Практическая работа №5 (8 часов)

СВЯЗНЫЕ СПИСКИ

Цель работы: исследовать возможности создания и использования структуры данных – связный список.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Связный список (Linked List) является простейшим типом данных динамической структуры, состоящей из элементов (***узлов***). Каждый узел включает в себя в классическом варианте два поля:

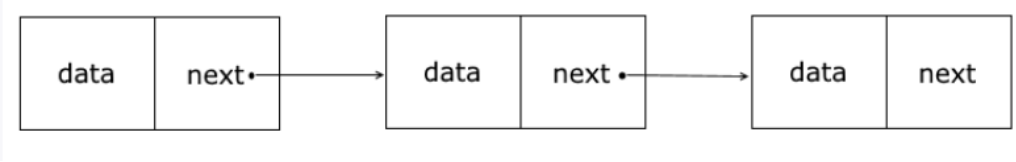
- данные (в качестве данных может выступать переменная, объект класса или структуры и т. д.)
- указатель или ссылка на следующий и (или) предыдущий узел в списке.

Таким образом, если в массиве положение элементов определяется индексами, то в связном списке - указателями на следующий и (или) на предыдущий элемент.

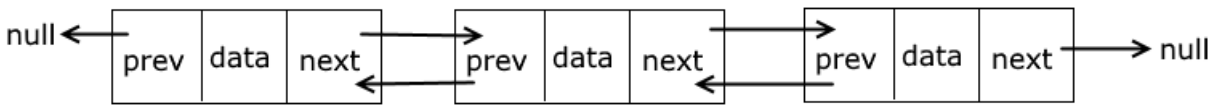
Элементы связанного списка можно помещать и исключать произвольным образом. Доступ к списку осуществляется через указатель, который содержит адрес первого элемента списка, называемый ***корнем списка***.

По количеству полей указателей различают однонаправленный (односвязный) и двунаправленный (двусвязный) списки.

Связный список, содержащий только один указатель на следующий элемент, называется односвязным.



Связный список, содержащий два поля указателя – на следующий элемент и на предыдущий, называется двусвязным.



Различают 4 основных вида списков:

- Односвязный линейный список (ОЛС). Каждый узел ОЛС содержит 1 поле указателя на следующий узел. Поле указателя последнего узла содержит нулевое значение (указывает на NULL).
- Односвязный циклический список (ОЦС). Каждый узел ОЦС содержит 1 поле указателя на следующий узел. Поле указателя последнего узла содержит адрес первого узла (корня списка).
- Двусвязный линейный список (ДЛС). Каждый узел ДЛС содержит два поля указателей: на следующий и на предыдущий узел. Поле указателя на следующий узел последнего узла содержит нулевое значение (указывает на NULL). Поле указателя на предыдущий узел первого узла (корня списка) также содержит нулевое значение (указывает на NULL).
- Двусвязный циклический список (ДЦС). Каждый узел ДЦС содержит два поля указателей: на следующий и на предыдущий узел. Поле указателя на следующий узел последнего узла содержит адрес первого узла (корня списка). Поле указателя на предыдущий узел первого узла (корня списка) содержит адрес последнего узла.

Односвязный линейный список

Основные действия, производимые над элементами ОЛС:

- Инициализация списка
- Добавление узла в список
- Удаление узла из списка
- Вывод элементов списка
- Взаимообмен двух узлов списка

Рассмотрим создание односвязного списка.

Перед созданием списка нам надо определить класс узла, который будет представлять одиночный объект в списке:

```
public class Node <T>
{
   public Node (T data)
   {
      Data = data;
   }
```

```
        public T Data { get; set; }
        public Node <T> Next { get; set; }
    }
```

Класс Node является обобщенным, поэтому может хранить данные любого типа. Для хранения данных предназначено свойство Data. Для ссылки на следующий узел определено свойство Next.

Разберем основные моменты. В зависимости от конкретных задач реализация списков может отличаться, но для всех реализаций характерны прежде всего два метода: добавление и удаление.

Но прежде чем выполнять различные операции с данными, в классе списка определяются три переменные: head - головной/первый элемент, tail - последний/хвостовой элемент и count - количество элементов в списке.

Определим сам класс списка:

```
using System.Collections;
using System.Collections.Generic;
 namespace SimpleAlgorithmsApp
{
   public class LinkedList<T> // односвязный список
   {
      Node<T> head;       // головной/первый элемент
      Node<T> tail;       // последний/хвостовой элемент
      int count;          // количество элементов в списке

 // добавление элемента
      public void Add(T data){…}
// удаление элемента
      public bool Remove(T data) {…}

      public int Count { get { return count; } }
      public bool IsEmpty { get { return count == 0; } }

// очистка списка
      public void Clear() {…}

// содержит ли список элемент
      public bool Contains(T data) {…}
```

```
 // добавление в начало
      public void AppendFirst(T data) {…}
   }
}
```

Метод добавления:

```
public void Add(T data)
{
   Node<T> node = new Node<T>(data);

   if (head == null)
      head = node;
   else
      tail.Next = node;

   tail = node;
   count++;
}
```

Если у нас не установлена переменная head (то есть список пуст), то устанавливаем head и tail. После добавления первого элемента они будут указывать на один и тот же объект.

Если же в списке есть как минимум один элемент, то устанавливаем свойство tail.Next - теперь оно хранит ссылку на новый узел. И переустанавливаем tail - теперь она ссылается на новый узел.

Особняком стоит метод добавления в начало списка, где достаточно переустановить ссылку на головной элемент:

```
public void AppendFirst(T data)
{
   Node<T> node = new Node<T>(data);
   node.Next = head;
   head = node;
   if (count == 0)
      tail = head;
   count++;
}
```

Удаление элемента:

```
public bool Remove(T data)
{
    Node<T> current = head;
    Node<T> previous = null;

    while (current != null)
    {
        if (current.Data.Equals(data))
        {
            // Если узел в середине или в конце
            if (previous != null)
            {
                // убираем узел current, теперь previous ссылается не на current,
                // а на current.Next
                previous.Next = current.Next;

                // если current.Next не установлен, значит узел последний,
                // изменяем переменную tail
                if (current.Next == null)
                    tail = previous;
            }
            else
            {
                // если удаляется первый элемент
                // переустанавливаем значение head
                head = head.Next;

                // если после удаления список пуст, сбрасываем tail
                if (head == null)
                    tail = null;
            }
            count--;
            return true;
        }
        previous = current;
        current = current.Next;
    }
    return false;
```

}

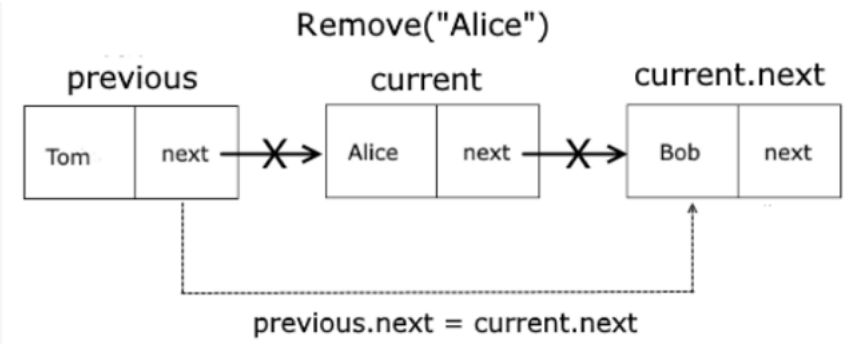
Алгоритм удаления элемента представляет следующую последовательность шагов:

1. Поиск элемента в списке путем перебора всех элементов
2. Установка свойства Next у предыдущего узла (по отношению к удаляемому) на следующий узел по отношению к удаляемому.

Для отслеживания предыдущего узла применяется переменная previous. Если элемент найден, и переменная previous равна null, то удаление идет сначала, и в этом случае происходит переустановка переменной head, то есть головного элемента.

Если же previous не равна null, то реализуются шаги выше описанного алгоритма.

Сложность такого алгоритма составляет **O(n)**. Графически удаление можно представить так:



Чтобы проверить наличие элемента, используется метод Contains:

```
public bool Contains(T data)
{
   Node<T> current = head;
   while (current != null)
   {
      if (current.Data.Equals(data))
         return true;
      current = current.Next;
   }
   return false;
}
```

Реализация интерфейса **IEnumerable** не является неотъемлемой частью односвязных списков, однако предоставляет эффективный метод для перебора коллекции во внешней программе с помощью цикла foreach, иначе бы пришлось реализовать иные конструкции по перебору списка:

```
// выводим элементы
foreach(var item in linkedList)
{
   Console.WriteLine(item);
}
```

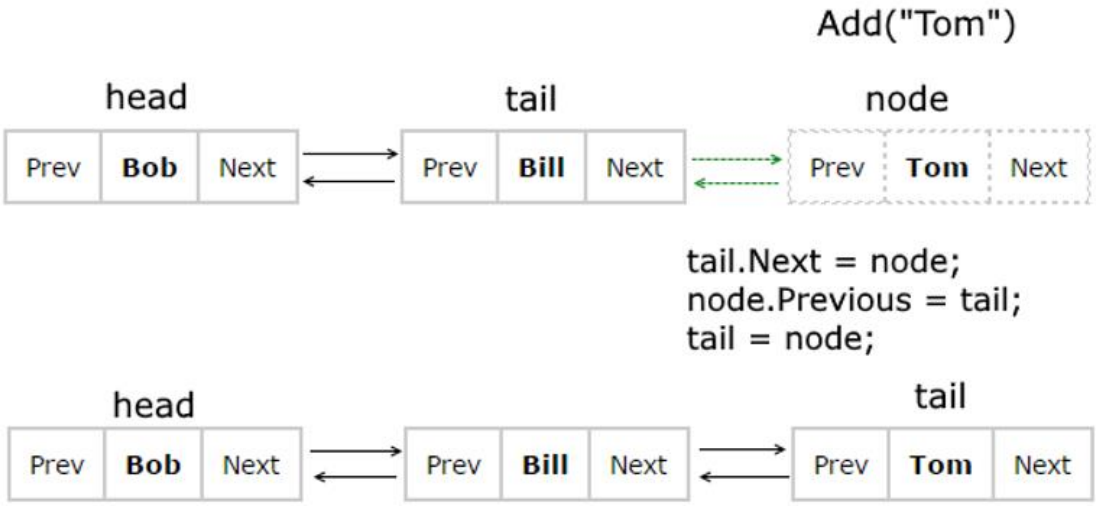
Двусвязные списки

Двусвязные списки также представляют последовательность связанных узлов, однако теперь каждый узел хранит ссылку на следующий и на предыдущий элементы.

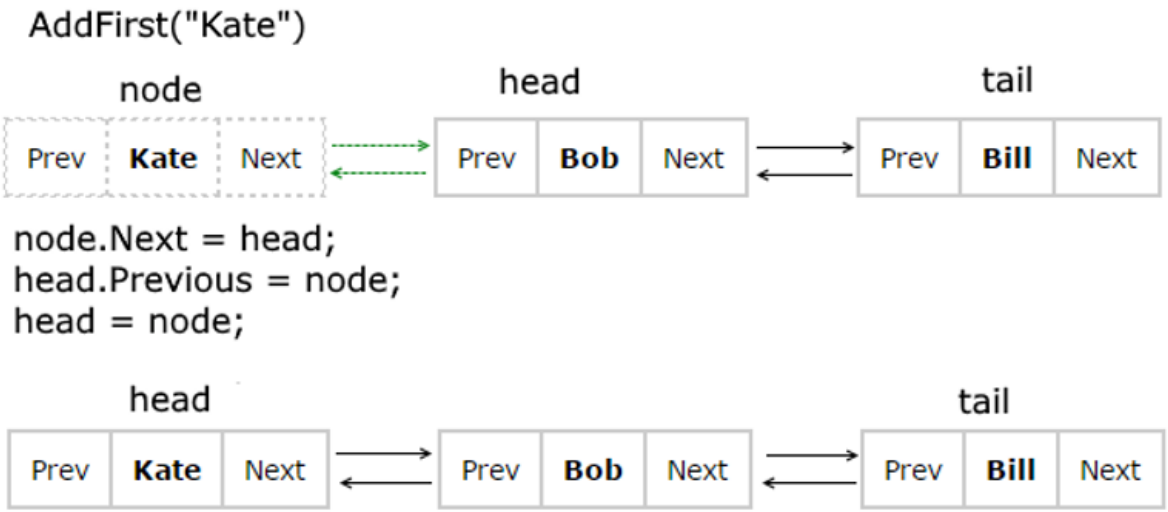
Двунаправленность списка приходится учитывать при добавлении или удалении элемента, так как кроме ссылки на следующий элемент надо устанавливать и ссылку на предыдущий. Но в то же время появляется возможность обходить список как от первого к последнему элементу, так и наоборот - от последнего к первому элементу. В остальном двусвязный список ничем не будет отличаться от односвязного списка. Разница заключается в необходимости установки свойства Previous для узлов списка.

В методе добавления Add(), если в списке уже есть элементы, то у добавляемого узла свойство Previous указывает на узел, который до этого хранился в переменной tail:

```
if (head == null)
   head = node;
else
{
   tail.Next = node;
   node.Previous = tail;
}
tail = node;
```



Аналогично в методе AddFirst() добавления в начало списка для головного элемента свойство Previous начинает указывать на новый элемент, а новый элемент, таким образом, становиться первым элементом в списке.



При удалении вначале необходимо найти удаляемый элемент. Затем в общем случае надо переустановить две ссылки:

```
current.Next.Previous = current.Previous;
current.Previous.Next = current.Next;
```

Если удаляются первый и последний элемент, соответственно надо переустановить переменные head и tail.

И также в отличие от односвязной реализации необходимо добавить метод BackEnumerator() для перебора элементов с конца.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Выполнить реализацию класса односвязного списка.
2. Создать список из 20 элементов. Элементы списка вводятся пользователем.
3. Выполнить реализацию следующих функций:
   - Вставить элемент в начало списка.
   - Включить элемент внутрь списка (после i-го элемента).
   - Включить элемент внутрь списка (перед i-м элементом).
   - Включить элемент в конец списка.
   - Подсчитать количество элементов списка.
   - Удалить i-тый элемент из списка.
   - Удалить каждый k-й (по счету) элемент из списка.
   - Удалить элемент из конца списка.
   - Удалить весь список.
4. Создать метод, удаляющий k элементов с конца списка.
5. Создать метод, добавляющий k одинаковых элементов в конец списка.
6. Добавить в класс LinkedList конструктор, имеющий два целых аргумента a и b (a<b), который создает список из b-a узлов, значения которых меняются от a до b с шагом 1. Например, LinkedList list(2, 7); (список должен содержать элементы {2, 3, 4, 5, 6, 7}
7. Вывести элементы списка в обратном порядке.
8. Задания 1-7 выполнить для двусвязного списка.
9*. Выполнить сортировку списка методом пузырька (без использования массивов).

Литература

1. Томас Х. Кормен Алгоритмы: вводный курс Томаса Х. Кормена. - М: Вильямс, 2014. - 208 с

2. Ахо А.В., Хопкрофт Д., Ульман Д.Д. Структуры данных и алгоритмы. – М.: Вильямс, 2001. – 384 с.

3. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ. – 2-е изд. – М.: Вильямс, 2005. – 1296 с.

4. Левитин А.В. Алгоритмы: введение в разработку и анализ. – М.: Вильямс, 2006. – 576 с.

5. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: ДиаСофт, 2001. – 688 с.

6. Скиена С.С. Алгоритмы. Руководство по разработке. – 2-е изд. – СПб: БХВ, 2011 – 720 с.

7. Макконнелл Дж. Основы современных алгоритмов. – 2е изд. – М.: Техносфера, 2004. – 368 с.

8. Миллер Р. Последовательные и параллельные алгоритмы: общий подход. – М.: БИНОМ, 2006. – 406 с.

9. Дональд Э. Кнут Искусство программирования, том 1. Основные алгоритмы, 3-е изданиею – М: Вильямс, 2015. – 720 с.