Министерство образования и науки Российской Федерации

Калужский филиал
федерального государственного бюджетного образовательного
учреждения высшего образования
**«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»**
(КФ МГТУ им. Н.Э. Баумана)

С.С. Рыжов, Н.И. Пчелинцева

СТРУКТУРЫ ДАННЫХ

Учебное пособие по курсу «Типы и структуры данных»

Калуга - 2017

УДК 004.421
ББК 32.972.1
Р93

Учебное пособие составлено в соответствии с учебным планом КФ МГТУ им. Н.Э. Баумана по направлению подготовки 09.03.04 «Программная инженерия» кафедры «Программного обеспечения ЭВМ, информационных технологий и прикладной математики».

Методические указания рассмотрены и одобрены:

- Кафедрой «Программного обеспечения ЭВМ, информационных технологий и прикладной математики» (ФН1-КФ) протокол № 2 от «27» сентября 2017 г.

Зав. кафедрой ФН1-КФ _____ д.ф.-м.н., профессор Б.М. Логинов

- Методической комиссией факультета ФНК протокол № 5 от «2» октября 2017 г.

Председатель методической комиссии факультета ФНК _____ к.х.н., доцент К.Л. Анфилов

- Методической комиссией КФ МГТУ им.Н.Э. Баумана протокол № 1 от «03» 10 2017 г.

Председатель методической комиссии КФ МГТУ им.Н.Э. Баумана _____ д.э.н., профессор О.Л. Перерва

Рецензент:
к. т. н., доцент,
зав. каф. «Бизнес-информатика и информационные технологии» Калужского филиала Финансового университета при Правительстве Российской Федерации _____ С.В. Полпудников

Авторы
ведущий аналитик CorePartners Soft _____ С.С. Рыжов

к.т.н., доцент кафедры ФН1-КФ _____ Н.И. Пчелинцева

Аннотация

Учебное пособие «Структуры данных» по курсу «Типы и структуры данных» в рамках самостоятельной работы студентов содержат описание базовых структур данных, описание основных алгоритмов, применимых для рассмотренных структур, и их асимптотическая сложность, примеры реализации алгоритмов на языке программирования C#. Предназначены для студентов 2-го курса бакалавриата КФ МГТУ им. Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В настоящем учебном пособии рассматриваются базовые структуры данных, используемые при разработке программного обеспечения. Пособие состоит из 8 глав, в которых рассматриваются следующие структуры данных: списки, связные списки, двусвязные списки, стеки, очереди, множества, деревья и карты (англ. – Map).

Структура данных — программная единица, позволяющая хранить и обрабатывать множество однотипных или логически связанных данных. Для добавления, поиска, изменения и удаления данных структура данных предоставляет определенный набор функций.

Структуры данных формируются с помощью типов данных, ссылок и операций над ними в зависимости от конкретной реализации в разных языках программирования. Выбор структуры данных совершается в зависимости от потребностей разработчика, реализующего программное обеспечение. Ввиду этого, необходимо четко понимать особенности и нюансы работы различных структур данных. В данном пособии рассматриваются, в том числе подвиды базовых структур: двусторонняя очередь, двусвязный список.

Несмотря на бурное развитие технологий и вычислительных мощностей, вопрос построения высокопроизводительных программных средств остается актуальным. В пособии для каждой структуры данных приведены ее основные методы и указана средняя асимптотическая сложность их алгоритмов в «Big O» нотации.

Пособие предназначено для студентов 2-го курса бакалавриата КФ МГТУ им Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

ГЛАВА 1. СПИСОК

ArrayList — это коллекция, которая реализует интерфейс IList<T> и использует массив для хранения элементов. Как и связный список, ArrayList может хранить произвольное число элементов (ограниченное только объемом доступной памяти), но в остальном ведет себя как массив.

Интерфейс IList<T> предоставляет все методы ICollection<T> и дополнительно — методы для чтения, вставки и удаления элементов по индексу. Код ниже сгенерирован с помощью команды «Implement Interface» в MS Visual Studio и, кроме автоматически сгенерированных заглушек для методов, содержит также:

- Массив из T (_items) для хранения элементов
- Конструктор по умолчанию, который создает пустой список
- Конструктор, принимающий целое число, который создает список с заданной вместимостью. Заметьте, что вместимость списка и его длина — это не одно и то же. На практике может встретиться ситуация, когда такой конструктор позволит пользователю избежать большого количества расширений внутреннего массива.

```
1  public class ArrayList :
 2 System.Collections.Generic.IList
 3 {
 4     T[] _items;
 5
 6     public ArrayList()
 7         : this(0)
 8     {
 9     }
10
11     public ArrayList(int length)
12     {
13         if (length < 0)
```

```
14          {
15              throw new ArgumentException("length");
16          }
17
18          _items = new T[length];
19      }
20
21      public int IndexOf(T item)
22      {
23          throw new NotImplementedException();
24      }
25
26      public void Insert(int index, T item)
27      {
28          throw new NotImplementedException();
29      }
30
31      public void RemoveAt(int index)
32      {
33          throw new NotImplementedException();
34      }
35
36      public T this[int index]
37      {
38          get
39          {
40              throw new NotImplementedException();
41          }
42          set
43          {
44              throw new NotImplementedException();
45          }
46      }
47
48      public void Add(T item)
49      {
50          throw new NotImplementedException();
51      }
52
53      public void Clear()
54      {
```

```
55          throw new NotImplementedException();
56      }
57
58      public bool Contains(T item)
59      {
60          throw new NotImplementedException();
61      }
62
63      public void CopyTo(T[] array, int arrayIndex)
64      {
65          throw new NotImplementedException();
66      }
67
68      public int Count
69      {
70          get { throw new NotImplementedException(); }
71      }
72
73      public bool IsReadOnly
74      {
75          get { throw new NotImplementedException(); }
76      }
77
78      public bool Remove(T item)
79      {
80          throw new NotImplementedException();
81      }
82
83      public System.Collections.Generic.IEnumerator
84  GetEnumerator()
85      {
86          throw new NotImplementedException();
87      }
88
89      System.Collections.IEnumerator
90  System.Collections.IEnumerable.GetEnumerator()
91      {
            throw new NotImplementedException();
        }
    }
```

1.1. Вставка элементов

Вставка элементов в динамический массив отличается от вставки в связный список. На это есть две причины: динамический массив поддерживает вставку в середину массива, тогда как в связный список можно вставлять только в конец или начало. Вставка элемента в связный список всегда выполняется за константное время. Вставка в динамический массив может занимать как O(1), так и O(n) времени.

По мере добавления элементов внутренний массив может переполниться. В этом случае необходимо сделать следующее:

1. Создать массив большего размера.
2. Скопировать элементы в новый массив.
3. Обновить ссылку на внутренний массив списка так, чтобы она указывала на новый.

Размер нового массива определяется стратегией роста динамического массива. Мы рассмотрим две стратегии и для обеих посмотрим, насколько быстро растет массив и насколько его рост снижает производительность.

Увеличение массива вдвое (подход Mono и Rotor):

Существуют две реализации ArrayList: Mono и Rotor. Обе используют простой алгоритм увеличения размера массива, увеличивая его вдвое при необходимости. Если изначальный размер массива равен нулю, то новый будет вмещать 16 элементов:

```
size = size == 0 ? 16 : size * 2;
```

Этот алгоритм делает меньше выделений памяти, но тратит больше места, чем вариант, принятый в Java. Другими словами, он обеспечивает более частые случаи вставки за константное время ценой использования большего количества памяти. Процесс создания нового массива и копирования в него элементов будет происходить реже.

Медленный рост (подход Java):

В Java используется похожий подход, но массив растет медленнее. Размер нового массива определяется следующим образом:

```
size = (size * 3) / 2 + 1;
```

При использовании этого алгоритма используется меньше памяти.

Какая стратегия лучше? Не существует правильного и неправильного ответа на этот вопрос. При использовании удвоения будет меньше операций вставки за O(n), но больше расход памяти. При более медленном росте будет использовано меньше памяти. В общем случае допустимы оба варианта. Если приложение имеет специфические требования, возможно, потребуется реализовать ту или иную стратегию расширения. В любом случае, внешнее поведение динамического массива не изменится.

Наша реализация будет использовать увеличение вдвое (подход Mono/Rotor):

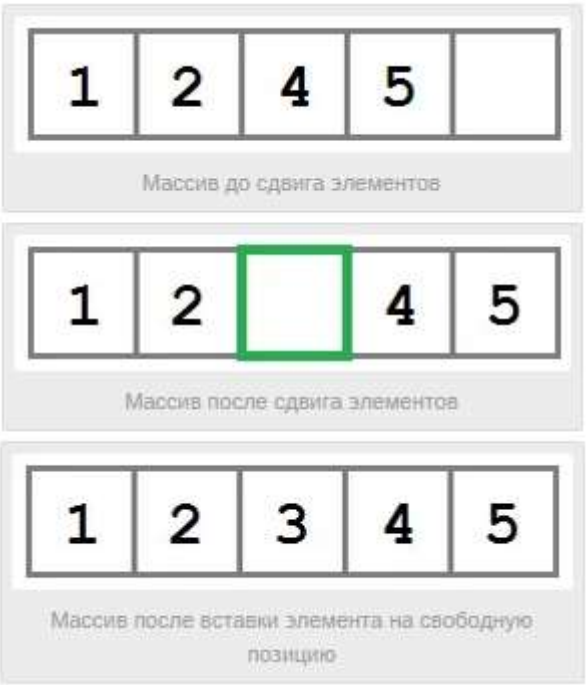
```
private void GrowArray()
{
    int newLength = _items.Length == 0 ? 16 : _items.Length * 2;

    T[] newArray = new T[newLength];

    _items.CopyTo(newArray, 0);

    _items = newArray;
}
```

1.2. Метод Insert

- Поведение: добавляет элемент по указанному индексу. Если индекс равен количеству элементов или больше него, генерирует исключение.
- Сложность: O(n).

Вставка по определенному индексу требует сдвига всех элементов, начиная с этого индекса, на одну позицию вправо. Если внутренний массив заполнен, вставка потребует увеличения его размера.

В следующем примере дается массив с пятью позициями, четыре из которых заполнены. Значение «3» вставляется на третью позицию (с индексом 2):




```
1  public void Insert(int index, T item)
 2 {
 3     if (index >= Count)
 4     {
 5         throw new IndexOutOfRangeException();
```

```
 6     }
 7
 8     if (_items.Length == this.Count)
 9     {
10         this.GrowArray();
11     }
12
13     Array.Copy(_items, index, _items, index + 1,
14 Count - index);
15
16     _items[index] = item;
17
18     Count++;
19 }
```

1.3. Метод Add

- Поведение: добавляет элемент в конец списка.
- Сложность: O(1), если осталось более одного свободного места; O(n), если необходимо расширение массива.

```
1 public void Add(T item)
2 {
3     if (_items.Length == Count)
4     {
5         GrowArray();
6     }
7
8     _items[Count++] = item;
9 }
```

1.4. Метод RemoveAt

- Поведение: удаляет элемент, расположенный по заданному индексу.

- Сложность: O(n).

Удаление элемента по индексу — операция, обратная вставке. Указанный элемент удаляется, а остальные сдвигаются на одну позицию влево.


```
1  public void RemoveAt(int index)
 2 {
 3     if (index &gt;= Count)
 4     {
 5         throw new IndexOutOfRangeException();
 6     }
 7
 8     int shiftStart = index + 1;
 9     if (shiftStart < Count)
10     {
11         Array.Copy(_items, shiftStart, _items, index,
12 Count - shiftStart);
13     }
14
15     Count--;
```

```
16 }
```

1.5. Метод Remove

- Поведение: удаляет первый элемент, значение которого равно предоставленному. Возвращает true, если элемент был удален, или false в противном случае.
- Сложность: O(n).

```
 1 public bool Remove(T item)
 2 {
 3     for (int i = 0; i < Count; i++)
 4     {
 5         if (_items[i].Equals(item))
 6         {
 7             RemoveAt(i);
 8             return true;
 9         }
10     }
11
12     return false;
13 }
```

1.6. Метод IndexOf

- Поведение: возвращает индекс первого элемента, значение которого равно предоставленному или -1, если такого значения нет.
- Сложность: O(n).

```
1  public int IndexOf(T item)
 2 {
 3     for (int i = 0; i < Count; i++)
 4     {
 5         if (_items[i].Equals(item))
 6         {
 7             return i;
```

```
 8          }
 9      }
10
11      return -1;
12  }
```

1.7. Метод Item

- Поведение: позволяет прочитать или изменить значение по индексу.
- Сложность: O(1).

```
1  public T this[int index]
 2 {
 3      get
 4      {
 5          if(index < Count)
 6          {
 7              return _items[index];
 8          }
 9
10          throw new IndexOutOfRangeException();
11      }
12      set
13      {
14          if (index < Count)
15          {
16              _items[index] = value;
17          }
18          else
19          {
20              throw new IndexOutOfRangeException();
21          }
22      }
23  }
```

1.8. Метод Contains

- Поведение: возвращает true, если значение есть в списке, и false в противном случае.
- Сложность: O(n).

```
1 public bool Contains(T item)
2 {
3     return IndexOf(item) != -1;
4 }
```

ГЛАВА 2. Связный список

Основное назначение связного списка — предоставление механизма для хранения и доступа к произвольному количеству данных. Как следует из названия, это достигается связыванием данных вместе в список.

Прежде чем мы перейдем к рассмотрению связного списка, вспомним, как хранятся данные в массиве.

Как показано на рисунке, данные в массиве хранятся в непрерывном участке памяти, разделенном на ячейки определенного размера. Доступ к данным в ячейках осуществляется по ссылке на их расположение — индексу.

Это отличный способ хранить данные. Большинство языков программирования позволяют так или иначе выделить память в виде массива и оперировать его содержимым. Последовательное хранение данных увеличивает производительность (data locality), позволяет легко итерироваться по содержимому и получать доступ к произвольному элементу по индексу.

Тем не менее, иногда массив — не самая подходящая структура. Предположим, что у нашей программы следующие требования:

- Прочесть некоторое количество целых чисел из источника (метод NextValue), пока не встретится определенное число.
- Передать считанные числа в метод ProcessItems

Поскольку в требованиях указано, что считанные числа передаются в метод ProcessItems за один раз, очевидным решение будет массив целых чисел:

```
1
2  void LoadData()
3  {
4      int[] values = new int[20];
5      for (int i = 0; i < values.Length; i++)
6      {
7          if (values[i] == 0xFFFF)
8          {
9              break;
10         }
11
12         values[i] = NextValue();
13     }
14
15     ProcessItems(values);
16 }
17
18 void ProcessItems(int[] values)
19 {
20
21 }
```

У этого решения есть ряд проблем, но самая очевидная из них — что случится, если будет необходимо прочесть больше 20 значений? В данной реализации значения с 21 и далее просто проигнорируются. Можно выделить больше памяти — 200 или 2000 элементов. Можно дать пользователю возможность

выбрать размер массива. Или выделить память под новый массив большего размера при заполнении старого и скопировать элементы. Но все эти решения усложняют код и бесполезно тратят память.

Нам нужна коллекция, которая позволяет добавить произвольное число элементов и перебрать их в порядке добавления. Размер коллекции должен быть неограничен, а произвольный доступ нам не нужен. Нам нужен связный список.

Прежде чем перейти к его реализации, посмотрим на то, как могло бы выглядеть решение нашей задачи.

```
1  static void LoadItems()
 2 {
 3     LinkedList list = new LinkedList();
 4     while (true)
 5     {
 6         int value = NextValue();
 7         if (value != 0xFFFF)
 8         {
 9             list.Add(value);
10         }
11         else
12         {
13             break;
14         }
15     }
16
17     ProcessItems(list);
18 }
19
20 static void ProcessItems(LinkedList list)
21 {
22
23 }
```

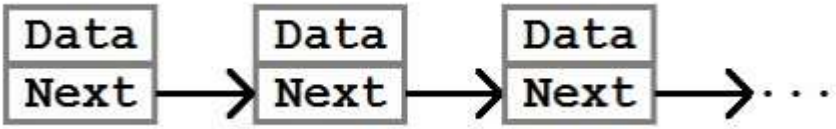
Обратите внимание: проблем, присущих первому варианту решения больше нет — мы не можем выделить недостаточно или, наоборот, слишком много памяти под массив.

Кроме того, из этого кода можно увидеть, что список будет принимать параметр типа <T> и реализовывать интерфейс IEnumerable

2.1. Реализация класса LinkedList

Класс Node

В основе связного списка лежит понятие узла, или элемента (Node). Узел — это контейнер, который позволяет хранить данные и получать следующий узел.



В самом простом случае класс Node можно реализовать так:

```
1 public class Node
2 {
3     public int Value { get; set; }
4     public Node Next { get; set; }
5 }
```

Теперь мы можем создать примитивный связный список. Выделим память под три узла (first, middle, last) и соединим их последовательно:

```
1 Node first = new Node { Value = 3 };
2 Node middle = new Node { Value = 5 };
3 first.Next = middle;
4 Node last = new Node { Value = 7 };
5 middle.Next = last;
```

Теперь у нас есть список из трех элементов, начиная с first и заканчивая last. Поле Next последнего узла имеет значение null, что показывает, что это – последний элемент. С

этим списком уже можно производить различные операции. Например, напечатать данные из каждого элемента:

```
private static void PrintList(Node node)
{
    while (node != null)
    {
        Console.WriteLine(node.Value);
        node = node.Next;
    }
}
```

Метод PrintList итерируется по элементам списка: печатает значение поля Value и переходит к следующему узлу по ссылке в поле Next.

Теперь, когда мы знаем, как должен выглядеть узел связанного списка, рассмотрим реализацию класса LinkedListNode.

```
public class LinkedListNode
{
    public LinkedListNode(T value)
    {
        Value = value;
    }

    public T Value { get;  internal set; }

    public LinkedListNode Next { get;  internal set;
}
}
```

Класс LinkedList

Прежде чем реализовывать наш связный список, нужно понять, как мы будем с ним работать.

Ранее мы увидели, что коллекция должна поддерживать любой тип данных, а значит, нам нужно реализовать обобщенный интерфейс.

Поскольку мы используем платформу .NET, имеет смысл реализовать наш класс таким образом, чтобы его поведение было похоже на поведение встроенных коллекций. Самый простой способ сделать это — реализовать интерфейс ICollection<T>. Заметьте, что мы реализуем ICollection<T>, а не Ilist<T>, поскольку интерфейс Ilist<T> позволяет получать доступ к элементам по индексу. Несмотря на то, что произвольный доступ к элементам в целом полезен, его невозможно эффективно реализовать в связном списке.

Учитывая все вышесказанное структура класса будет выглядеть следующим образом:

```
1  public class LinkedList :
2      System.Collections.Generic.ICollection
3 {
4      public void Add(T item)
5      {
6          throw new System.NotImplementedException();
7      }
8
9      public void Clear()
10     {
11         throw new System.NotImplementedException();
12     }
13
14     public bool Contains(T item)
15     {
16         throw new System.NotImplementedException();
17     }
18
19     public void CopyTo(T[] array, int arrayIndex)
20     {
21         throw new System.NotImplementedException();
22     }
23
24     public int Count
25     {
```

```
26          get;
27          private set;
28      }
29
30      public bool IsReadOnly
31      {
32          get { throw new
33  System.NotImplementedException(); }
34      }
35
36      public bool Remove(T item)
37      {
38          throw new System.NotImplementedException();
39      }
40
41      public System.Collections.Generic.IEnumerator
42  GetEnumerator()
43      {
44          throw new System.NotImplementedException();
45      }
46
47      System.Collections.IEnumerator
48  System.Collections.IEnumerable.GetEnumerator()
49      {
            throw new System.NotImplementedException();
        }
    }
```

2.2. Метод Add

- Поведение: Добавляет элемент в конец списка.
- Сложность: O(1)

Добавление элемента в связный список производится в три этапа:

1. Создать экземпляр класса LinkedListNode.
2. Найти последний узел списка.
3. Установить значение поля Next последнего узла списка так, чтобы оно указывало на созданный узел.

Основная сложность заключается в том, чтобы найти последний узел списка. Можно сделать это двумя способами. Первый — сохранять указатель на первый узел списка и перебирать узлы, пока не дойдем до последнего. В этом случае не требуется сохранять указатель на последний узел, что позволяет использовать меньше памяти (в зависимости от размера указателя на вашей платформе), но требует прохода по всему списку при каждом добавлении узла. Это значит, что метод Add займет O(n) времени.

Второй метод заключается в сохранении указателя на последний узел списка, и тогда при добавлении нового узла мы поменяем указатель так, чтобы он указывал на новый узел. Этот способ предпочтительней, поскольку выполняется за O(1) времени.

Первое, что необходимо сделать — добавить два приватных поля в класс LinkedList: ссылки на первый (head) и последний (tail) узлы.

```
1 private LinkedListNode _head;
2 private LinkedListNode _tail;
```

Теперь мы можем добавить метод, который выполняет три необходимых шага.

```
1  public void Add(T value)
 2 {
 3     LinkedListNode node = new LinkedListNode(value);
 4
 5     if (_head == null)
 6     {
 7         _head = node;
 8         _tail = node;
 9     }
10     else
11     {
12         _tail.Next = node;
13         _tail = node;
14     }
```

```
15
16      Count++;
17 }
```

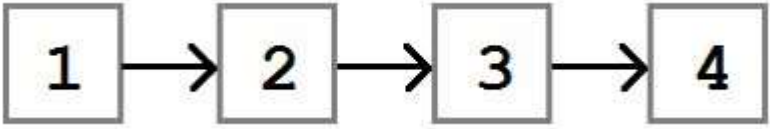
Сначала мы создаем экземпляр класса LinkedListNode. Затем проверяем, является ли список пустым. Если список пуст, мы просто устанавливаем значения полей _head и_tail так, чтобы они указывали на новый узел. Этот узел в данном случае будет являться одновременно и первым, и последним в списке. Если список не пуст, узел добавляется в конец списка, а поле _tail теперь указывает на новый конец списка.

Поле Count инкрементируется при добавлении узла для того, чтобы сохранялся контракт интерфейса ICollection<T>. Поле Count возвращает точное количество элементов списка.

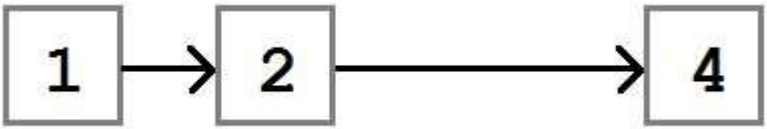
2.3.Каскадная модель

- Поведение: Удаляет первый элемент списка со значением, равным переданному. Возвращает true, если элемент был удален и false в противном случае.
- Сложность: O(n)

Прежде чем разбирать метод Remove, посмотрим, чего мы хотим добиться. На следующем рисунке список с четырьмя элементами. Мы удаляем элемент со значением «3».



После удаления узла поле Next узла со значением «2» будет указывать на узел со значением «4».



Основной алгоритм удаления элемента такой:

1. Найти узел, который необходимо удалить
2. Изменить значение поля Next предыдущего узла так, чтобы оно указывало на узел, следующий за удаляемым

Некоторые из случаев, которые необходимо предусмотреть:

- Список может быть пустым, или значение, которое мы передаем в метод может не присутствовать в списке. В этом случает список останется без изменений
- Удаляемый узел может быть единственным в списке. В этом случае мы установим значения полей _head и _tail равными null
- Удаляемый узел будет в начале списка. В этом случае мы записываем в _head ссылку на следующий узел
- Удаляемый узел будет в середине списка
- Удаляемый узел будет в конце списка. В этом случае мы записываем в _tail ссылку на предпоследний узел, а в его поле Next записываем null

```
 1 public bool Remove(T item)
 2 {
 3     LinkedListNode previous = null;
 4     LinkedListNode current = _head;
 5 
 6     while (current != null)
 7     {
 8         if (current.Value.Equals(item))
 9         {
10             if (previous != null)
```

```
11              {
12                  previous.Next = current.Next;
13                  if (current.Next == null)
14                  {
15                      _tail = previous;
16                  }
17              }
18              else
19              {
20                  _head = _head.Next;
21
22                  // Список теперь пустой?
23                  if (_head == null)
24                  {
25                      _tail = null;
26                  }
27              }
28
29              Count--;
30
31              return true;
32          }
33
34          previous = current;
35          current = current.Next;
36      }
37
38      return false;
39  }
```

Поле Count декрементируется при удалении узла.

2.4. Метод Contains

- Поведение: Возвращает true или false в зависимости от того, присутствует ли искомый элемент в списке.
- Сложность: O(n)

Метод Contains достаточно простой. Он просматривает каждый элемент списка, от первого до последнего, и возвращает true как только найдет узел, чье значение равно

переданному параметру. Если такой узел не найден, и метод дошел до конца списка, то возвращается false.

```
public bool Contains(T item)
{
    LinkedListNode current = _head;
    while (current != null)
    {
        if (current.Value.Equals(item))
        {
            return true;
        }

        current = current.Next;
    }

    return false;
}
```

2.5. Метод GetEnumerator

- Поведение: Возвращает экземпляр IEnumerator, который позволяет итерироваться по элементам списка.
- Сложность: Получение итератора — O(1). Проход по всем элементам — O(n).

Возвращаемый итератор проходит по всему списку от первого до последнего узла и возвращает значение каждого элемента с помощью ключевого слова yield.

```
IEnumerator IEnumerable.GetEnumerator()
{
    LinkedListNode current = _head;
    while (current != null)
    {
        yield return current.Value;
        current = current.Next;
    }
}

```

```
11 IEnumerator IEnumerable.GetEnumerator()
12 {
13     return ((IEnumerable)this).GetEnumerator();
14 }
```

2.6. Метод Clear

- Поведение: Удаляет все элементы из списка.
- Сложность: O(1)

Метод Clear устанавливает значения полей _head и _tail равными null. Поскольку C# — язык с автоматическим управлением памятью, нет необходимости явно удалять неиспользуемые узлы. Клиент, вызывающий метод, должен убедиться в корректном удалении значений узлов, если это необходимо.

```
1 public void Clear()
2 {
3     _head = null;
4     _tail = null;
5     Count = 0;
6 }
```

2.7. Метод CopyTo

- Поведение: Копирует содержимое списка в указанный массив, начиная с указанного индекса.
- Сложность: O(n)

Метод CopyTo проходит по списку и копирует элементы в массив с помощью присваивания. Клиент, вызывающий метод должен убедиться, что массив имеет достаточный размер для того, чтобы вместить все элементы списка.

```
1 public void CopyTo(T[] array, int arrayIndex)
2 {
3     LinkedListNode current = _head;
4     while (current != null)
5     {
6         array[arrayIndex++] = current.Value;
```

```
7           current = current.Next;
8       }
9   }
```

2.8. Метод Count

- Поведение: Возвращает количество элементов списка. Возвращает 0, если список пустой.
- Сложность: O(1)

Count — поле с публичным геттером и приватным сеттером. Изменение его значения осуществляется в методах Add, Remove и Clear.

```
1 public int Count
2 {
3     get;
4     private set;
5 }
```

2.9. Метод IsReadOnly

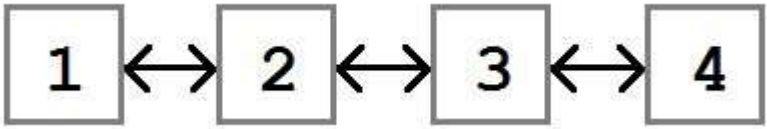
- Поведение: Возвращает false, если список только для чтения.
- Сложность: O(1)

```
1 public bool IsReadOnly
2 {
3     get { return false; }
4 }
```

ГЛАВА 3. Двусвязный список

Связный список, рассмотренный в предыдущей главе, называется также «односвязным». Это значит, что между узлами есть только одна связь в единственном направлении от первого узла к последнему. Также существует распространенный вариант списка, который предоставляет доступ к обоим концам — двусвязный список.

Для того, чтобы создать двусвязный список, необходимо добавить в класс LinkedListNode поле Previous, которое будет содержать ссылку на предыдущий элемент списка.



Далее, рассмотрим только отличия в реализации односвязного и двусвязного списка.

Класс Node

Единственное изменение, которое надо внести в класс LinkedListNode — добавить поле со ссылкой на предыдущий узел.

```
 1 public class LinkedListNode
 2 {
 3     public LinkedListNode(T value)
 4     {
 5         Value = value;
 6     }
 7
 8     public T Value { get;  internal set; }
 9
10     public LinkedListNode Next { get; internal set; }
11
12     public LinkedListNode Previous { get;  internal
13 set; }
   }
```

3.1. Метод AddFirst

В то время, как односвязный список позволяет добавлять элементы только в конец, используя двусвязный список мы можем добавлять элементы как в начало, так и в конец, с помощью методов AddFirst и AddLast соответственно. Метод ICollection<T>.Add будет вызывать AddLast для совместимости с односвязным списком.

- Поведение: Добавляет переданный элемент в начало списка.
- Сложность: O(1)

При добавлении элемента в начало списка последовательность действий примерно такая же, как и при добавлении элемента в односвязный список.

1. Установить значение поля Next в новом узле так, чтобы оно указывало на бывший первый узел.
2. Установить значение поля Previous в бывшем первом узле так, чтобы оно указывало на новый узел.
3. Обновить поле _tail при необходимости и инкрементировать поле Count

```
1  public void AddFirst(T value)
 2 {
 3     LinkedListNode node = new LinkedListNode(value);
 4
 5     LinkedListNode temp = _head;
 6
 7     _head = node;
 8
 9     _head.Next = temp;
10
11     if (Count == 0)
12     {
13         _tail = _head;
14     }
15     else
16     {
17         temp.Previous = _head;
18     }
19
20     Count++;
21 }
```

3.2. Метод AddLast

- Поведение: Добавляет переданный элемент в конец списка.
- Сложность: O(1)

Добавление узла в конец списка легче, чем в начало. Необходимо создать новый узел и обновить поля _head и _tail, а затем инкрементировать поле Count.

```
1
2
3  public void AddLast(T value)
4  {
5      LinkedListNode node = new LinkedListNode(value);
6
7      if (Count == 0)
8      {
9          _head = node;
10     }
11     else
12     {
13         _tail.Next = node;
14         node.Previous = _tail;
15     }
16
17     _tail = node;
18     Count++;
19 }
```

Как было замечено ранее, ICollection<T>. Add вызывает AddLast.

```
1 public void Add(T value)
2 {
3     AddLast(value);
4 }
```

3.3. Метод RemoveFirst

Как и метод Add, Remove будет разделен на два метода, позволяющих удалять элементы из начала и из конца списка. Метод ICollection<T>.Remove будет также удалять элементы из

начала, но теперь будет еще обновлять поля Previous в тех узлах, где это необходимо.

- Поведение: Удаляет первый элемент списка. Если список пуст, не делает ничего. Возвращает true, если элемент был удален и false в противном случае.
- Сложность: O(1)

RemoveFirst устанавливает ссылку head на второй узел списка и обнуляет поле Previous этого узла, удаляя таким образом все ссылки на предыдущий первый узел. Если список был пуст или содержал только один элемент, то поля _head и _tail становятся равны null.

```
public void RemoveFirst()
{
    if (Count != 0)
    {
        _head = _head.Next;

        Count--;

        if (Count == 0)
        {
            _tail = null;
        }
        else
        {
            _head.Previous = null;
        }
    }
}
```

3.4. Метод RemoveLast

- Поведение: Удаляет последний элемент списка. Если список пуст, не делает ничего. Возвращает true, если элемент был удален и false в противном случае.
- Сложность: O(1)

RemoveLast устанавливает значение поля _tail так, чтобы оно указывало на предпоследний элемент списка и, таким образом, удаляет последний элемент. Если список был пустым, или содержал только один элемент, то поля _head и _tail становятся равны null.

```
 1 public void RemoveLast()
 2 {
 3     if (Count != 0)
 4     {
 5         if (Count == 1)
 6         {
 7             _head = null;
 8             _tail = null;
 9         }
10         else
11         {
12             _tail.Previous.Next = null;
13             _tail = _tail.Previous;
14         }
15
16         Count--;
17     }
18 }
```

3.5. Метод Remove

- Поведение: Удаляет первый элемент списка со значением, равным переданному. Возвращает true, если элемент был удален и false в противном случае.
- Сложность: O(n)

Метод ICollection<T>.Remove() почти такой же, как и в односвязном списке. Единственное отличие — теперь необходимо поменять значение поля Previous при удалении узла. Для того, чтобы не повторять код, этот метод вызывает RemoveFirst при удалении первого узла.

```
1  public bool Remove(T item)
 2 {
```

```
 3      LinkedListNode previous = null;
 4      LinkedListNode current = _head;
 5
 6      while (current != null)
 7      {
 8          if (current.Value.Equals(item))
 9          {
10              if (previous != null)
11              {
12                  previous.Next = current.Next;
13                  if (current.Next == null)
14                  {
15                      _tail = previous;
16                  }
17                  else
18                  {
19                      current.Next.Previous = previous;
20                  }
21
22                  Count--;
23              }
24              else
25              {
26                  RemoveFirst();
27              }
28
29              return true;
30          }
31
32          previous = current;
33          current = current.Next;
34      }
35
36      return false;
37  }
```

Для чего нужен двусвязный список:

Итак, мы можем добавлять элементы в начало списка и в его конец. В том виде, в котором он реализован сейчас, нет особых преимуществ перед обычным односвязным списком. Но если добавить геттеры для полей head и tail, пользователь этой

реализации списка сможет использовать множество различных алгоритмов.

```
public LinkedListNode Head
{
    get
    {
        return _head;
    }
}

public LinkedListNode Tail
{
    get
    {
        return _tail;
    }
}
```

Так мы сможем итерироваться по списку вручную, в том числе от последнего элемента к первому.

В этом примере используются поля Tail и Previous для того, чтобы обойти список в обратном порядке.

```
public void ProcessListBackwards()
{
    LinkedList list = new LinkedList();
    PopulateList(list);

    LinkedListNode current = list.Tail;
    while (current != null)
    {
        ProcessNode(current);
        current = current.Previous;
    }
}
```

Кроме того, двусвязный список позволяет легко реализовать двусвязную очередь, которая, в свою очередь, является строительным блоком для других структур данных.

ГЛАВА 4. Стек

Стек — это коллекция, элементы которой выстроены по принципу «последний вошел, первый вышел» (Last-In-First-Out или LIFO). Это значит, что мы будем иметь доступ только к последнему добавленному элементу.

В отличие от списков, мы не можем получить доступ к произвольному элементу стека. Возможно только добавлять или удалять элементы с помощью специальных методов. У стека нет метода Contains, как у списков. Кроме того, у стека нет итератора. Для того, чтобы понимать, почему на стек накладываются такие ограничения, рассмотрим как он работает и как используется.

Наиболее часто встречающаяся аналогия для объяснения стека — стопка тарелок. Вне зависимости от того, сколько тарелок в стопке, мы всегда можем снять верхнюю. Чистые тарелки точно так же кладутся на верх стопки, и мы всегда будем первой брать ту тарелку, которая добавлена последней.

Если мы добавим, например, красную тарелку, затем синюю, а затем зеленую, то сначала надо будет снять зеленую, потом синюю, и, наконец, красную. Главное, что надо запомнить — тарелки всегда добавляются на верх стопки. Когда кто-то берет тарелку, он также снимает ее сверху. Получается, что тарелки разбираются в порядке, обратном тому, в котором ставились.

Теперь, введем несколько терминов. Операция добавления элемента на стек называется «push», удаления — «pop». Последний добавленный элемент называется верхушкой стека, или «top», и его можно посмотреть с помощью операции «peek». Рассмотрим заготовку класса, реализующего стек.

Класс Stack

Класс Stack отпределяет методы Push, Pop, Peek для доступа к элементам и поле Count. В реализации мы будем использовать LinkedList<T> для хранения элементов.

```
1  public class Stack
 2 {
 3     LinkedList _items = new LinkedList();
 4
 5     public void Push(T value)
 6     {
 7         throw new NotImplementedException();
 8     }
 9
10     public T Pop()
11     {
12         throw new NotImplementedException();
13     }
14
15     public T Peek()
16     {
17         throw new NotImplementedException();
18     }
19
20     public int Count
21     {
22         get;
23     }
24 }
```

4.1. Метод Push

- Поведение: Добавляет элемент на вершину стека.
- Сложность: O(1).

Поскольку мы используем связный список для хранения элементов, можно просто добавить новый в конец списка.

```
1 public void Push(T value)
2 {
3     _items.AddLast(value);
4 }
```

4.2. Метод Pop

- Поведение: Удаляет элемент с вершины стека и возвращает его. Если стек пустой, кидает InvalidOperationException.
- Сложность: O(1).

Push добавляет элементы в конец списка, поэтому забирать их будет также с конца. В случае, если список пуст, будет сгенерировано исключение.

```
1  public T Pop()
2  {
3      if (_items.Count == 0)
4      {
5          throw new InvalidOperationException("The
6  stack is empty");
7      }
8
9      T result = _items.Tail.Value;
10
11     _items.RemoveLast();
12
13     return result;
   }
```

4.3. Метод Peek

- Поведение: Возвращает верхний элемент стека, но не удаляет его. Если стек пустой, генерирует InvalidOperationException.
- Сложность: O(1).

```
public T Peek()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The stack
is empty");
    }

    return _items.Tail.Value;
}
```

4.4. Метод Count

- Поведение: Возвращает количество элементов в стеке.
- Сложность: O(1).

С помощью этого поля мы можем проверить, есть ли элементы в стеке или он пуст. Это полезно, учитывая, что метод Pop генерирует исключение.

```
public int Count
{
    get
    {
        return _items.Count;
    }
}
```

ГЛАВА 5. Очередь

Очереди очень похожи на стеки. Они также не дают доступа к произвольному элементу, но, в отличие от стека, элементы добавляются (enqueue) и забираются (dequeue) с разных концов. Такой метод называется «первый вошел, первый вышел» (First-In-First-Out или FIFO). То есть забирать элементы из очереди мы будем в том же порядке, что и клали. Как реальная очередь или конвейер.

Очереди часто используются в программах для реализации буфера, в который можно положить элемент для последующей обработки, сохраняя порядок поступления. Например, если база

данных поддерживает только одно соединение, можно использовать очередь потоков, которые будут, ждать своей очереди на доступ к БД.

Класс Queue

Класс Queue, как и стек, будет реализован с помощью связного списка. Он будет предоставлять метод Enqueue для добавления элемента, Dequeue для удаления, Peek и Count. Как и класс Stack, он не будет реализовывать интерфейс ICollection<T>, поскольку это коллекции специального назначения.

```
1  public class Queue
 2 {
 3     LinkedList _items = new LinkedList();
 4
 5     public void Enqueue(T value)
 6     {
 7         throw new NotImplementedException();
 8     }
 9
10     public T Dequeue()
11     {
12         throw new NotImplementedException();
13     }
14
15     public T Peek()
16     {
17         throw new NotImplementedException();
18     }
19
20     public int Count
21     {
22         get;
23     }
24 }
```

5.1. Метод Enqueue

- Поведение: Добавляет элемент в очередь.

- Сложность: O(1).

Новые элементы очереди можно добавлять как в начало списка, так и в конец. Важно только, чтобы элементы доставались с противоположного края. В данной реализации мы будем добавлять новые элементы в начало внутреннего списка.

```
1 Public void Enqueue(T value)
2 {
3     _items.AddFirst(value);
4 }
```

5.2. Метод Dequeue

- Поведение: Удаляет первый помещенный элемент из очереди и возвращает его. Если очередь пустая, генерирует InvalidOperationException.
- Сложность: O(1).

Поскольку мы вставляем элементы в начало очереди, убирать их будем с конца. Если очередь пуста, генерируем исключение.

```
1  public T Dequeue()
2  {
3      if (_items.Count == 0)
4      {
5          throw new InvalidOperationException("The
6  queue is empty");
7      }
8
9      T last = _items.Tail.Value;
10
11     _items.RemoveLast();
12
13     return last;
   }
```

5.3. Метод Peek

- Поведение: Возвращает элемент, который вернет следующий вызов метода Dequeue. Очередь остается

без изменений. Если очередь пустая, генерирует InvalidOperationException.

- Сложность: O(1).

```
public T Peek()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The queue is empty");
    }

    return _items.Tail.Value;
}
```

5.4. Метод Count

- Поведение: Возвращает количество элементов в очереди или 0, если очередь пустая.
- Сложность: O(1).

```
public int Count
{
    get
    {
        return _items.Count;
    }
}
```

ГЛАВА 6. Двусторонняя очередь

Двусторонняя очередь (Double-ended queue), или дек (Deque), расширяет поведение очереди. В дек можно добавлять или удалять элементы как с начала, так и с конца очереди. Такое поведение полезно во многих задачах, например, планирование выполнения потоков или реализация других структур данных.

Класс Deque

Класс Deque проще всего реализовать с помощью двусвязного списка. Он позволяет просматривать, удалять и

добавлять элементы в начало и в конец списка. Основное отличие двусторонней очереди от обычной — методы Enqueue, Dequeue, и Peek разделены на пары для работы с обоими концами списка.

```
public class Deque
{
    LinkedList _items = new LinkedList();

    public void EnqueueFirst(T value)
    {
        throw new NotImplementedException();
    }

    public void EnqueueLast(T value)
    {
        throw new NotImplementedException();
    }

    public T DequeueFirst()
    {
        throw new NotImplementedException();
    }

    public T DequeueLast()
    {
        throw new NotImplementedException();
    }

    public T PeekFirst()
    {
        throw new NotImplementedException();
    }

    public T PeekLast()
    {
        throw new NotImplementedException();
    }

    public int Count
    {
```

```
37          get;
38      }
39 }
```

6.1. Метод EnqueueFirst

- Поведение: Добавляет элемент в начало очереди. Этот элемент будет взят из очереди следующим при вызове метода DequeueFirst.
- Сложность: O(1).

```
1 public void EnqueueFirst(T value)
2 {
3     _items.AddFirst(value);
4 }
```

6.2. Метод EnqueueLast

- Поведение: Добавляет элемент в конец очереди. Этот элемент будет взят из очереди следующим при вызове метода DequeueLast.
- Сложность: O(1).

```
1 public void EnqueueLast(T value)
2 {
3     _items.AddLast(value);
4 }
```

6.3. Метод DequeueFirst

- Поведение: Удаляет элемент из начала очереди и возвращает его. Если очередь пустая, кидает InvalidOperationException.
- Сложность: O(1).

```
1  public T DequeueFirst()
 2 {
 3     if (_items.Count == 0)
 4     {
 5         throw new
 6 InvalidOperationException("DequeueFirst called when
```

```
deque is empty");
    }

    T temp = _items.Head.Value;

    _items.RemoveFirst();

    return temp;
}
```

6.4. Метод DequeueLast

- Поведение: Удаляет элемент с конца очереди и возвращает его. Если очередь пустая, кидает InvalidOperationException.
- Сложность: O(1).

```
public T DequeueLast()
{
    if (_items.Count == 0)
    {
        throw new
InvalidOperationException("DequeueLast called when
deque is empty");
    }

    T temp = _items.Tail.Value;

    _items.RemoveLast();

    return temp;
}
```

6.5. Метод PeekFirst

- Поведение: Возвращает элемент из начала очереди, не изменяя ее. Если очередь пустая, кидает InvalidOperationException.
- Сложность: O(1).

```
public T PeekFirst()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("PeekFirst
called when deque is empty");
    }

    return _items.Head.Value;
}
```

6.6. Метод PeekLast

- Поведение: Возвращает элемент с конца очереди, не изменяя ее. Если очередь пустая, кидает InvalidOperationException.
- Сложность: O(1).

```
public T PeekLast()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("PeekLast
called when deque is empty");
    }

    return _items.Tail.Value;
}
```

6.7. Метод Count

- Поведение: Возвращает количество элементов в очереди или 0, если очередь пустая.
- Сложность: O(1).

```
public int Count
{
    get
    {
        return _items.Count;
    }
```

```
7 }
```

ГЛАВА 7. Сет (множеств)

Множество — коллекция, которая реализует основные математические операции над множествами: пересечение (intersection), объединение (union), разность (difference) и симметрическую разность (symmetric difference). Каждый из алгоритмов мы разберем в соответствующем разделе.

В математике множества — это коллекции объектов, у которых есть что-то общее. Например, мы можем объявить множество четных положительных целых чисел:

```
1 [2, 4, 6, 8, 10, ...]
```

Или множество нечетных положительных целых:

```
1 [1, 3, 5, 7, 9, ...]
```

В этих двух множествах нет общих элементов. Давайте посмотрим на множество делителей числа 100:

```
1 [1, 2, 4, 5, 10, 20, 25, 50, 100]
```

Теперь мы можем узнать, какие делители числа 100 — нечетные, просто глядя на множество нечетных чисел и на множество делителей и выбирая те из чисел, которые присутствуют в обоих. Мы также можем ответить на вопрос: «Какие нечетные числа не являются делителями ста?» или «Какие положительные целые, четные или нечетные, не являются делителями ста?».

Предположим, что у нас есть множество всех работников предприятия и множество работников, прошедших ежемесячную проверку. Тогда мы с легкостью сможем ответить на вопрос: «Кто из работников не прошел проверку?».

Мы также можем добавить различные множества и построить более сложный запрос, например: «Кто из штатных

работников отдела продаж, имеющих корпоративную кредитную карточку, не прошел обязательный курс повышения квалификации?».

Класс Set

Класс Set реализует интерфейс IEnumerable и принимает аргумент типа, который является наследником IComparable, так как для работы алгоритмов нужна проверка элементов на равенство.

Элементы множества будут храниться в экземпляре стандартного класса .NET List, но на практике для хранения обычно используются древовидные структуры, например, двоичное дерево поиска. Выбор внутреннего представления будет влиять на сложность алгоритмов работы с множеством. К примеру, при использовании списка метод Contains будет выполняться за O(n) времени, в то время как множество, реализованное с помощью дерева, будет работать в среднем за O(log n) времени.

Кроме методов для работы с множеством класс Set также имеет конструктор, который принимает IEnumerable с начальными элементами.

```
1  public class Set : IEnumerable
 2     where T: IComparable
 3 {
 4     private readonly List _items = new List();
 5
 6     public Set()
 7     {
 8     }
 9
10     public Set(IEnumerable items)
11     {
12         AddRange(items);
13     }
14
15     public void Add(T item);
```

```
16
17      public void AddRange(IEnumerable items);
18
19      public bool Remove(T item);
20
21      public bool Contains(T item);
22
23      public int Count
24      {
25          get;
26      }
27
28      public Set Union(Set other);
29
30      public Set Intersection(Set other);
31
32      public Set Difference(Set other);
33
34      public Set SymmetricDifference(Set other);
35
36      public IEnumerator GetEnumerator();
37
38      System.Collections.IEnumerator
39  System.Collections.IEnumerable.GetEnumerator();
    }
```

7.1. Метод Add

- Поведение: Добавляет элементы в множество. Если элемент уже присутствует в множестве, бросается исключение InvalidOperationException.
- Сложность: O(n)

При реализации метода Add необходимо решить, будем ли мы разрешать дублирующиеся элементы. К примеру, если у нас есть множество:

```
1  [1, 2, 3, 4]
```

Если пользователь попытается добавить число 3, в результате получится:

```
[1, 2, 3, 3, 4]
```

В некоторых случаях это допустимо, но в нашей реализации мы не будем поддерживать дубликаты.

Метод Add использует метод Contains, который будет рассмотрен далее.

```
public void Add(T item)
{
    if (Contains(item))
    {
        throw new InvalidOperationException("Item
already exists in Set");
    }

    _items.Add(item);
}
```

7.2. Метод AddRange

- Поведение: Добавляет несколько элементов в множество. Если какой-либо из добавляемых элементов присутствует в множестве, или в добавляемых элементах есть дублирующиеся, выбрасывается исключение InvalidOperationException.
- Сложность: O(m·n), где m — количество вставляемых элементов и n — количество элементов множества.

```
public void AddRange(IEnumerable items)
{
    foreach (T item in items)
    {
        Add(item);
    }
}
```

7.3. Метод Remove

- Поведение: Удаляет указанный элемент из множества и возвращает true. В случае, если элемента нет, возвращает false.
- Сложность: O(n)

```
public bool Remove(T item)
{
    return _items.Remove(item);
}
```

7.4. Метод Contains

- Поведение: Возвращает true, если множество содержит указанный элемент. В противном случае возвращает false.
- Сложность: O(n)

```
public bool Contains(T item)
{
    return _items.Contains(item);
}
```

7.5. Метод Count

- Поведение: Возвращает количество элементов множества или 0, если множество пусто.
- Сложность: O(1)

```
public int Count
{
    get
    {
        return _items.Count;
    }
}
```

7.6. Метод GetEnumerator

- Поведение: Возвращает итератор для перебора элементов множества.
- Сложность: Получение итератора — O(1), обход элементов множества — O(n).

```
public IEnumerator GetEnumerator()
{
    return _items.GetEnumerator();
}

System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
{
    return _items.GetEnumerator();
}
```

7.7. Метод Union

- Поведение: Возвращает множество, полученное операцией объединения его с указанным.
- Сложность: O(m·n), где m и n — количество элементов переданного и текущего множеств соответственно.

Объединение множеств — это множество, которое содержит элементы, присутствующие хотя бы в одном из двух.

Например, на рисунке 1 изображены два множества (выделены красным).:

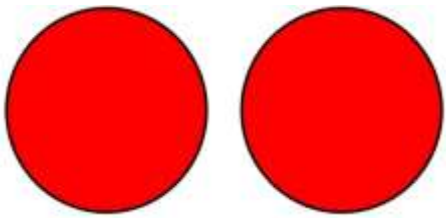


Рис.1 Визуальное представление двух разных множеств

При операции объединения выбираются элементы обоих множеств. Если элемент присутствует в обоих множествах, берется только одна его копия. Объединение двух множеств показано желтым цветом на рисунке 2.

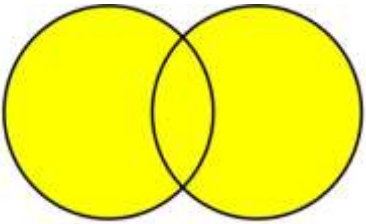


Рис.2 Визуальное представление объединения множеств

Такая диаграмма, наглядно показывающая операции над множествами, называется диаграммой Венна.

Другой пример с использованием множеств целых чисел:

```
1 [1, 2, 3, 4] union [3, 4, 5, 6] = [1, 2, 3, 4, 5, 6]
```

```
1  public Set Union(Set other)
 2 {
 3     Set result = new Set(_items);
 4
 5     foreach (T item in other._items)
 6     {
 7         if (!Contains(item))
 8         {
 9             result.Add(item);
10         }
11     }
12
13     return result;
14 }
```

7.8. Метод Intersection

- Поведение: Возвращает множество, полученное операцией пересечения его с указанным.
- Сложность: O(m·n), где m и n — количество элементов переданного и текущего множеств соответственно.

Пересечение множеств содержит только те элементы, которые есть в обоих множествах. Используя диаграмму Венна, пересечение можно изобразить так, как показано на рисунке 3:

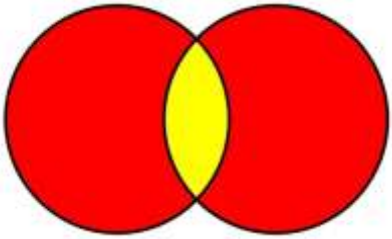


Рис.3 Визуальное представление пересечения множеств

Или, используя целые числа:

```
1 [1, 2, 3, 4] intersect [3, 4, 5, 6] = [3, 4]
```

```
1  public Set Intersection(Set other)
 2 {
 3     Set result = new Set();
 4
 5     foreach (T item in _items)
 6     {
 7         if (other._items.Contains(item))
 8         {
 9             result.Add(item);
10         }
11     }
12
13     return result;
14 }
```

7.9. Метод Difference

- Поведение: Возвращает множество, являющееся разностью текущего с указанным.
- Сложность: O(m·n), где m и n — количество элементов переданного и текущего множеств соответственно.

Разность множеств — это элементы, которые содержатся в одном множестве (в том, у которого вызывается метод), но не содержатся в другом (в том, которое передается в качестве аргумента). Диаграмма Венна для разности множеств будет выглядеть так, как показано на рисунке 4:

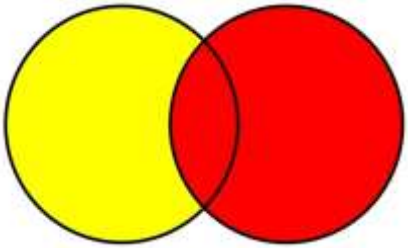


Рис.4 Визуальное представление разности множеств

Или, используя целые числа:

```
1 [1, 2, 3, 4] difference [3, 4, 5, 6] = [1, 2]
```

```
1  public Set Difference(Set other)
 2 {
 3     Set result = new Set(_items);
 4
 5     foreach (T item in other._items)
 6     {
 7         result.Remove(item);
 8     }
 9
10     return result;
11 }
```

7.10. Метод Symmetric Difference

- Поведение: Возвращает множество, являющееся симметрической разностью текущего с указанным.
- Сложность: O(m·n), где m и n — количество элементов переданного и текущего множеств соответственно.

Симметрическая разность — это все элементы, которые содержатся только в одном из рассматриваемых множеств. Диаграмма Венна для симметрической разности будет выглядеть так, как показано на рисунке 5:

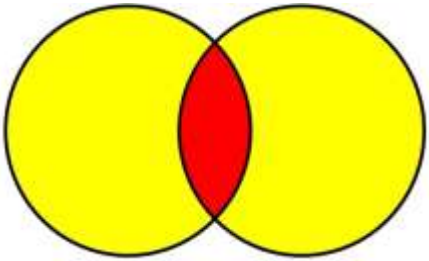


Рис.5 Визуальное представление симметрической разности множеств

Или, используя целые числа:

```
1 [1, 2, 3, 4] symmetric difference [3, 4, 5, 6] = [1,
2, 5, 6]
```

Симметрическая разность — это «пересечение наоборот». Учитывая это, определим её, используя уже имеющиеся операции.

Итак, мы хотим получить множество, которое содержит все элементы из двух множеств, за исключением тех, которые есть в обоих. Другими словами, необходимо получить разность объединения двух множеств и их пересечения.

Или, если рассматривать по шагам:

```
  [1, 2, 3, 4] union [3, 4, 5, 6] = [1, 2, 3, 4, 5, 6]
1
2 [1, 2, 3, 4] intersection [3, 4, 5, 6] = [3, 4]
3
4 [1, 2, 3, 4, 5, 6] set difference [3, 4] = [1, 2, 5,
5 6]
```

Что дает нам нужный результат: [1, 2, 5, 6].

```
1 public Set SymmetricDifference(Set other)
2 {
3     Set union = Union(other);
4     Set intersection = Intersection(other);
5
6     return union.Difference(intersection);
7 }
```

7.11. Метод IsSubset

Так же метод IsSubset, который проверяет, содержится ли одно множество целиком в другом, может быть реализован в рассматриваемом классе. Например:

```
1 [1, 2, 3] is subset [0, 1, 2, 3, 4, 5] = true
```

В то время, как:

```
  [1, 2, 3] is subset [0, 1, 2] = false
1
```

Дело в том, что эту проверку можно провести, используя уже имеющиеся методы. К примеру:

```
  [1, 2, 3] difference [0, 1, 2, 3, 4, 5] = []
1
```

Пустое множество в результате говорит о том, что все элементы первого множества содержатся во втором, а значит, первое является подмножеством второго. Другой способ проверить это:

```
  [1, 2, 3] intersection [0, 1, 2, 3, 4, 5] = [1, 2, 3]
1
```

Если в результате получим множество с таким же количеством элементов, что и изначальное, значит, оно является подмножеством второго.

В общем случае класс Set может иметь метод IsSubset (который может быть реализован более эффективно). Однако стоит помнить, что это уже не новая возможность, а другое применение существующих.

ГЛАВА 8. Деревья

Дерево — это структура, в которой у каждого узла может быть ноль или более подузлов — «детей». Например, дерево может выглядеть так, как это изображено на рисунке 6:

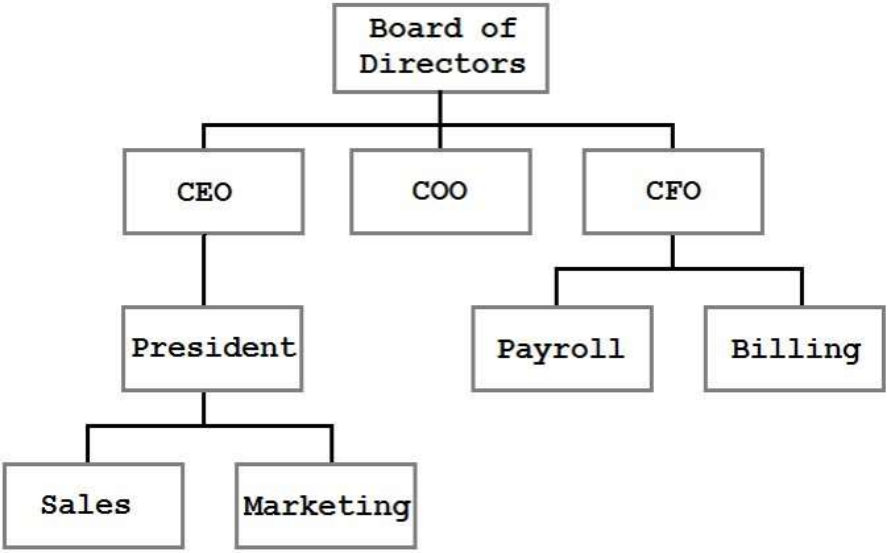


Рис.6 Пример древовидной структуры данных

Это дерево показывает структуру компании. Узлы представляют людей или подразделения, линии — связи и

отношения. Дерево — самый эффективный способ представления и хранения такой информации.

Дерево на картинке выше достаточно тривиально. Оно отражает только отношение родства категорий, но не накладывает никаких ограничений на свою структуру. У генерального директора может быть как один непосредственный подчиненный, так и несколько или ни одного. На рисунке отдел продаж находится левее отдела маркетинга, но порядок на самом деле не имеет значения. Единственное ограничение дерева — каждый узел может иметь не более одного родителя. Самый верхний узел (совет директоров, в нашем случае) родителя не имеет. Этот узел называется «корневым», или «корнем».

8.1. Двоичное дерево поиска

Двоичное дерево поиска похоже на дерево из примера выше, но строится по определенным правилам:

- У каждого узла не более двух детей (child).
- Любое значение меньше значения узла становится левым ребенком (left child) или ребенком левого ребенка.
- Любое значение больше или равное значению узла становится правым ребенком (right child) или ребенком правого ребенка.

Давайте посмотрим на дерево, построенное по этим правилам:

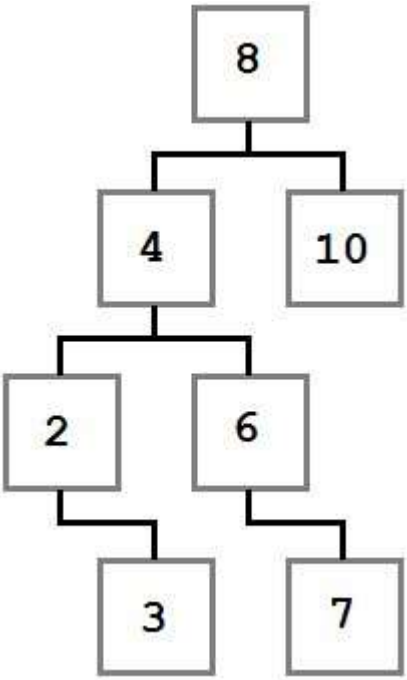


Рис.7 Пример двоичного дерева поиска

Обратите внимание, как указанные ограничения влияют на структуру дерева. Каждое значение слева от корня (8) меньше восьми, каждое значение справа — больше либо равно корню. Это правило применимо к любому узлу дерева.

Учитывая это, представим, как можно построить такое дерево. Поскольку вначале дерево было пустым, первое добавленное значение — восьмерка — стало его корнем.

Мы не знаем точно, в каком порядке добавлялись остальные значения, но можем представить один из возможных путей. Узлы добавляются методом Add, который принимает добавляемое значение.

```
1 BinaryTree tree = new BinaryTree();
```

```
2 tree.Add(8);
3 tree.Add(4);
4 tree.Add(2);
5 tree.Add(3);
6 tree.Add(10);
7 tree.Add(6);
8 tree.Add(7);
```

Рассмотрим подробнее первые шаги. В первую очередь добавляется 8. Это значение становится корне дерева. Затем мы добавляем 4. Поскольку 4 меньше 8, мы добавляем ее в качестве левого ребенка, согласно правилу 2. Поскольку у узла с восьмеркой нет детей слева, 4 становится единственным левым ребенком.

После этого мы добавляем 2. 2 меньше 8, поэтому идем налево. Так как слева уже есть значение, сравниваем его со ним. 2 меньше 4, а у четверки нет детей слева, поэтому 2 становится левым ребенком для 4.

Затем мы добавляем тройку. Она идет левее 8 и 4. Но так как 3 больше, чем 2, она становится правым ребенком для 2, согласно третьему правилу.

Последовательное сравнение вставляемого значения с потенциальным родителем продолжается до тех пор, пока не будет найдено место для вставки, и повторяется для каждого вставляемого значения до тех пор, пока не будет построено все дерево целиком.

Класс BinaryTreeNode

Класс BinaryTreeNode представляет один узел двоичного дерева. Он содержит ссылки на левое и правое поддеревья (если поддерева нет, ссылка имеет значение null), данные узла и метод IComparable.CompareTo для сравнения узлов. Он пригодится для определения, в какое поддерево должен идти данный узел. Как видите, класс BinaryTreeNode достаточно прост:

```
class BinaryTreeNode : IComparable
    where TNode : IComparable
{
    public BinaryTreeNode(TNode value)
    {
        Value = value;
    }

    public BinaryTreeNode Left { get; set; }
    public BinaryTreeNode Right { get; set; }
    public TNode Value { get; private set; }

    public int CompareTo(TNode other)
    {
        return Value.CompareTo(other);
    }
}
```

Класс BinaryTree

Класс BinaryTree предоставляет основные методы для манипуляций с данными: вставка элемента (Add), удаление (Remove), метод Contains для проверки, есть ли такое значение в дереве, несколько методов для обхода дерева различными способами, метод Count и Clear.

Кроме того, в классе есть ссылка на корневой узел дерева и поле с общим количеством узлов.

```
public class BinaryTree : IEnumerable
    where T : IComparable
{
    private  BinaryTreeNode _head;
    private int _count;

    public void Add(T value)
    {
        throw new NotImplementedException();
    }

    public bool Contains(T value)
```

```
13      {
14          throw new NotImplementedException();
15      }
16
17      public bool Remove(T value)
18      {
19          throw new NotImplementedException();
20      }
21
22      public void PreOrderTraversal(Action action)
23      {
24          throw new NotImplementedException();
25      }
26
27      public void PostOrderTraversal(Action action)
28      {
29          throw new NotImplementedException();
30      }
31
32      public void InOrderTraversal(Action action)
33      {
34          throw new NotImplementedException();
35      }
36
37      public IEnumerator GetEnumerator()
38      {
39          throw new NotImplementedException();
40      }
41
42      System.Collections.IEnumerator
43  System.Collections.IEnumerable.GetEnumerator()
44      {
45          throw new NotImplementedException();
46      }
47
48      public void Clear()
49      {
50          throw new NotImplementedException();
51      }
52
53      public int Count
```

```
54      {
55          get;
56      }
    }
```

8.1.1. Метод Add

- Поведение: Добавляет элемент в дерево на корректную позицию.
- Сложность: O(log n) в среднем; O(n) в худшем случае.

Добавление узла не представляет особой сложности. Оно становится еще проще, если решать эту задачу рекурсивно. Есть всего два случая, которые надо учесть:

1. Дерево пустое.
2. Дерево не пустое.

Если дерево пустое, создаем новый узел и добавляем его в дерево. Во втором случае сравниваем переданное значение со значением в узле, начиная от корня. Если добавляемое значение меньше значения рассматриваемого узла, повторяем ту же процедуру для левого поддерева. В противном случае — для правого.

```
1  public void Add(T value)
 2 {
 3      if (_head == null)
 4      {
 5          _head = new BinaryTreeNode(value);
 6      }
 7      else
 8      {
 9          AddTo(_head, value);
10      }
11
12      _count++;
13 }
14
15 private void AddTo(BinaryTreeNode node, T value)
16 {
```

```
17      if (value.CompareTo(node.Value) < 0)
18      {
19          if (node.Left == null)
20          {
21              node.Left = new BinaryTreeNode(value);
22          }
23          else
24          {
25              AddTo(node.Left, value);
26          }
27      }
28      else
29      {
30          if (node.Right == null)
31          {
32              node.Right = new BinaryTreeNode(value);
33          }
34          else
35          {
36              AddTo(node.Right, value);
37          }
38      }
39  }
```

8.1.2. Метод Remove

- Поведение: Удаляет первый узел с заданным значением.
- Сложность: O(log n) в среднем; O(n) в худшем случае.

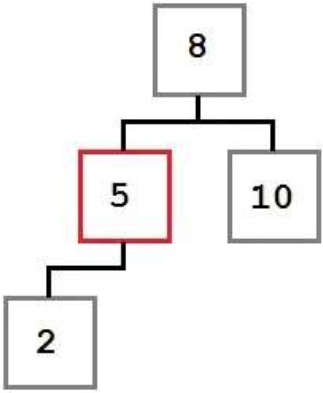
Удаление узла из дерева — одна из тех операций, которые кажутся простыми, но на самом деле таят в себе немало подводных камней.

В целом, алгоритм удаления элемента выглядит так:

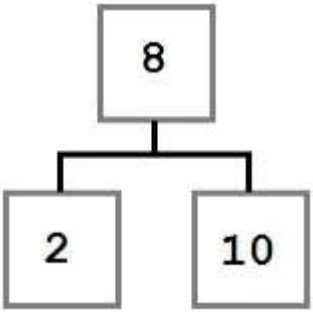
- Найти узел, который надо удалить
- Удалить его

Первый шаг достаточно простой. Рассмотрим поиск узла в методе Contains далее. После того, как узел, который необходимо удалить, найдет, возможны три случая.

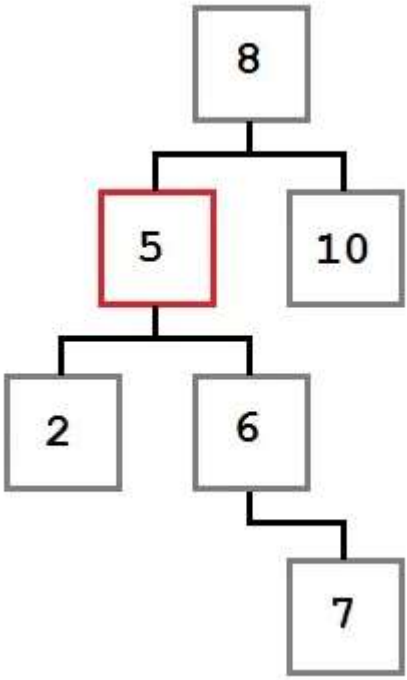
Случай 1: У удаляемого узла нет правого ребенка.



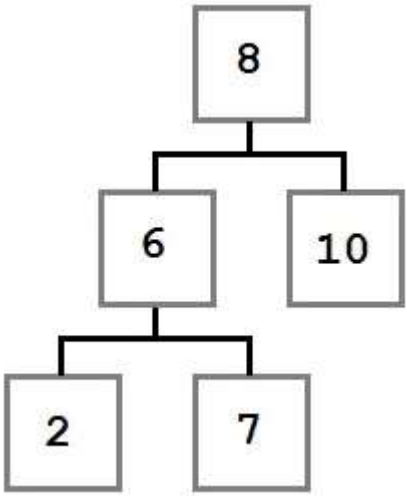
В этом случае мы просто перемещаем левого ребенка (при его наличии) на место удаляемого узла. В результате дерево будет выглядеть так:



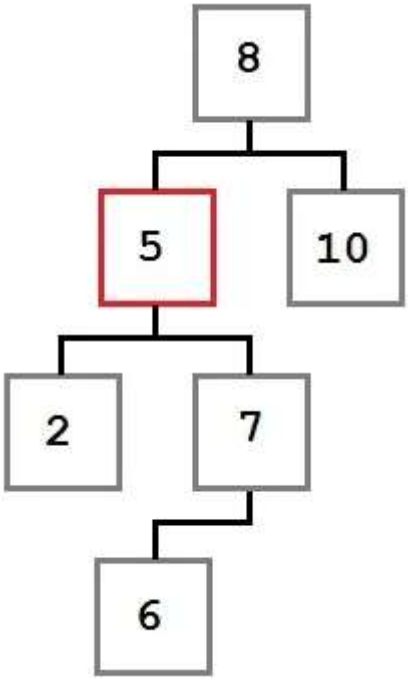
Случай 2: У удаляемого узла есть только правый ребенок, у которого, в свою очередь нет левого ребенка.



В этом случае надо переместить правого ребенка удаляемого узла (6) на его место. После удаления дерево будет выглядеть так:



Случай 3: У удаляемого узла есть первый ребенок, у которого есть левый ребенок.



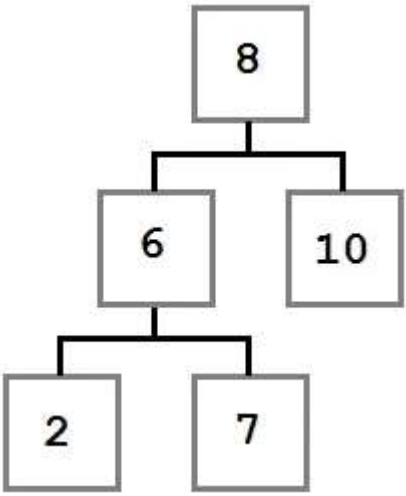
В этом случае место удаляемого узла занимает крайний левый ребенок правого ребенка удаляемого узла.

Рассмотрим, почему это так. Мы знаем о поддереве, начинающимся с удаляемого узла следующее:

1. Все значения справа от него больше или равны значению самого узла
2. Наименьшее значение правого поддерева — крайнее левое

Мы должны поместить на место удаляемого узел со значением, меньшим или равным любому узлу справа от него. Для этого нам необходимо найти наименьшее значение в правом поддереве. Поэтому мы берем крайний левый узел правого поддерева.

После удаления узла дерево будет выглядеть так:



Теперь, когда мы знаем, как удалять узлы, посмотрим на код, который реализует этот алгоритм.

Отметим, что метод FindWithParent (см. метод Contains) возвращает найденный узел и его родителя, поскольку

необходимо заменить левого или правого ребенка родителя удаляемого узла.

Этого можно избежать, если будем хранить ссылку на родителя в каждом узле, но это увеличит расход памяти и сложность всех алгоритмов, несмотря на то, что ссылка на родительский узел используется только в одном.

```
1  public bool Remove(T value)
 2 {
 3     BinaryTreeNode current, parent;
 4
 5     current = FindWithParent(value, out parent);
 6
 7     if (current == null)
 8     {
 9         return false;
10     }
11
12     _count--;
13
14     if (current.Right == null)
15     {
16         if (parent == null)
17         {
18             _head = current.Left;
19         }
20         else
21         {
22             int result =
23 parent.CompareTo(current.Value);
24             if (result > 0)
25             {
26                 parent.Left = current.Left;
27             }
28             else if (result < 0)
29             {
30                 parent.Left = current.Right;
31             }
32             else if (result < 0)
33             {
```

```
34                  parent.Left = leftmost;
35              }
36              else if (result < 0)
37              {
38                  parent.Right = leftmost;
39              }
40          }
41      }
42
43      return true;
    }
```

8.1.3. Метод Contains

- Поведение: Возвращает true если значение содержится в дереве. В противном случает возвращает false.
- Сложность: O(log n) в среднем; O(n) в худшем случае.

Метод Contains выполняется с помощью метода FindWithParent, который проходит по дереву, выполняя в каждом узле следующие шаги:

1. Если текущий узел null, вернуть null.
2. Если значение теккущего узна равно искомому, вернуть текущий узел.
3. Если искомое значение меньше значения текущего узла, установить левого ребенка в качестве текущего узла и перейти к шагу 1.
4. В противном случае, установить правого ребенка в качестве текущего узла и перейти к шагу 1.

Поскольку Contains возвращает булево значение, оно определяется сравнением результата выполнения FindWithParent с null. Если FindWithParent вернул непустой узел, Contains возвращает true.

Метод FindWithParent также используется в методе Remove.

```
1 public bool Contains(T value)
```

```
{
    BinaryTreeNode parent;
    return FindWithParent(value, out parent) != null;
}

private BinaryTreeNode FindWithParent(T value, out
BinaryTreeNode parent)
{
    BinaryTreeNode current = _head;
    parent = null;

    while (current != null)
    {
        int result = current.CompareTo(value);

        if (result > 0)
        {
            parent = current;
            current = current.Left;
        }
        else if (result < 0)
        {
            parent = current;
            current = current.Right;
        }
        else
        {
            break;
        }
    }

    return current;
}
```

8.1.4. Метод Count

- Поведение: Возвращает количество узлов дерева или 0, если дерево пустое.
- Сложность: O(1)

Это поле инкрементируется методом Add и декрементируется методом Remove.

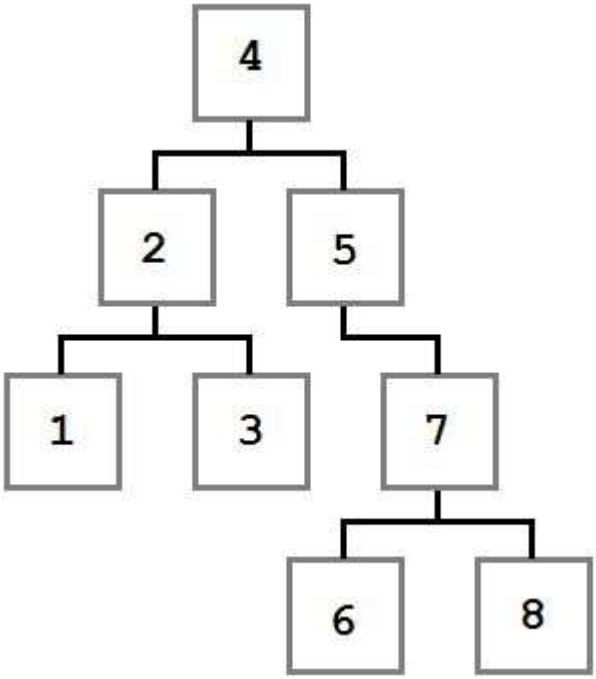
```
public int Count
{
    get
    {
        return _count;
    }
}
```

8.1.5. Метод Clear

- Поведение: Удаляет все узлы дерева.
- Сложность: O(1)

```
public void Clear()
{
    _head = null;
    _count = 0;
}
```

Обходы дерева — это семейство алгоритмов, которые позволяют обработать каждый узел в определенном порядке. Для всех алгоритмов обхода ниже в качестве примера будет использоваться следующее дерево:



Методы обхода в примерах будут принимать параметр Action<T>, который определяет действие, поизводимое над каждым узлом.

Также, кроме описания поведения и алгоритмической сложности метода будет указываться порядок значений, полученный при обходе.

8.1.6. Метод Preorder (префиксный обход)

- Поведение: Обходит дерево в префиксном порядке, выполняя указанное действие над каждым узлом.
- Сложность: O(n)
- Порядок обхода: 4, 2, 1, 3, 5, 7, 6, 8

При префиксном обходе алгоритм получает значение текущего узла перед тем, как перейти сначала в левое

поддерево, а затем в правое. Начиная от корня, сначала получим значение 4. Затем таким же образом обходятся левый ребенок и его дети, затем правый ребенок и все его дети.

Префиксный обход обычно применяется для копирования дерева с сохранением его структуры.

```
public void PreOrderTraversal(Action action)
{
    PreOrderTraversal(action, _head);
}

private void PreOrderTraversal(Action action,
BinaryTreeNode node)
{
    if (node != null)
    {
        action(node.Value);
        PreOrderTraversal(action, node.Left);
        PreOrderTraversal(action, node.Right);
    }
}
```

8.1.7. Метод Postorder (постфиксный обход)

- Поведение: Обходит дерево в постфиксном порядке, выполняя указанное действие над каждым узлом.
- Сложность: O(n)
- Порядок обхода: 1, 3, 2, 6, 8, 7, 5, 4

При постфиксном обходе мы посещаем левое поддерево, правое поддерево, а потом, после обхода всех детей, переходим к самому узлу.

Постфиксный обход часто используется для полного удаления дерева, так как в некоторых языках программирования необходимо убирать из памяти все узлы явно, или для удаления поддерева. Поскольку корень в данном случае обрабатывается последним, мы, таким образом, уменьшаем работу, необходимую для удаления узлов.

```
public void PostOrderTraversal(Action action)
{
    PostOrderTraversal(action, _head);
}

private void PostOrderTraversal(Action action,
BinaryTreeNode node)
{
    if (node != null)
    {
        PostOrderTraversal(action, node.Left);
        PostOrderTraversal(action, node.Right);
        action(node.Value);
    }
}
```

8.1.8. Метод Inorder (инфиксный обход)

- Поведение: Обходит дерево в инфиксном порядке, выполняя указанное действие над каждым узлом.
- Сложность: O(n)
- Порядок обхода: 1, 2, 3, 4, 5, 6, 7, 8

Инфиксный обход используется тогда, когда необходимо обойти дерево в порядке, соответствующем значениям узлов. В примере выше в дереве находятся числовые значения, поэтому мы обходим их от самого маленького до самого большого. То есть от левых поддеревьев к правым через корень.

В примере ниже показаны два способа инфиксного обхода. Первый — рекурсивный. Он выполняет указанное действие с каждым узлом. Второй использует стек и возвращает итератор для непосредственного перебора.

```
Public void InOrderTraversal(Action action)
{
    InOrderTraversal(action, _head);
}

private void InOrderTraversal(Action action,
```

```
 7 BinaryTreeNode node)
 8 {
 9     if (node != null)
10     {
11         InOrderTraversal(action, node.Left);
12 
13         action(node.Value);
14 
15         InOrderTraversal(action, node.Right);
16     }
17 }
18 
19 public IEnumerator InOrderTraversal()
20 {
21     if (_head != null)
22     {
23         Stack stack = new Stack();
24 
25         BinaryTreeNode current = _head;
26         bool goLeftNext = true;
27         stack.Push(current);
28 
29         while (stack.Count &gt; 0)
30         {
31             if (goLeftNext)
32             {
33                 while (current.Left != null)
34                 {
35                     stack.Push(current);
36                     current = current.Left;
37                 }
38             }
39 .
40             yield return current.Value;
41 
42             if (current.Right != null)
43             {
44                 current = current.Right;
45                 goLeftNext = true;
46             }
47             else
```

```
48              {
49                  current = stack.Pop();
50                  goLeftNext = false;
51              }
52          }
53      }
    }
```

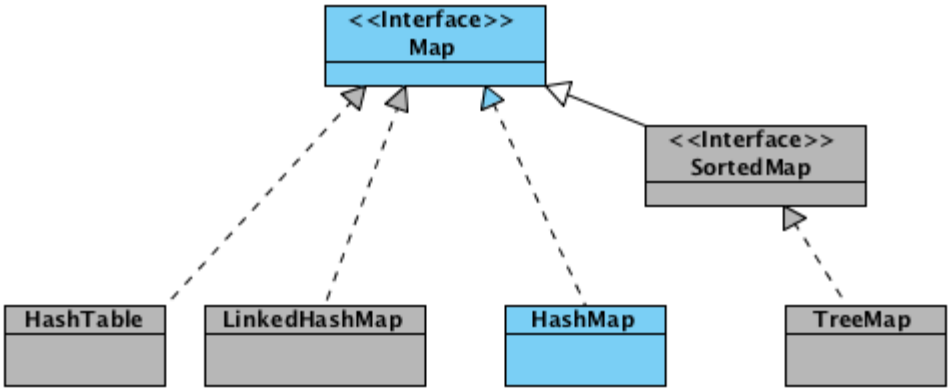
8.1.9. Метод GetEnumerator

- Поведение: Возвращает итератор для обхода дерева инфиксным способом.
- Сложность: Получение итератора — O(1). Обход дерева — O(n).

```
1 public IEnumerator GetEnumerator()
2 {
3     return InOrderTraversal();
4 }
5
6 System.Collections.IEnumerator
7 System.Collections.IEnumerable.GetEnumerator()
8 {
9     return GetEnumerator();
  }
```

ГЛАВА 9. Мэп

HashMap — основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени. Разрешение коллизий осуществляется с помощью метода цепочек.



9.1. Создание объекта

```
1 Map<String, String> hashmap = new HashMap<String,
String>();
```

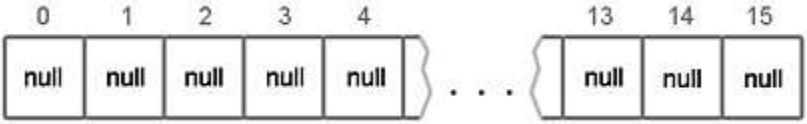
```
1 Footprint{Objects=2, References=20, Primitives=[int x
3, float]}
2 Object size: 120 bytes
```

Объект hashmap, содержит ряд свойств:

- *table* — Массив типа Entry[], который является хранилищем ссылок на списки (цепочки) значений;
- *loadFactor* — Коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;
- *threshold* — Предельное количество элементов, при достижении которого, размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле (capacity * loadFactor);
- *size* — Количество элементов HashMap-а;

В конструкторе, выполняется проверка валидности переданных параметров и установка значений в соответствующие свойства класса. Словом, ничего необычного.

```
1
2 table = new Entry[capacity];
3
```



Возможно указать емкость и коэффициент загрузки, используя конструкторы HashMap(capacity) и HashMap(capacity, loadFactor). Максимальная емкость, которую возможно установить, равна половине максимального значения int (1073741824).

Добавление элементов:

```
1 hashmap.put("0", "zero");
```

```
1 Footprint{Objects=7, References=25, Primitives=[int x
10, char x 5, float]}
2 Object size: 232 bytes
```

При добавлении элемента, последовательность шагов следующая:

1. Сначала ключ проверяется на равенство null. Если эта проверка вернула true, будет вызван метод putForNullKey(value).
2. Далее генерируется хэш на основе ключа. Для генерации используется метод hash(hashCode), в который передается key.hashCode().

```
1 static int hash(int h)
2 {
3     h ^= (h >>> 20) ^ (h >>> 12);
4     return h ^ (h >>> 7) ^ (h >>> 4);
5 }
```

Метод hash(key) гарантирует что полученные хэш-коды, будут иметь только ограниченное количество коллизий (примерно 8, при дефолтном значении коэффициента загрузки).

В данном случае, для ключа со значением "0" метод hashCode() вернул значение 48, в итоге:

```
h ^ (h >>> 20) ^ (h >>> 12) = 48

h ^ (h >>>  7) ^ (h >>>  4) = 51
```

С помощью метода indexFor(hash, tableLength) определяется позиция в массиве, на которую будет помещен элемент.

```
static int indexFor(int h, int length)
{
    return h & (length - 1);
}
```

При значении хэша 51 и размере таблице 16, получаем индекс в массиве:

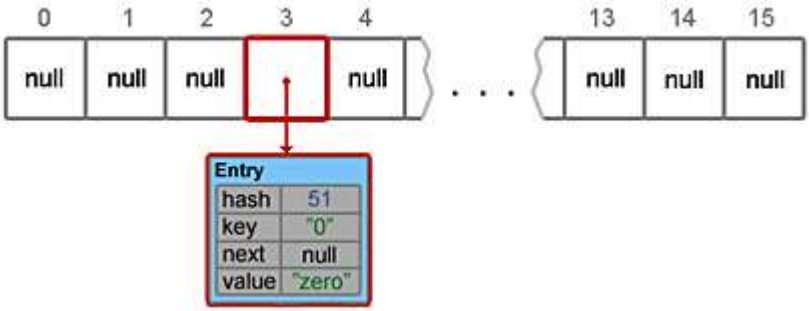
```
h & (length - 1) = 3
```

Теперь, зная индекс в массиве, получаем список (цепочку) элементов, привязанных к этой ячейке. Хэш и ключ нового элемента поочередно сравниваются с хэшами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается.

```
if (e.hash == hash && (e.key == key ||
key.equals(e.key)))
{
    V oldValue = e.value;
    e.value = value;

    return oldValue;
}
```

Если же предыдущий шаг не выявил совпадений, будет вызван метод addEntry(hash, key, value, index) для добавления нового элемента.

```
1 void addEntry(int hash, K key, V value, int index)
2 {
3     Entry<K, V> e = table[index];
4     table[index] = new Entry<K, V>(hash, key, value,
5 e);
6 }
```



Как было сказано выше, если при добавлении элемента в качестве ключа был передан null, действия будут отличаться. Будет вызван метод putForNullKey(value), внутри которого нет вызова методов hash() и indexFor() (все элементы с null-ключами всегда помещаются в table[0]), но есть следующие действия:

1. Все элементы цепочки, привязанные к table[0], поочередно просматриваются в поисках элемента с ключом null. Если такой элемент в цепочке существует, его значение перезаписывается.
2. Если элемент с ключом null не был найден, будет вызван уже знакомый метод addEntry().

Рассмотрим случай, когда при добавлении элемента возникает коллизия.

1. Шаг пропускается – ключ не равен null

2. "idx".hashCode() = 104125

```
h ^ (h >>> 20) ^ (h >>> 12) = 104100
```

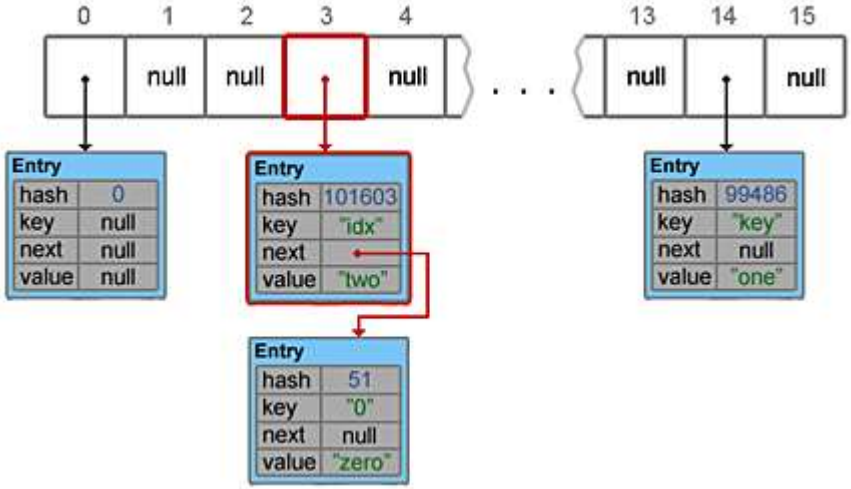
```
h ^ (h >>>  7) ^ (h >>>  4) = 101603
```

3. Определение позиции в массиве

```
h & (length - 1) = 3
```

4. Подобные элементы не найдены
5. Добавление элемента

```
Entry<K, V> e = table[index];
table[index] = new Entry<K, V>(hash, key, value, e);
```



9.2. Resize и Transfer

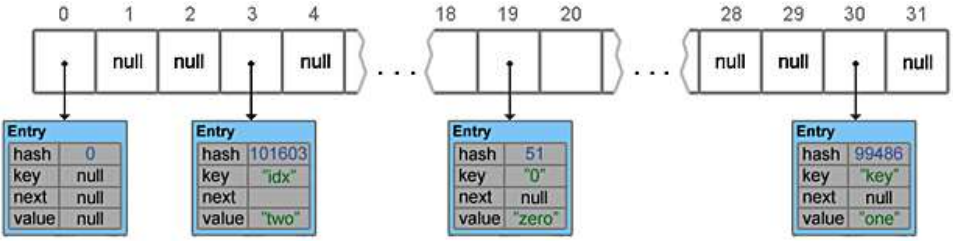
Когда массив table[] заполняется до предельного значения, его размер увеличивается вдвое и происходит перераспределение элементов.

```
void resize(int newCapacity)
{
    if (table.length == MAXIMUM_CAPACITY)
    {
```

```
 5          threshold = Integer.MAX_VALUE;
 6          return;
 7      }
 8
 9      Entry[] newTable = new Entry[newCapacity];
10      transfer(newTable);
11      table = newTable;
12      threshold = (int)(newCapacity * loadFactor);
13  }
```

Метод transfer() перебирает все элементы текущего хранилища, пересчитывает их индексы (с учетом нового размера) и перераспределяет элементы по новому массиву.

Если в исходный hashmap добавить, скажем, еще 15 элементов, то в результате размер будет увеличен и распределение элементов изменится.



9.3. Удаление элементов

У HashMap есть такая же «проблема» как и у ArrayList — при удалении элементов размер массива table[] не уменьшается. И если в ArrayList предусмотрен метод trimToSize(), то в HashMap таких методов нет.

Для примера возьмем исходный объект, занимающий 496 байт. Добавим 150 элементов.

```
1 Footprint{Objects=768, References=1028,
2 Primitives=[int x 1075, char x 2201, float]}
  Object size: 21064 bytes
```

Теперь удалим те же 150 элементов, и снова замерим.

```
1 Footprint{Objects=18, References=278, Primitives=[int
```

```
2 x 25, char x 17, float]}
  Object size: 1456 bytes
```

Как видно, размер не вернулся к исходному. Если есть потребность исправить ситуацию, можно воспользоваться конструктором HashMap(Map).

```
1 hashmap = new HashMap<String, String>(hashmap);
```

```
1 Footprint{Objects=18, References=38, Primitives=[int x
2 25, char x 17, float]}
  Object size: 496 bytes
```

9.4. Итераторы

HashMap имеет встроенные итераторы, с помощью которых можно получить список всех ключей keySet(), всех значений values() или все пары ключ/значение entrySet(). Ниже представлены некоторые варианты для перебора элементов:

```
1

2 for (Map.Entry<String, String> entry:
  hashmap.entrySet())
3     System.out.println(entry.getKey() + " = " +
  entry.getValue());
4
  for (String key: hashmap.keySet())
5     System.out.println(hashmap.get(key));

6 Iterator<Map.Entry<String, String>> itr =
  hashmap.entrySet().iterator();
7 while (itr.hasNext())
      System.out.println(itr.next());
8

9
```

ОСНОВНАЯ ЛИТЕРАТУРА

1. Тюкачев, Н.А. C#. Алгоритмы и структуры данных. [Электронный ресурс] / Н.А. Тюкачев, В.Г. Хлебостроев. — Электрон. дан. — СПб.: Лань, 2017. — 232 с. — Режим доступа: http://e.lanbook.com/book/94748 — Загл. с экрана.
2. Вирт, Н. Алгоритмы и структуры данных. Новая версия для Оберона. [Электронный ресурс]: учеб. пособие — Электрон. дан. — М.: ДМК Пресс, 2010. — 272 с. — Режим доступа: http://e.lanbook.com/book/1261 — Загл. с экрана.
3. Круз, Р.Л. Структуры данных и проектирование программ. [Электронный ресурс]: учеб. пособие — Электрон. дан. — М.: Издательство "Лаборатория знаний", 2014. — 765 с. — Режим доступа: http://e.lanbook.com/book/66126 — Загл. с экрана.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

4. Самуйлов С.В. Алгоритмы и структуры обработки данных [Электронный ресурс]: учебное пособие/ С.В. Самуйлов— Электрон. текстовые данные. — Саратов: Вузовское образование, 2016.— 132 с.— Режим доступа: http://www.iprbookshop.ru/47275.html.— ЭБС «IPRbooks»
5. Мейер, Б. Инструменты, алгоритмы и структуры данных / Б. Мейер. - 2-е изд., испр. - М.: Национальный Открытый Университет «ИНТУИТ», 2016. - 543 с.: схем., ил. - Библиогр. в кн.; То же [Электронный ресурс]. - URL: http://biblioclub.ru/ index.php?page= book&id=429033 (21.09.2017).