**Содержание**

[Введение 2](#_Toc482575872)

[1. Теоретическая часть 3](#_Toc482575873)

[1.1. Основные задания управления памятью 3](#_Toc482575874)

[1.2. Связный список 4](#_Toc482575875)

[1.3. Управление свободной памятью 6](#_Toc482575876)

[2. Практическая часть 12](#_Toc482575877)

[2.1. Программная реализация алгоритма двунаправленного связного списка 12](#_Toc482575878)

[Вывод 19](#_Toc482575879)

[Список литературы 20](#_Toc482575880)

# Введение

|  |
| --- |
|  |

Память представляет собой очень важный ресурс, требующий четкого управления. Несмотря на то что в наши дни объем памяти среднего домашнего компьютера в десятки тысяч раз превышает ресурсы компьютера, бывшего в начале 60-х годов, размер компьютерных программ растет быстрее, чем объем памяти. Закон Паркинсона можно перефразировать следующим образом: «Программы увеличиваются в размерах, стремясь заполнить всю память, доступную для их размещения».

Организация и управление основной (первичной, физической, реальной) памятью вычислительной машины - один из важнейших факторов, определяющих построение операционных систем.

Для непосредственного выполнения программ или обращения к данным необходимо, чтобы они размещались в основной памяти. Внешняя память имеет, как правило, гораздо большую емкость, чем основная, стоит дешевле и позволяет хранить данные и программы, которые должны быть наготове для обработки.

Существует большое количество разнообразных алгоритмов управления памятью. В данной работе будет рассмотрен алгоритм с использованием связных списков и предложена его программная реализация.

# Теоретическая часть

## Основные задания управления памятью

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Распределению подлежит вся оперативная память, не занятая операционной системой. Функциями ОС по управлению памятью являются: отслеживание свободной и занятой памяти, выделение памяти процессам и освобождение памяти при завершении процессов, вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место, а также настройка адресов программы на конкретную область физической памяти.

С появлением мультипрограммирования задачи ОС, связанные с распределением имеющейся памяти между несколькими одновременно выполняющимися программами, существенно усложнились.

Функциями ОС по управлению памятью в мультипрограммных системах являются:

* отслеживание (учет) свободной и занятой памяти;
* первоначальное и динамическое выделение памяти процессам приложений и самой операционной системе, и освобождение памяти по завершении процессов;
* настройка адресов программы на конкретную область физической памяти;
* полное или частичное вытеснение кодов и данных процессов из ОП на диск, когда размеры ОП недостаточны для размещения всех процессов, и возвращение их в ОП;
* защита памяти, выделенной процессу, от возможных вмешательств со стороны других процессов;
* дефрагментация памяти.

## Связный список

Связный список — базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Виды линейных связных списков:

* Линейный однонаправленный список
* Двунаправленный связный список

Линейный однонаправленный список — это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей. Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на NULL. Элемент, на который нет указателя, является первым (головным) элементом списка. Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

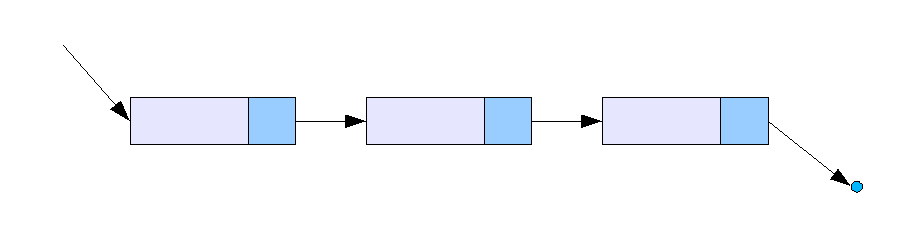


Рис. 1. Линейный однонаправленный список

Характеристики:

* Длина списка. Количество элементов в списке.

Списки могут быть типизированными или нетипизированными. Если список типизирован, то тип его элементов задан, и все его элементы должны иметь типы, совместимые с заданным типом элементов списка. Обычно списки типизированы.

* Список может быть сортированным или несортированным.

В зависимости от реализации может быть возможен произвольный доступ к элементам списка.

Двусвязный список (двунаправленный связный список) - ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом дает возможность перемещения в обе стороны. В этом списке проще производить удаление и перестановку элементов, так как легко доступны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

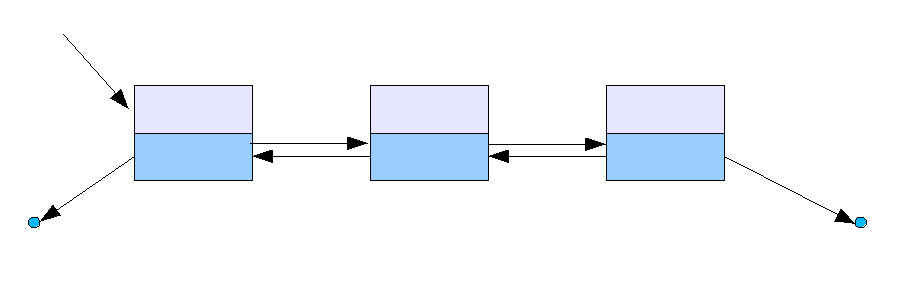


Рис. 2. Двунаправленный связный список

## Управление свободной памятью

Если память распределяется в динамическом режиме, то управлять этим должна операционная система. В общих чертах, существуют два способа отслеживания использования памяти: битовые матрицы и списки свободного пространства.

Управление памятью с помощью битовых матриц

При использовании битовых матриц память делится на единичные блоки размером от нескольких слов до нескольких килобайт. С каждым единичным блоком соотносится один бит в битовой матрице, который содержит 0, если единичный блок свободен, и 1, если он занят (или наоборот). На рис. 3 показаны часть памяти и соответствующая ей битовая матрица.



Важным вопросом для разработчика является размер единичного блока памяти. Чем меньше блок, тем больше битовая матрица. Но даже с таким небольшим единичным блоком памяти, размер которого равен 4 байта, для 32 бит памяти понадобится 1 бит матрицы. Память, состоящая из 32n бит, будет использовать n бит матрицы, таким образом, битовая матрица займет лишь 1/32 памяти. Если выбран более объемный единичный блок памяти, битовая матрица будет меньше, но тогда в последнем блоке процесса, если он не будет в точности кратен размеру единичного блока, будет впустую теряться довольно существенный объем памяти. Битовая матрица предоставляет довольно простой способ отслеживания слов памяти в фиксированном объеме памяти, поскольку ее размер зависит только от размера памяти и размера единичного блока памяти. Основная проблема заключается в том, что при решении поместить в память процесс, занимающий k единичных блоков, диспетчер памяти должен искать в битовой матрице непрерывную последовательность нулевых битов. Поиск в битовой матрице последовательности заданной длины — довольно медленная операция (поскольку последовательность может пересекать границы слов в матрице), и это обстоятельство служит аргументом против применения битовых матриц.

Управление памятью с помощью связанных списков

Другим способом отслеживания памяти является ведение связанных списков распределенных и свободных сегментов памяти, где сегмент либо содержит процесс, либо является пустым пространством между двумя процессами. Участок памяти, изображенный на рис. 3, *а*, представлен на рис. 3, *в*, как связанный список сегментов.



Каждая запись в списке хранит обозначение, содержит сегмент «дыру» — hole (H) или процесс — process (P), адрес, с которого сегмент начинается, его длину и указатель на следующую запись. В этом примере список сегментов составлен отсортированным по адресам. Преимущество такой сортировки заключается в том, что при завершении процесса или его свопинге на диск упрощается обновление списка. У завершившегося процесса есть, как правило, два соседа (за исключением тех случаев, когда он находился в верхних или нижних адресах памяти). Этими соседями могут быть либо процессы, либо пустые пространства, из чего можно составить четыре комбинации, показанные на Рис. 4, а обновление списка требует замены P на H. На рис. 4, б и в две записи объединяются в одну, и список становится на одну запись короче. На рис. 4, г объединяются три записи, и из списка удаляются уже две записи. Поскольку запись в таблице процессов, относящаяся к завершающемуся процессу, будет, как правило, указывать на запись в списке именно для этого процесса, возможно, удобнее будет вести не односвязный список, как показано на рис. 3, в, а двусвязный список. Такая структура облегчит поиск предыдущей записи и определение возможности объединения.

Когда процессы и пустые пространства содержатся в списке отсортированными по адресам, то для выделения памяти создаваемому процессу (или существующему процессу, загружаемому в результате свопинга с диска) могут быть использованы несколько алгоритмов. Предположим, что диспетчер памяти знает, сколько памяти нужно выделить. Простейший алгоритм называется «первое подходящее». Диспетчер памяти сканирует список сегментов до тех пор, пока не найдет пустое пространство подходящего размера. Затем пустое пространство разбивается на две части: одна для процесса и одна для неиспользуемой памяти, за исключением того статистически маловероятного случая, когда процесс в точности помещается в пустое пространство. «Первое подходящее» — это быстрый алгоритм, поскольку поиск ведется с наименьшими затратами времени. Незначительной вариацией алгоритма «первое подходящее» является алгоритм «следующее подходящее». Он работает так же, как и «первое подходящее», за исключением того, что отслеживает свое местоположение, как только находит подходящее пустое пространство. При следующем вызове для поиска пустого пространства он начинает поиск в списке с того места, на котором остановился в прошлый раз, а не приступает к поиску с самого начала, как при работе алгоритма «первое подходящее». Моделирование работы алгоритма «следующее подходящее», выполненное Бэйсом (Bays, 1977), показало, что его производительность несколько хуже, чем алгоритма «первое подходящее». Другой хорошо известный и широко используемый алгоритм — «наиболее подходящее». При нем поиск ведется по всему списку, от начала до конца, и выбирается наименьшее соответствующее пустое пространство. Вместо того чтобы разбивать большое пустое пространство, которое может пригодиться чуть позже, алгоритм «наиболее подходящее» пытается подыскать пустое пространство, близкое по размеру к необходимому, чтобы наилучшим образом соответствовать запросу и имеющимся пустым пространствам. В качестве примера алгоритмов «первое подходящее» и «наиболее подходящее» вернемся к Рис. 3 Если необходимо пространство в два единичных блока, то, согласно алгоритму «первое подходящее», будет выделено пустое пространство, начинающееся с адреса 5, а согласно алгоритму «наиболее подходящее», будет выделено пустое пространство, начинающееся с адреса 18. Алгоритм «наиболее подходящее» работает медленнее, чем «первое подходящее», поскольку при каждом вызове он должен вести поиск по всему списку. Как ни странно, но его применение приводит к более расточительному использованию памяти, чем использование алгоритмов «первое подходящее» и «следующее подходящее», поскольку он стремится заполнить память, оставляя небольшие бесполезные пустые пространства. В среднем при использовании алгоритма «первое подходящее» образуются более протяженные пустые пространства. При попытке обойти проблему разбиения практически точно подходящих пространств памяти на память, отводимую под процесс, и небольшие пустые пространства можно прийти к идее алгоритма «наименее подходящее», то есть к неизменному выбору самого большого подходящего пустого пространства, чтобы вновь образующееся пустое пространство было достаточно большим для дальнейшего использования. Моделирование показало, что применение алгоритма «наименее подходящее» также далеко не самая лучшая идея. Работа всех четырех алгоритмов может быть ускорена за счет ведения отдельных списков для процессов и пустых пространств. При этом все усилия этих алгоритмов сосредоточиваются на просмотре списков пустых пространств, а не списков процессов. Неизбежной ценой за это ускорение распределения памяти становится дополнительное усложнение и замедление процедуры ее освобождения, поскольку освободившийся сегмент должен быть удален из списка процессов и внесен в список пустых пространств. Если для процессов и пустых пространств ведутся разные списки, то для более быстрого обнаружения наиболее подходящих свободных мест список пустых пространств должен сортироваться по размеру. Когда при работе алгоритма «наиболее подходящее» поиск пустых пространств ведется от самых маленьких до самых больших, то при обнаружении подходящего пространства становится понятно, что найденное пространство является наименьшим, в котором может быть выполнено задание, следовательно, оно и есть наиболее подходящее. Дальнейший поиск, как при системе, использующей один список, уже не потребуется. При использовании списка пустых пространств, отсортированного по размеру, алгоритмы «первое подходящее» и «наиболее подходящее» работают с одинаковой скоростью, а алгоритм «следующее подходящее» теряет смысл. Когда список пустых пространств ведется отдельно от списка процессов, можно провести небольшую оптимизацию. Вместо того чтобы создавать отдельный набор структур данных для обслуживания списка пустых пространств, как это сделано на рис. 3, в, можно хранить информацию в самих пустых пространствах. В первом слове каждого пустого пространства может содержаться размер этого пространства, а второе слово может служить указателем на следующую запись. Элементы списка, показанного на рис. 3, в, для которых требуются три слова и один бит (P/H), больше не нужны. Еще один алгоритм распределения памяти называется «быстро искомое подходящее». Его использование предусматривает ведение отдельных списков для некоторых наиболее востребованных искомых размеров. К примеру, у него может быть таблица из n записей, в которой первая запись является указателем на вершину списка пустых пространств размером 4 Кбайт, вторая — указателем на список пустых пространств размером 8 Кбайт, третья — указателем на список пустых пространств размером 12 Кбайт и т. д. Пустые пространства размером, скажем, 21 Кбайт могут быть помещены либо в список пустых пространств размером 20 Кбайт, либо в специальный список пустых пространств с нечетным размером. При использовании алгоритма «быстро искомое подходящее» поиск пустого пространства требуемого размера выполняется исключительно быстро, но в нем имеется недостаток, присущий всем системам, сортирующим пустые пространства по размеру, а именно когда процесс завершается или выгружается процедурой свопинга, слишком много времени тратится на то, чтобы определить, можно ли высвобождаемое пространство объединить с соседними. Если не проводить объединение, то память очень быстро окажется разбитой на большое количество небольших по размеру пустых фрагментов, в которых не смогут поместиться процессы.

# Практическая часть

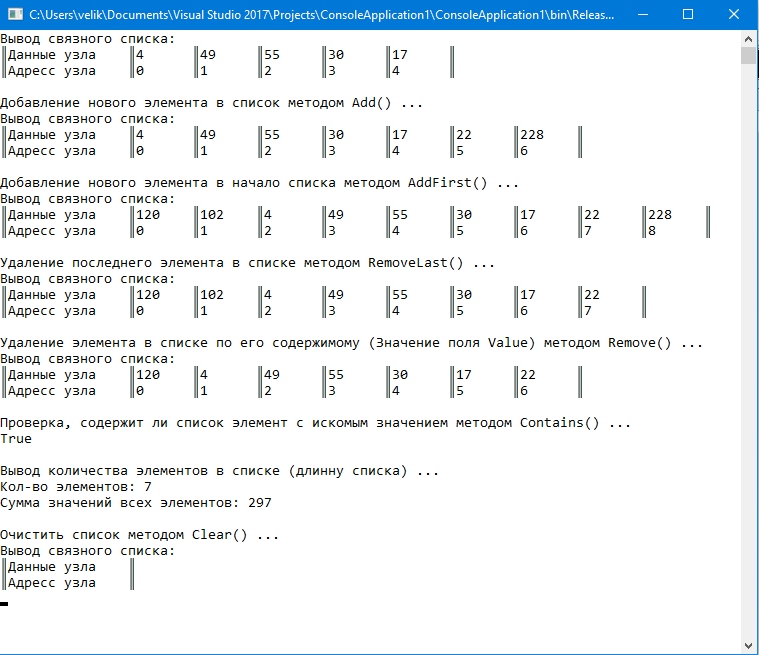
## Программная реализация алгоритма двунаправленного связного списка

Данный алгоритм реализован на языке программирования C#

В показанном варианте связный список изначально состоит из 5 узлов.

Наблюдающему показан список после каждой манипуляции с ним.

**Результат работы программы:**



**Листинг программы:**

**using** System**;**

**using** System**.**Collections**.**Generic**;**

**using** System**.**Linq**;**

**using** System**.**Text**;**

**using** System**.**Threading**.**Tasks**;**

**using** System**.**Collections**;**

**namespace** ConsoleApplication1

**{**

**public** class LinkedListNode**<**T**>**

**{**

///

/// Конструктор нового узла со значением Value.

///

///

**public** LinkedListNode**(**T **value)**

**{**

Value **=** **value;**

**}**

///

/// Поле Value.

///

**public** T Value **{** **get;** **internal** **set;** **}**

///

/// Ссылка на следующий узел списка (если узел последний, то null).

///

**public** LinkedListNode**<**T**>** Next **{** **get;** **internal** **set;** **}**

///

/// Ссылка на предыдущий узел списка (если узел первый, то null).

///

**public** LinkedListNode**<**T**>** Previous **{** **get;** **internal** **set;** **}**

**}**

**public** class LinkedList**<**T**>** **:**

System**.**Collections**.**Generic**.**ICollection**<**T**>**

**{**

**private** LinkedListNode**<**T**>** \_head**;**

**private** LinkedListNode**<**T**>** \_tail**;**

**public** LinkedListNode**<**T**>** Head

**{**

**get**

**{**

**return** \_head**;**

**}**

**}**

**public** LinkedListNode**<**T**>** Tail

**{**

**get**

**{**

**return** \_tail**;**

**}**

**}**

**public** void Add**(**T **value)**

**{**

AddLast**(value);**

**}**

**public** void AddFirst**(**T **value)**

**{**

LinkedListNode**<**T**>** node **=** **new** LinkedListNode**<**T**>(value);**

// Сохраняем ссылку на первый элемент.

LinkedListNode**<**T**>** temp **=** \_head**;**

// \_head указывает на новый узел.

\_head **=** node**;**

// Вставляем список позади первого элемента.

\_head**.**Next **=** temp**;**

**if** **(**Count **==** 0**)**

**{**

// Если список был пуст, то head and tail должны

// указывать на новой узел.

\_tail **=** \_head**;**

**}**

**else**

**{**

// До: head -------> 5 7 -> null

// После: head -> 3 5 7 -> null

temp**.**Previous **=** \_head**;**

**}**

Count**++;**

**}**

**public** void AddLast**(**T **value)**

**{**

LinkedListNode**<**T**>** node **=** **new** LinkedListNode**<**T**>(value);**

**if** **(**Count **==** 0**)**

**{**

\_head **=** node**;**

**}**

**else**

**{**

\_tail**.**Next **=** node**;**

// До: Head -> 3 5 -> null

// После:Head -> 3 5 7 -> null

// 7.Previous = 5

node**.**Previous **=** \_tail**;**

**}**

\_tail **=** node**;**

Count**++;**

**}**

**public** void RemoveFirst**()**

**{**

**if** **(**Count **!=** 0**)**

**{**

// До: Head -> 3 5

// После: Head -------> 5

// Head -> 3 -> null

// Head ------> null

\_head **=** \_head**.**Next**;**

Count**--;**

**if** **(**Count **==** 0**)**

**{**

\_tail **=** **null;**

**}**

**else**

**{**

// 5.Previous было 3; теперь null.

\_head**.**Previous **=** **null;**

**}**

**}**

**}**

**public** void RemoveLast**()**

**{**

**if** **(**Count **!=** 0**)**

**{**

**if** **(**Count **==** 1**)**

**{**

\_head **=** **null;**

\_tail **=** **null;**

**}**

**else**

**{**

// До: Head --> 3 --> 5 --> 7

// Tail = 7

// После: Head --> 3 --> 5 --> null

// Tail = 5

// Обнуляем 5.Next

\_tail**.**Previous**.**Next **=** **null;**

\_tail **=** \_tail**.**Previous**;**

**}**

Count**--;**

**}**

**}**

**public** void Clear**()**

**{**

\_head **=** **null;**

\_tail **=** **null;**

Count **=** 0**;**

**}**

**public** bool Contains**(**T item**)**

**{**

LinkedListNode**<**T**>** current **=** \_head**;**

**while** **(**current **!=** **null)**

**{**

**if** **(**current**.**Value**.**Equals**(**item**))**

**{**

**return** **true;**

**}**

current **=** current**.**Next**;**

**}**

**return** **false;**

**}**

**public** void CopyTo**(**T**[]** array**,** int arrayIndex**)**

**{**

LinkedListNode**<**T**>** current **=** \_head**;**

**while** **(**current **!=** **null)**

**{**

array**[**arrayIndex**++]** **=** current**.**Value**;**

current **=** current**.**Next**;**

**}**

**}**

**public** int Count

**{**

**get;**

**private** **set;**

**}**

**public** bool IsReadOnly

**{**

**get** **{** **return** **false;** **}**

**}**

**public** bool Remove**(**T item**)**

**{**

LinkedListNode**<**T**>** previous **=** **null;**

LinkedListNode**<**T**>** current **=** \_head**;**

// 1: Пустой список: ничего не делать.

// 2: Один элемент: установить Previous = null.

// 3: Несколько элементов:

// a: Удаляемый элемент первый.

// b: Удаляемый элемент в середине или конце.

**while** **(**current **!=** **null)**

**{**

// Head -> 3 -> 5 -> 7 -> null

// Head -> 3 ------> 7 -> null

**if** **(**current**.**Value**.**Equals**(**item**))**

**{**

// Узел в середине или в конце.

**if** **(**previous **!=** **null)**

**{**

// Случай 3b.

previous**.**Next **=** current**.**Next**;**

// Если в конце, то меняем \_tail.

**if** **(**current**.**Next **==** **null)**

**{**

\_tail **=** previous**;**

**}**

**else**

**{**

// До: Head -> 3 5 7 -> null

// После: Head -> 3 7 -> null

// previous = 3

// current = 5

// current.Next = 7

// Значит... 7.Previous = 3

current**.**Next**.**Previous **=** previous**;**

**}**

Count**--;**

**}**

**else**

**{**

// Случай 2 или 3a.

RemoveFirst**();**

**}**

**return** **true;**

**}**

previous **=** current**;**

current **=** current**.**Next**;**

**}**

**return** **false;**

**}**

**public** System**.**Collections**.**Generic**.**IEnumerator**<**T**>** GetEnumerator**()**

**{**

LinkedListNode**<**T**>** current **=** \_head**;**

**while** **(**current **!=** **null)**

**{**

**yield** **return** current**.**Value**;**

current **=** current**.**Next**;**

**}**

**}**

System**.**Collections**.**IEnumerator IEnumerable**.**GetEnumerator**()**

**{**

**return** **((**IEnumerable**)this).**GetEnumerator**();**

**}**

**public** void PrintList**()**

**{**

LinkedListNode**<**T**>** current **=** \_head**;**

int i **=** 0**;**

Console**.**Write**(**"Вывод связного списка:\n"**);**

Console**.**Write**(**"║Данные узла\t║"**);**

**while** **(**current **!=** **null)**

**{**

Console**.**Write**(**current**.**Value **+** "\t║"**);**

current **=** current**.**Next**;**

**}**

Console**.**Write**(**"\n║Адресс узла\t║"**);**

**while** **(**i **!=** Count**)** **{**

Console**.**Write**(**i **+** "\t║"**);**

i**++;**

**}**

Console**.**Write**(**"\n"**);**

**}**

**}**

class Program

**{**

static void Main**(**string**[]** args**)**

**{**

// Создаём новый связный список и инициализируем переменные

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

int i **=** 0**;**

Random rand **=** **new** Random**();** // генератор случайных чисел

// Заполняем список случайными числами

**while** **(**i **<** 5**)**

**{**

list**.**Add**(**rand**.**Next**(**0**,**100**));**

i**++;**

**}**

list**.**PrintList**();**

// Добавление нового элемента в список

Console**.**WriteLine**(**"\nДобавление нового элемента в список методом Add() ..."**);**

list**.**Add**(**22**);**

list**.**Add**(**228**);**

list**.**PrintList**();**

// Добавление нового элемента в начало списка

Console**.**WriteLine**(**"\nДобавление нового элемента в начало списка методом AddFirst() ..."**);**

list**.**AddFirst**(**102**);**

list**.**AddFirst**(**120**);**

list**.**PrintList**();**

// Удаление последнего элемента в списке

Console**.**WriteLine**(**"\nУдаление последнего элемента в списке методом RemoveLast() ..."**);**

list**.**RemoveLast**();**

list**.**PrintList**();**

// Удаление элемента в списке по его содержимому (Значение поля Value)

Console**.**WriteLine**(**"\nУдаление элемента в списке по его содержимому (Значение поля Value) методом Remove() ..."**);**

list**.**Remove**(**102**);**

list**.**PrintList**();**

// Проверка, содержит ли список элемент с искомым значением методом Contains()

Console**.**WriteLine**(**"\nПроверка, содержит ли список элемент с искомым значением методом Contains() ..."**);**

Console**.**WriteLine**(**list**.**Contains**(**22**));**

// Вывод количества элементов в списке (длинну списка)

Console**.**WriteLine**(**"\nВывод количества элементов в списке (длинну списка) ..."**);**

Console**.**WriteLine**(**"Кол-во элементов: " **+** list**.**Count**);**

// Проход по списку с конца и нахождение суммы всех элементов

LinkedListNode**<**int**>** current **=** list**.**Tail**;**

int Sum **=** 0**;**

**while** **(**current **!=** **null)**

**{**

Sum **+=** current**.**Value**;**

current **=** current**.**Previous**;**

**}**

Console**.**WriteLine**(**"Сумма значений всех элементов: " **+** Sum**);**

// Очистить список

Console**.**WriteLine**(**"\nОчистить список методом Clear() ..."**);**

list**.**Clear**();**

list**.**PrintList**();**

//Задержка на экране

Console**.**ReadKey**();**

**}**

**}**

**}**

# Вывод

В работе были рассмотрены основные задачи управления памятью, способы управления свободной памятью, связные списки.

Рассмотрен алгоритм управления свободной памятью с помощью связных списков, написана программа, реализующая этот алгоритм.

Получены знания о важности выбора правильного алгоритма управления памятью для быстродействия системы.

# Список литературы

1. Таненбаум Э. Современные операционные системы / Э. Таненбаум, Х. Бос. – СПб.: Питер, 2015. – 1120 с.
2. https://ru.wikipedia.org/wiki/Связный\_список
3. https://prog-cpp.ru/data-list/
4. http://khpi-iip.mipk.kharkiv.edu/library/datastr/book/prt05.html