**Разработка класса связанного списка**

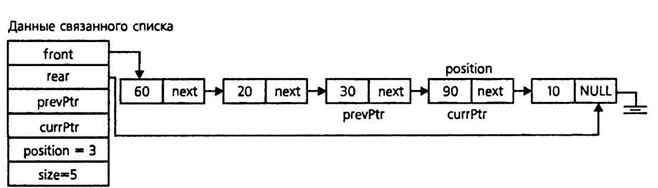
Программист может использовать класс Node вместе с утилитными функ­ циями, находящимися в nodelib.h, для работы с приложениями связанных списков. Такой подход вынуждает программиста создавать каждый узел и выполнять непосредственно списочные операции низкого уровня. Более структурированный подход определяет класс связанных списков, который организует базовые списочные операции как функции-члены. В этом разделе обсуждается тип данных-членов и операций, которые должны быть включены в класс связанных списков, с использованием знаний, приобретенных при разработке алгоритмов с классом Node. Мы также заранее оговариваем тот факт, что наш класс связанных списков будет использоваться для реализации других списочных коллекций, которые включают связанные стеки, очереди и класс SeqList. В следующем разделе мы объявляем класс LinkedList и определяем его функции-члены.

**Данные-члены связанных списков**

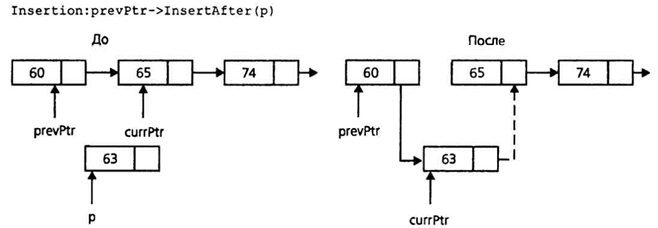
Связанный список состоит из множества объектов Node, связанных вместе от начала до конца списка. Список имеет начальный узел, называемый го­ ловой, который определяет первый узел списка. Последний узел в списке имеет поле указателей со значением NULL, и на него ссылаются как на указатель-хвост. Нам необходимо, чтобы класс связанных списков содержал указатели на начало и на хвост списка, так как это полезно для многих приложений и важно при реализации связанной очереди.

Связанный список предоставляет последовательный доступ к элементам данных и использует указатель для задания текущего положения прохождения в списке. Наш связанный список содержит указатель currPtr для ссылки на это текущее положение в терминах его места в списке. Первый элемент в списке имеет позицию (position) 0 со следующим элементом в позиции 1 и так далее.

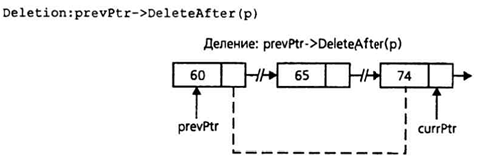
Количество элементов в связанном списке содержится в переменной size. Это позволяет нам определять пустой список или возвращать счетчик количества элементов в списке. В следующем списке текущим положением является эле­ мент со значением 90, расположенный в позиции 3: Адрес prevPtr используется для вставки и удаления узла в текущем по­ ложении с использованием Node-методов InsertAfter и DeleteAfter. Например,

****

в следующем связанном списке показана вставка Node, на который ссылается prevPtr:

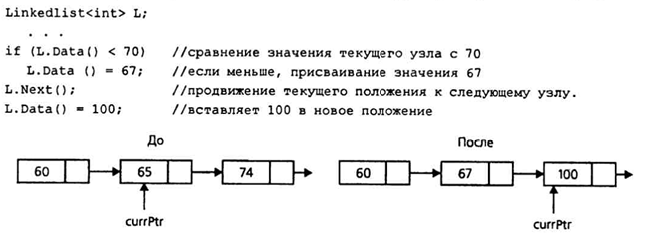


Узел, на который ссылается prevPtr, также используется при удалении элемента из связанного списка:

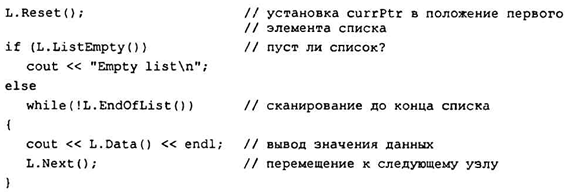


**Операции связанных списков**

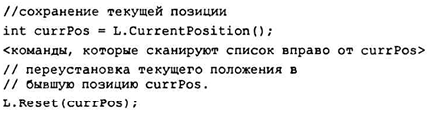
Пользователь должен иметь возможность переходить от элемента к эле­ менту в списке. Простой метод с именем Next передвигает текущее значение к следующему узлу. В текущем положении операция Data возвращает ссылку на поле данных узла. Это дает возможность выполнять выборку или изменять поле данных узла без необходимости понимания пользователем того, как эти данные сохраняются классом Node. Для иллюстрации этих операций пред­ положим, что L является связанным списком целых, чье текущее положение находится в узле со значением данных 65. Следующие операторы изменяют значение текущего узла на 67 и значение следующего узла — на 100:



В приложениях нам иногда необходимо устанавливать текущее положение в определенное место в списке. Это выполняет метод Reset. Он принимает параметр pos и перемещает текущее положение списка в эту позицию. Зна­ чением по умолчанию параметра pos является 0, которое устанавливает те­ кущее положение на первый элемент списка. От него приложение может сканировать узлы, используя Next. Сканирование списка завершается, когда условие EndOfList является равным TRUE. Например, простой цикл выводит элементы в списке. Перед сканированием списка мы сначала проверяем ус­ ловие ListEmpty (пустой список):



Пользователь имеет доступ к текущей позиции списка при помощи метода Currentposition. Метод Reset может применяться для возвращения указателей списка в исходное положение. Эта возможность используется в таких задачах, как нахождение максимального значения в списке и установка списка на этот узел для последующего удаления или вставки.



Добавление и удаление узлов являются основными операциями в связан­ ном списке. Эти операции могут выполняться в начале и хвосте списка или в текущем положении.

Операции вставки. Операции вставки создают новый узел с новым полем данных. Затем узел помещается в список в текущее положение или непо­ средственно после него.

Операция InsertAfter помещает новый узел после текущей позиции и присваивает currPtr новому узлу. Эта операция служит той же цели, что и метод InsertAfter в классе Node.

Метод InsertAfter помещает новый узел в текущее положение. Новый узел помещается непосредственно перед текущим узлом. Текущая позиция устанавливается на новый узел. Эта операция используется при создании упорядоченного списка.

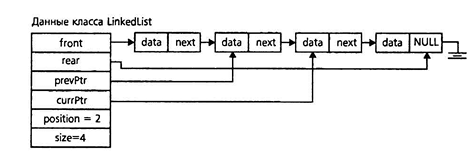
Класс упорядоченных списков имеет операции Insertfront и InsertRear для добавления новых узлов в голову и в хвост списка. Эти операции уста­ навливают текущую позицию на новый узел.

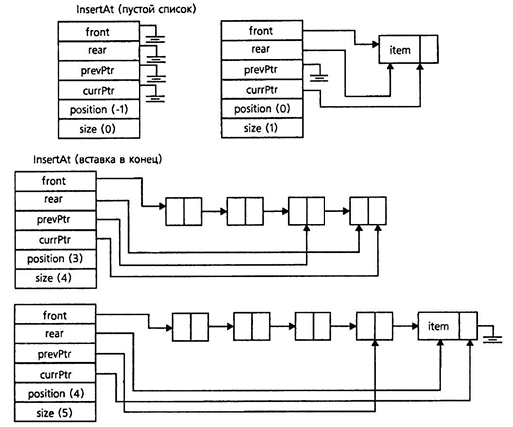
Операции удаления. Операции удаления удаляют узел из списка. DeleteAt удаляет узел в текущей позиции, a DeleteFront удаляет первый узел в списке.

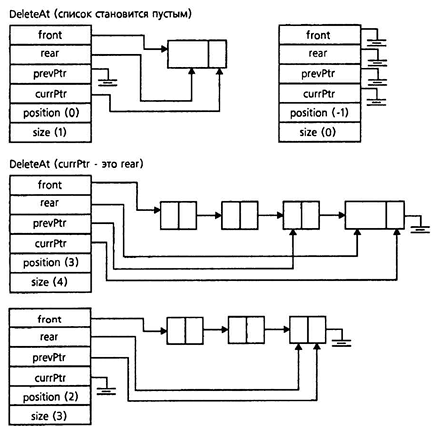
Другие методы. Класс связанных списков создает динамические данные, поэтому он должен иметь конструктор копирования, деструктор и перегру­ женный оператор присваивания. Пользователь может явно очистить список с помощью оператора ClearList.

**9.5. Реализация класса LinkedList**

Спецификация класса LinkedList ссылается на класс Node. Реализация класса LinkedList использует много методов, описанных разделе 9.1 с классом Node. Алгоритмы, использующемые функциями в nodelib.h, являются основой для разработки класса LinkedList. Однако мы должны знать о дополнительной сложности поддержания указателей front и rear, которые определяют доступ к списку, указателей currPtr и prevPtr, которые сохраняют информацию о текущем положении прохождения, местонахождения данных и size (размера списка). Методы LinkedList отвечают за изменение этих данных всякий раз, когда изменяется состояние списка.







using System;

using System.Runtime.InteropServices;

public class LinkedList<T>

{

private Node<T> front, rear; // указатели на начало и конец списка

private Node<T> prevPtr, currPtr; // указатели, используемые для извлечения, вставки и удаления данных.

private int size; // число элементов в списке.

private int position; // текущее положение в списке, используется методом Reset.

private class Node<T> // представляет узел списка и имеет два поля

{

public T Data; // данные

public Node<T> Next; // указатель на следующий узел в списке.

public Node(T data)

{

Data = data; // присвоим данные узлу

Next = null; // и ссылки на следующий нет - узел первый в списке

}

}

public LinkedList() // конструктор по умолчанию, инициализирует переменные-члены значениями по умолчанию (null и 0).

{

front = rear = prevPtr = currPtr = null;

size = position = 0;

}

public LinkedList(LinkedList<T> L) // конструктор копирования, инициализирует переменные-члены значениями по умолчанию (null и 0)

{

front = rear = prevPtr = currPtr = null;

size = position = 0;

CopyList(L);

}

~LinkedList() // деструктор

{

ClearList(); // для удаления всех элементов списка и освобождения памяти.

}

public void ClearList()

{

Node<T> curr = front; // объявление указателя на первый элемент списка

while (curr != null) // пока указатель не указывает на null

{

Node<T> temp = curr; // создание временного указателя на текущий элемент списка

curr = curr.Next; // переход к следующему элементу списка

FreeNode(temp); // удаление временного указателя

}

front = null; // обнуление указателя на первый элемент

rear = null; // обнуление указателя на последний элемент

currPtr = null; // обнуление указателя на текущий элемент

prevPtr = null; // обнуление указателя на предыдущий элемент

size = 0; // обнуление размера списка

position = -1; // обнуление позиции текущего элемента

}

public void Assign(LinkedList<T> L)

{

if (this != L) // если текущий объект не является тем же самым объектом, что и L

{

ClearList(); // очистить текущий список

CopyList(L); // скопировать содержимое списка L в текущий список

}

}

public int ListSize()

{

return size; // возвращает размер списка

}

public bool ListEmpty()

{

return size == 0; // возвращает true, если размер списка равен 0

}

private void FreeNode(Node<T> p)

{

if (p != null) // если указатель p не указывает на null

{

FreeNode(p.Next); // вызов этой же функции для следующего элемента списка

p = null; // удаляем текущий элемент списка

}

}

private void CopyList(LinkedList<T> L)

{

if (L.front == null) // Если переданный список пуст, то обнуляем поля текущего списка.

{

front = null;

rear = null;

size = 0;

}

else

{

Node<T> currPtr = L.front; // currPtr указывает на текущий узел в списке L

Node<T> lastPtr = null; // lastPtr указывает на последний скопированный узел в текущем списке.

while (currPtr != null) //Пока не конец списка L.

{

Node<T> newNode = new Node<T>(currPtr.Data); // новый узел, который содержит данные из текущего узла списка L.

if (front == null) // Если текущий список пуст

{

front = newNode; // новый узел становится первым

}

else

{

lastPtr.Next = newNode; // иначе новый узел добавляется в конец списка.

}

// Обновляем lastPtr и currPtr на новый узел и следующий узел списка L соответственно.

lastPtr = newNode;

currPtr = currPtr.Next;

}

rear = lastPtr; // Обновляем поле на последний скопированный узел

size = L.size; // Обновляем поле на размер списка L

}

}

// Метод, который устанавливает текущую позицию в списке на позицию pos.

public void Reset(int pos = 0)

{

if (pos >= 0 && pos <= size) // Проверяем, что позиция находится в пределах списка.

{

position = pos; // Обновляем поле position

currPtr = front; // устанавливаем на первый узел списка

prevPtr = null; // обнуляем.

for (int i = 0; i < position; i++) //Переходим на узел с позицией position,

//чтобы prevPtr указывал на предыдущий узел.

{

prevPtr = currPtr;

currPtr = currPtr.Next;

}

}

}

// метод для перехода к следующему элементу списка

public void Next()

{

// текущий указатель не равен null, т.е. элемент, на который указывает currPtr не является последним в списке

if (currPtr != null)

{

prevPtr = currPtr; // сохранение текущего указателя

currPtr = currPtr.Next; // перемещение текущего указателя на следующий элемент

position++; // увеличение значения переменной position, чтобы отразить изменение текущей позиции в списке

}

}

// метод для проверки, достигнут ли конец списка

public bool EndOfList()

{

return currPtr == null;

}

// метод для получения текущей позиции в списке

public int CurrentPosition()

{

return position; // eсли метод Reset() не был вызван, то значение по умолчанию равно 0.

}

// объявление метода для вставки элемента в начало списка.

public void InsertFront(T item)

{

Node<T> newNode = new Node<T>(item); // создание нового узла с переданным элементом.

if (front == null) // проверка, что список пустой.

{

rear = newNode;// если список пустой, то новый элемент будет последним в списке.

}

else

{

newNode.Next = front;// - новый элемент указывает на первый элемент списка.

}

front = newNode; // новый элемент становится первым в списке.

size++; // - увеличение размера списка.

}

// объявление метода для вставки элемента в конец списка.

public void InsertRear(T item)

{

Node<T> newNode = new Node<T>(item); // - создание нового узла с переданным элементом.

if (front == null) // проверка, что список пустой.

{

front = newNode; // - если список пустой, то новый элемент будет первым в списке.

}

else

{

rear.Next = newNode; // последний элемент списка указывает на новый элемент.

}

rear = newNode; // новый элемент становится последним в списке.

size++;

}

// объявление метода для вставки элемента перед текущим элементом.

public void InsertAt(T item)

{

if (currPtr == null) // проверка, что текущий элемент не задан.

{

InsertRear(item); // если текущий элемент не задан, то элемент вставляется в конец списка.

}

else if (prevPtr == null) // проверка, что текущий элемент первый в списке.

{

InsertFront(item); // - если текущий элемент первый в списке, то элемент вставляется в начало списка.

}

else // если текущий элемент не первый и не последний в списке.

{

Node<T> newNode = new Node<T>(item); // создание нового узла с переданным элементом.

newNode.Next = currPtr; // новый элемент указывает на текущий элемент.

prevPtr.Next = newNode; // предыдущий элемент указывает на новый элемент.

size++;

}

}

public T DeleteFront()

{

if (front == null)

{

throw new InvalidOperationException("List is empty.");

}

T data = front.Data;

front = front.Next;

if (front == null)

{

rear = null;

}

size--;

return data;

}

// объявление метода InsertAfter, который принимает элемент типа T и добавляет его в список после текущей позиции.

public void InsertAfter(T item)

{

if (currPtr == null) // проверка, что текущий указатель равен null, то есть список пустой.

{

InsertRear(item); // если список пустой, то вызываем метод InsertRear, чтобы вставить элемент в конец списка.

}

else

{

Node<T> newNode = new Node<T>(item); // создание нового узла, содержащего переданный элемент.

newNode.Next = currPtr.Next; // устанавливаем следующий указатель нового узла на следующий элемент после текущего.

currPtr.Next = newNode; // устанавливаем следующий указатель текущего узла на новый узел.

if (currPtr == rear) // является ли текущий узел последним элементом в списке.

{

rear = newNode; // Если да, то устанавливаем указатель на последний элемент на новый узел.

}

size++;

}

}

public T Data()

{

if (currPtr == null)

{

throw new InvalidOperationException("No current element.");

}

return currPtr.Data;

}

public void DeleteRear()

{

if (rear == null) // проверяется, что список не пустой

{

throw new InvalidOperationException("List is empty."); // exception

}

if (front == rear) // является ли удаляемый элемент последним элементом в списке.

{

// Если front и rear указывают на один и тот же узел

front = null; // обнуляются указатели на голову и хвост списка.

rear = null;

}

else // Если же rear не указывает на последний узел в списке

{

Node<T> currPtr = front;

// то происходит перебор элементов с головы списка,

// пока текущий элемент не будет указывать на предпоследний узел

while (currPtr.Next != rear)

{

currPtr = currPtr.Next;

}

currPtr.Next = null; // currPtr.Next устанавливается в null, тк последний элемент больше не является частью списка

rear = currPtr; // обновляется указатель на rear так, чтобы он указывал на новый последний элемент.

}

size--;

}

public void DeleteAt(int pos)

{

// Проверка, что позиция находится в допустимом диапазоне

if (pos < 0 || pos >= size)

{

throw new ArgumentOutOfRangeException(nameof(pos), "Position is out of range.");

}

// Если позиция равна 0, вызываем метод DeleteFront() для удаления первого элемента списка и возвращаемся

if (pos == 0)

{

DeleteFront();

return;

}

Node<T> currPtr = front;

Node<T> prevPtr = null;

// Перебираем элементы списка до позиции, которую нужно удалить

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

{

prevPtr = currPtr;

currPtr = currPtr.Next;

}

// Устанавливаем указатель предыдущего элемента на следующий элемент, пропуская текущий элемент и удаляя его из списка

prevPtr.Next = currPtr.Next;

// Если после удаления текущего элемента следующий элемент стал равен null, обновляем указатель на последний элемент

if (prevPtr.Next == null)

{

rear = prevPtr;

}

// Уменьшаем размер списка на единицу

size--;

}

public void BubbleSort()

{

if (size <= 1) return; // если список пуст или состоит из одного элемента, то он уже отсортирован

bool swapped = true;

while (swapped)

{

swapped = false;

Node<T> prev = null;

Node<T> curr = front;

Node<T> next = curr.Next;

while (next != null)

{

if (Comparer<T>.Default.Compare(curr.Data, next.Data) > 0) // если текущий элемент больше следующего

{

swapped = true;

if (prev != null) // если мы не находимся в начале списка

{

prev.Next = next; // перенаправляем ссылку предыдущего элемента на следующий элемент

}

else // если мы находимся в начале списка

{

front = next; // обновляем указатель на начало списка

}

curr.Next = next.Next; // перенаправляем ссылку текущего элемента на следующий элемент после следующего

next.Next = curr; // меняем ссылки у следующего элемента

Node<T> temp = curr; // меняем указатели на текущий и следующий узлы

curr = next;

next = temp;

}

prev = curr;

curr = curr.Next;

next = curr.Next;

}

}

}

public static void PrintList(LinkedList<T> L)

{

for (L.Reset(); !L.EndOfList(); L.Next())

{

Console.Write(L.Data()+" ");

}

Console.WriteLine();

}

public void RemoveDuplicates()

{

// Создаем словарь для хранения уникальных элементов списка

Dictionary<T, bool> dict = new Dictionary<T, bool>();

// Устанавливаем указатель на начало списка

Node<T> current = front;

// Устанавливаем указатель на предыдущий элемент списка

Node<T> previous = null;

// Перебираем все элементы списка

while (current != null)

{

// Если элемент уже встречался ранее, удаляем его из списка

if (dict.ContainsKey(current.Data))

{

previous.Next = current.Next;

size--;

}

else

{

// Если элемент встречается впервые, добавляем его в словарь

dict.Add(current.Data, true);

previous = current;

}

// Переходим к следующему элементу списка

current = current.Next;

}

}

}

class Program

{

static void Main(string[] args)

{

LinkedList<int> list = new LinkedList<int>();

list.InsertRear(2);

list.InsertRear(10);

list.InsertRear(3);

list.InsertRear(5);

list.InsertRear(2);

list.InsertRear(7);

list.InsertRear(10);

Console.WriteLine("Исходный список");

LinkedList<int>.PrintList(list);

list.BubbleSort();

Console.WriteLine("Отсортированный список");

LinkedList<int>.PrintList(list);

list.RemoveDuplicates();

Console.WriteLine("Конечный список");

LinkedList<int>.PrintList(list);

}

}

АТД

Операция "InsertFront"

Условия на вход: принимает значение типа T

Входные данные: значение типа T

Процесс: создание нового узла списка, добавление значения в новый узел, установка указателя front на новый узел, если список был пуст, то установка указателя rear на новый узел, увеличение размера списка на 1

Выход: нет

Постусловия: новый узел добавлен в начало списка

Операция "InsertRear"

Условия на вход: принимает значение типа T

Входные данные: значение типа T

Процесс: создание нового узла списка, добавление значения в новый узел, установка указателя rear на новый узел, если список был пуст, то установка указателя front на новый узел, увеличение размера списка на 1

Выход: нет

Постусловия: новый узел добавлен в конец списка

Операция "InsertAt"

Условия на вход: принимает значение типа T

Входные данные: значение типа T

Процесс: создание нового узла списка, добавление значения в новый узел, установка указателя prevPtr на предыдущий узел, установка указателя currPtr на текущий узел, установка указателя Next нового узла на currPtr, установка указателя Next prevPtr на новый узел, увеличение размера списка на 1

Выход: нет

Постусловия: новый узел добавлен в список после prevPtr и перед currPtr

Операция "InsertAfter"

Условия на вход: принимает значение типа T

Входные данные: значение типа T

Процесс: создание нового узла списка, добавление значения в новый узел, установка указателя Next нового узла на Next currPtr, установка указателя Next currPtr на новый узел, если currPtr указывает на последний узел, то установка указателя rear на новый узел, увеличение размера списка на 1

Выход: нет

Постусловия: новый узел добавлен в список после текущего узла

Операция "DeleteFront"

Условия на вход: нет

Входные данные: нет

Процесс: если список пуст, выбрасывается исключение "List is empty", сохранение значения Data первого узла списка, установка указателя front на следующий узел, уменьшение размера списка на 1, если после удаления первого узла список пуст, то установка указателя rear на null

Выход: значение типа T, которое было сохранено в первом

Операция "BubbleSort"

Условия на вход: нет

Входные данные: нет

Процесс:

Если размер списка меньше или равен 1, то возвращается (список уже отсортирован)

Инициализируется переменная swapped со значением true

Пока swapped равно true:

Устанавливаются указатели prev, curr и next на соответствующие узлы (начиная с первого)

Внутренний цикл выполняется, пока next не станет равным null:

Если значение curr.Data больше значения next.Data:

Устанавливается swapped в true

Если prev не равен null (не находимся в начале списка):

Устанавливается prev.Next на next (перенаправление ссылки предыдущего элемента на следующий элемент)

Иначе (находимся в начале списка):

Обновляется указатель front на next (обновление указателя на начало списка)

Устанавливается curr.Next на next.Next (перенаправление ссылки текущего элемента на элемент после next)

Устанавливается next.Next на curr (смена ссылок у next элемента)

Выполняется обмен указателями curr и next

Переход к следующим узлам: prev = curr, curr = curr.Next, next = curr.Next

Выход: нет

Постусловия: список отсортирован по возрастанию

Операция "DeleteRear"

Условия на вход: нет

Входные данные: нет

Процесс:

Если rear равен null (список пуст), выбрасывается исключение "List is empty"

Если front и rear указывают на один и тот же узел:

Обнуляются указатели front и rear

Иначе:

Создается указатель currPtr, который инициализируется значением front

Выполняется цикл, пока currPtr.Next не станет равным rear (перебор элементов с головы списка до предпоследнего узла)

Устанавливается currPtr.Next в null (удаление последнего элемента из списка)

Обновляется указатель rear, чтобы указывал на новый последний элемент

Уменьшается размер списка на 1

Выход: нет

Постусловия: последний узел списка удален