
|  | list.AddLast(new Node() { data = item }); |
|  | } |
|  |  |
|  | public Object Dequeue() |
|  | { |
|  | Object item = list.FirstElement(); |
|  | if ((Node)item != null) |
|  | { |
|  | list.Remove((Node)item); |
|  | return item; |
|  | } |
|  | else |
|  | return null; |
|  | } |
|  |  |
|  | public void Clear() => list.Clear(); |
|  |  |
|  | public List<Object> ToList() => list.ToList(); |
|  |  |
|  | public int Count() => list.Count(); |
|  | } |
|  |  |
|  | unsafe struct SmallData |
|  | { |
|  | public fixed byte a[16]; |
|  | } |
|  |  |
|  | unsafe struct LargeData |
|  | { |
|  | public fixed byte a[25600]; |
|  | } |
|  |  |
|  |  |
|  |  |
|  | public int Test(Action<int> func, int amountIter = 25000) |
|  | { |
|  | int time = Environment.TickCount; |
|  |  |
|  | for (var i = 0; i < amountIter; i++) |
|  | { |
|  | func(0); |
|  | } |
|  |  |
|  | return Environment.TickCount - time; |
|  | } |
|  |  |
|  | DoubleLinkedList list = new DoubleLinkedList(); |
|  | LinkedStack stack = new LinkedStack(); |
|  | LinkedQueue queue = new LinkedQueue(); |
|  |  |
|  | private void button1\_Click(object sender, EventArgs e) |
|  | { |
|  | listBox1.Items.Insert(0, String.Format("Enqueue: small data: {0} ms, big data: {1} ms", |
|  | Test(x => |
|  | { |
|  | SmallData data = new SmallData(); |
|  | queue.Enqueue(data); |
|  | }).ToString(), |
|  | Test(x => |
|  | { |
|  | LargeData data = new LargeData(); |
|  | queue.Enqueue(data); |
|  | }).ToString() |
|  | )); |
|  | } |
|  |  |
|  | private void button2\_Click(object sender, EventArgs e) |
|  | { |
|  | listBox1.Items.Insert(0, String.Format("Dequeue: {0} ms", |
|  | Test(x => |
|  | { |
|  | queue.Dequeue(); |
|  | }).ToString() |
|  | )); |
|  | } |
|  |  |
|  | private void button3\_Click(object sender, EventArgs e) |
|  | { |
|  | listBox1.Items.Insert(0, String.Format("Push: small data: {0} ms, big data: {1} ms", |
|  | Test(x => |
|  | { |
|  | SmallData data = new SmallData(); |
|  | stack.Push(data); |
|  | }).ToString(), |
|  | Test(x => |
|  | { |
|  | LargeData data = new LargeData(); |
|  | stack.Push(data); |
|  | }).ToString() |
|  | )); |
|  | } |
|  |  |
|  | private void button4\_Click(object sender, EventArgs e) |
|  | { |
|  | listBox1.Items.Insert(0, String.Format("Pop: {0} ms", |
|  | Test(x => |
|  | { |
|  | stack.Pop(); |
|  | }).ToString() |
|  | )); |
|  | } |
|  |  |
|  | private void button5\_Click(object sender, EventArgs e) |
|  | { |
|  | list.Clear(); |
|  | Test(x => |
|  | { |
|  | SmallData data = new SmallData(); |
|  | list.AddFirst(new Node() { data = data }); |
|  | LargeData data1 = new LargeData(); |
|  | list.AddLast(new Node() { data = data1 }); |
|  | }); |
|  |  |
|  | listBox1.Items.Insert(0, String.Format("Find: small data: {0} ms, big data (2500 iter): {1} ms", |
|  | Test(x => |
|  | { |
|  | SmallData data = new SmallData(); |
|  | list.Find(new Node() { data = data }); |
|  | }).ToString(), |
|  | Test(x => |
|  | { |
|  | LargeData data = new LargeData(); |
|  | list.Find(new Node() { data = data }); |
|  | }, 2500).ToString() |
|  | )); |
|  | } |
|  |  |
|  | private void button6\_Click(object sender, EventArgs e) |
|  | { |
|  | list.Clear(); |
|  | listBox1.Items.Insert(0, String.Format("AddLast: small data: {0} ms, big data: {1} ms", |
|  | Test(x => |
|  | { |
|  | SmallData data = new SmallData(); |
|  | list.AddLast(new Node() { data = data }); |
|  | }).ToString(), |
|  | Test(x => |
|  | { |
|  | LargeData data = new LargeData(); |
|  | list.AddLast(new Node() { data = data }); |
|  | }).ToString() |
|  | )); |
|  | } |
|  | } |
|  | } |