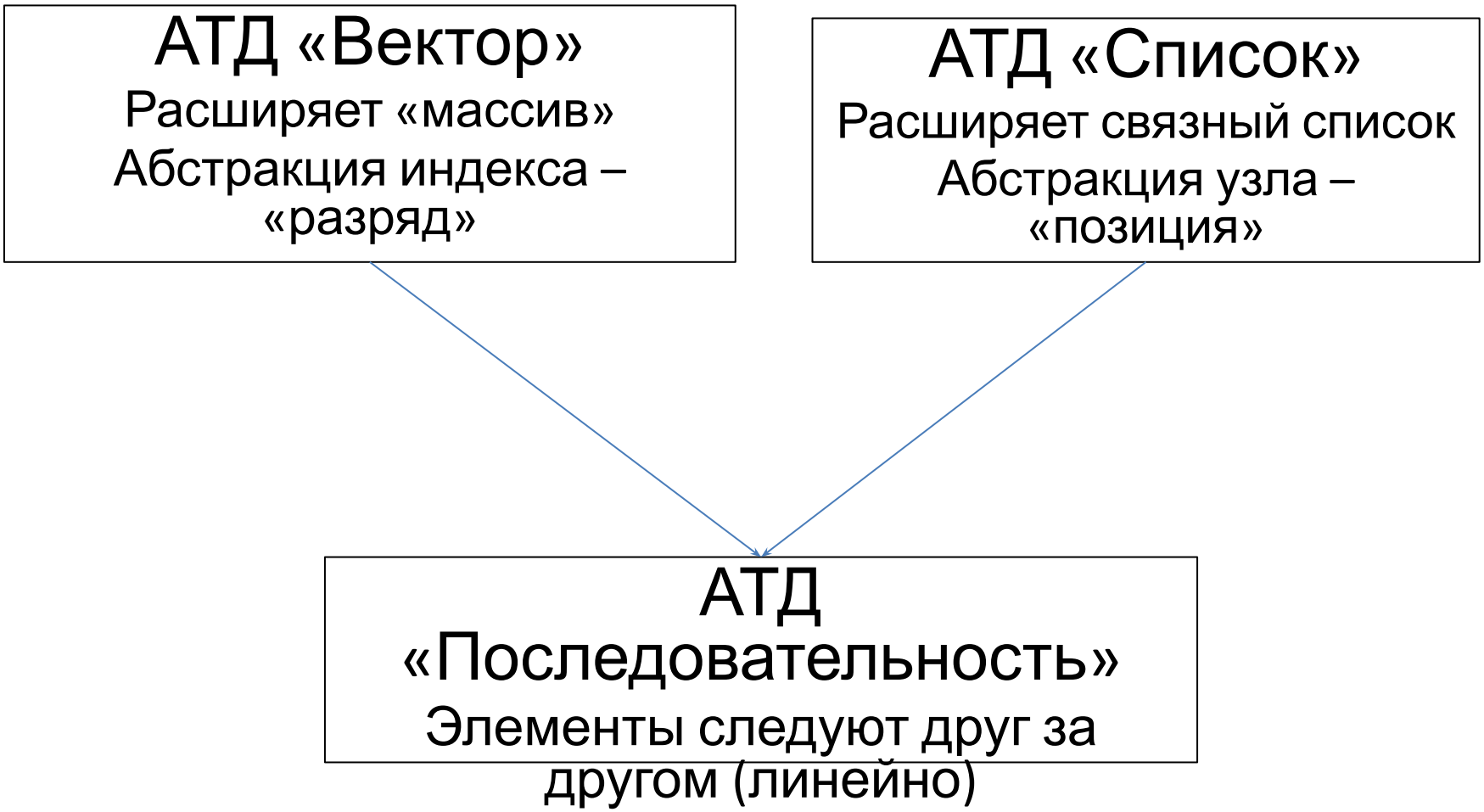
Векторы, списки, последовательности



АТД «Вектор» 1

- Пусть ***S*** — линейная последовательность из ***n*** элементов. ***Разряд*** элемента ***e*** последовательности ***S*** равен количеству элементов, находящихся в ***S*** перед ***e,*** то есть разряд первого элемента последовательности равен 0, а последнего — ***n***-1. Очевидно, что разряд каждого элемента в ***S*** уникален.
- Абстракция «Вектор» заключается в том, что последовательность ***S*** не обязана быть массивом.
- Кроме того, «Вектор» – более динамическая структура, поскольку разряд элемента в ***S*** может меняться вследствие удаления и добавления новых элементов.

АТД «Вектор» 2

Основные методы:

- **ElemAtRank**(***r***): возвращает элемент ***S*** с разрядом ***r***; если ***r*<**0 или ***r*>*п*-**1, где ***п*** — текущее число элементов, выдается сообщение об ошибке.
  Input: целое число; ***Output:*** объект.
- **ReplaceAtRank**(***r,e***): замещает объектом ***e*** элемент с разрядом ***r*** и возвращает замещаемый объект. Если ***r*<**0 или ***r>п*-**1, где ***п —*** текущее число элементов, выдается сообщение об ошибке.
  Input: целое число ***r и*** объект ***e; Output:*** объект.
- **InsertAtRank**(***r,e***): добавляет в ***S*** новый элемент ***e***; если ***r*<**0 или ***r*>*п,*** где ***п*** — текущее число элементов, выдается сообщение об ошибке.
  Input: целое число ***r*** и объект ***e; Output:*** нет.
- **RemoveAtRank**(***r***): удаляет из ***S*** элемент с разрядом ***r;*** если ***r*<**0 или ***r>п*-**1, где ***п —*** текущее число элементов, выдается сообщение об ошибке.
  Input: целое число; ***Output:*** объект.
- стандартные методы ***Size***() и ***IsEmpty***()

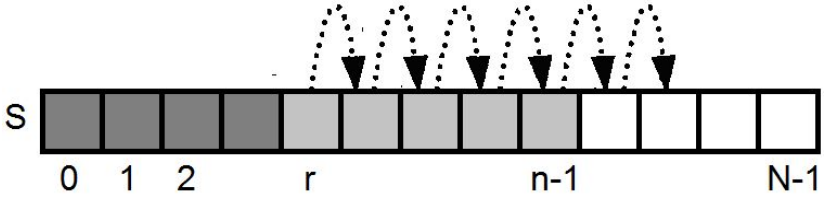
Адаптация вектора для реализации дека

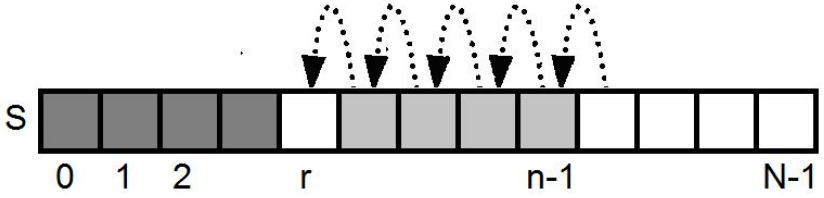
Операция дека	Реализация с помощью вектора
Size()	Size()
IsEmpty()	IsEmpty()
First() Last() InsertFirst(e) InsertLast(e) RemoveFirst() RemoveLast()	ElemAtRank(0) ElemAtRank(Size()-1) InsertAtRank(0,e) InsertAtRank(size(), e) RemoveAtRank(0) RemoveAtRank(Size()-1)

Реализация вектора с помощью массива

```
Алгоритм
InsertAtRank(r,e)
   for i = n-1, n-2,
..., r do
      A[i+1]←A[i]
   A[r]←e
   n←n+1
```

```
Алгоритм
RemoveAtRank(r)
   e←A[r]
   for i = r,
r+1,...,n-2 do
      A[i]←A[i+1]
   n←n-1
   return e
```

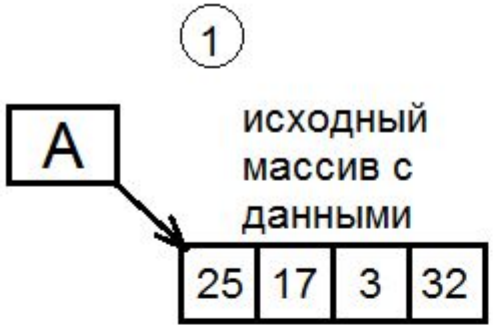


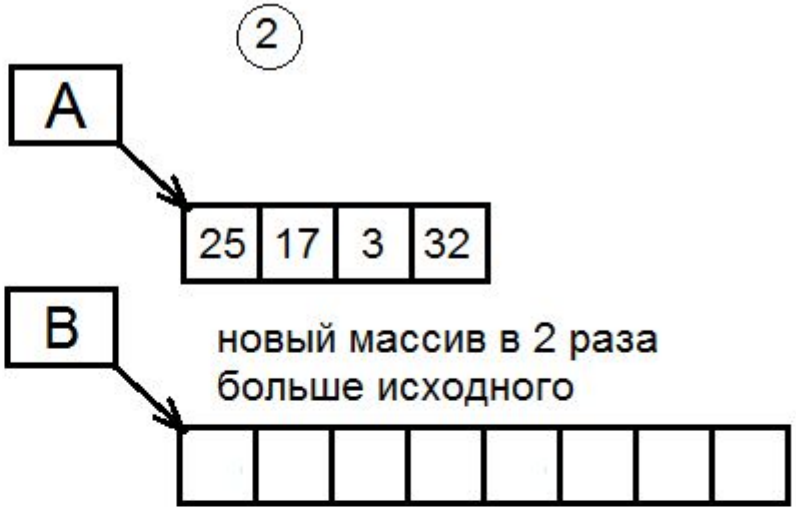


Недостатки:

1. InsertAtRank и RemoveAtRank выполняются за O(N) времени
2. Емкость вектора ограничена фиксированным размером массива

Реализация вектора на основе расширяемого массива (Имитация изменения размера массива)





Реализация вектора на основе расширяемого массива

```
public class ArrayVector : Vector
{   private Object[ ] a;
   private int capacity =16;  /* емкость вектора*/    private int size = 0;      /* текущий размер*/
   public ArrayVector() { a = new Object[capacity]; }   //время O(1)
   public Object ElemAtRank(int r) { return a[r]; }       //время O(1)
   public int Size() { return size; }
   public bool IsEmpty() { return (size == 0); }
   public Object ReplaceAtRank(int r, Object e) { Object temp = a[r]; return temp;   } //время O(1)
   public Object RemoveAtRank(int r)    // время O(N)
   {   Object temp = a[r];
      for (int i=r; i<size-1; i++) a[i] = a[i+1];
      size--;  return temp;
   }
   public void InsertAtRank(int r, Object e)   // время O(N)
   {   if (size == capacity) // переполнение
      {   capacity *= 2;   Object[ ] b = new Object[capacity];
         for (int i=0; i<size; i++) b[i] = a[i];
         a = b;
      }
      for (int i=size-1; i>=r; i--) a[i+1] = a[i];
      a[r] = e; size++;
   }
}
```

АТД «Список» 1

- В списке узлы «знают» друг о друге. Поэтому операции с параметрами-узлами быстрее, чем операции с индексами в массиве.
- Например: RemoveAtNode(***v***), InsertAfterNode(***v***,***e***)
- Абстракция узла – АТД «**Позиция**»
  - GetElement(): возвращает элемент, хранящийся в данной позиции.
    Input: нет; ***Output:*** объект.
  - SetElement(Object e): помещает элемент в позицию.
    Input: элемент; ***Output:*** нет

АТД «Список» - операции доступа по чтению

- **First**(): возвращает позицию первого элемента списка ***S;*** если список пуст, выдается сообщение об ошибке.
  Input: нет; ***Output:*** позиция.
- **Last**(): возвращает позицию последнего элемента списка ***S;*** если список пуст, выдается сообщение об ошибке.
  Input: нет; ***Output:*** позиция.
- **IsFirst**(***p***): возвращает логическое значение, показывающее, является ли данная позиция первой в списке. ***Input:*** позиция p; ***Output:*** логическое значение.
- **IsLast**(***p***): возвращает логическое значение, показывающее, является ли данная позиция последней в списке.
  Input: позиция ***p; Output:*** логическое значение.
- **Before**(***p***): возвращает позицию элемента ***S,*** который предшествует элементу позиции ***p;*** если ***p*** является первой позицией, выдается сообщение об ошибке.
  Input: позиция; ***Output:*** позиция.
- **After**(***p***)***:*** возвращает позицию элемента ***S,*** который следует за элементом позиции ***p;*** если ***p*** является последней позицией, выдается сообщение об ошибке. ***Input:*** позиция; ***Output:*** позиция.

АТД «Список» - модифицирующие операции

- **ReplaceElement**(***p***,***e***): замещает элемент в позиции ***p*** на ***e*** и возвращает элемент, который до этого был в позиции ***p.***
  Input: позиция ***p*** и объект ***e; Output:*** объект.
- **SwapElements**(***p***,***q***): меняет местами элементы в позициях ***p*** и ***q*** таким образом, что элемент в позиции ***p*** перемещается в позицию ***q,*** а элемент, бывший в позиции ***q,*** перемещается в позицию ***p.***
  Input: две позиции; ***Output:*** нет.
- **InsertFirst**(***e***): вставляет новый элемент ***e*** в ***S*** в качестве первого элемента списка.
  Input: объект ***e; Output:*** позиция вставленного элемента ***e.***
- **InsertLast**(***e***): вставляет новый элемент ***e*** в ***S*** в качестве последнего элемента списка.
  Input: объект ***e; Output:*** позиция вставленного элемента ***e.***
- **InsertBefore**(***p***, ***e):*** вставляет новый элемент ***e*** в ***S*** перед позицией ***p;*** если ***p*** является первой позицией, выдается сообщение об ошибке.
  Input: позиция ***p*** и объект ***e; Output:*** позиция вставленного элемента ***e.***
- **InsertAfter**(***p***, ***e):*** вставляет новый элемент ***e*** в ***S*** после позиции ***p;*** если ***p*** является последней позицией, выдается сообщение об ошибке.
  Input: позиция ***p*** и объект ***e; Output:*** позиция вставленного элемента ***e.***
- **Remove**(***p***): удаляет из ***S*** элемент в позиции ***p***
  Input: позиция; ***Output:*** удаленный элемент.

Пример использования списка

Операция	Output	S
InsertFirst(8)	p_1(8)	(8)
InsertAfter(*p_1, 5*)	p_2(5)	(8,5)
InsertBefore (p_2, 3)	p_3(3)	(8,3,5)
InsertFirst(9)	P_4(9)	(9,8,3,5)
Before(p_3)	p_1(8)	(9,8,3,5)
Last()	p_2(5)	(9,8,3,5)
Remove(p_4)	*9*	(8,3,5)
SwapElements(p_1, p_2)	-	(5,3,8)
ReplaceElement(p_3, 7)	*3*	(5,7,8)
InsertAfter(first(),2)	p_2(2)	(5,2,7,8)

Реализация АТД «Список» с помощью двусвязного списка – класс DNode

```
class DNode : Position
{   private DNode prev, next;
    private Object element;   // элемент, хранящийся в данной позиции
    public DNode(DNode newPrev, DNode newNext, Object elem)
    { prev = newPrev; next = newNext; element = elem; }
    public Object GetElement()
    {   if ((prev == null) && (next == null))
            throw new InvalidPositionException("Positionisnotinalist!");
        return element;
    }
    public void SetElement(Object newElement) {element = newElement;}
    public DNode GetNext() { return next; }
    public DNode GetPrev() { return prev; }
    public void SetNext(DNode newNext) { next = newNext; }
    public void SetPrev(DNode newPrev) { prev = newPrev; }
}
```

Реализация АТД «Список» с помощью двусвязного списка – операция InsertAfter

Алгоритм InsertAfter(*p*, *e*):

```
    Создать новый узел v
    v.SetElement(e)
  // связать v с предшествующим узлом
    v.SetPrev(p)
  // связать v с последующим узлом
    v.SetNext(p.getNext())
  // связывает ранее следовавший за p узел с v
    (p.GetNext()).SetPrev(v)
  // связывает p с новым последующим узпом v
    p.SetNext(v)
    return v   { позиция элемента e }
```

Реализация АТД «Список» с помощью двусвязного списка – вспомогательный метод **checkPosition**

```
protected DNode checkPosition(Position p)
 { if (p == null)
     throw new InvalidPositionException("Null Position passed to NodeList.");
   if (p == header)
     throw new InvalidPositionException("Header is not a valid position");
   if (p == trailer)
     throw new InvalidPositionException("Trailer is not a valid position");
   try
   { DNode temp = (DNode)p;
     if ((temp.GetPrev() == null) || (temp.GetNext() == null))
       throw new InvalidPositionException("Position does not belong to a valid
      NodeList");
     return temp;
   }
   catch (Exception e)
   { throw new InvalidPositionException("Position is of wrong type for this
      container.");
   }
 }
```

АТД «Последовательность»

- все методы векторов
- все методы списков
- два дополнительных «связующих» метода, которые обеспечивают переход между разрядами и позициями:
  - AtRank(r): возвращает позицию элемента с разрядом r.
    Input: целое число; ***Output***: позиция.
  - RankOf(***p***): возвращает разряд элемента в позиции ***p.***
    Input: позиция; ***Output:*** целое число.

АТД «Последовательность» – множественное наследование

```
public interface Sequence : List, Vector
{   // Дополнительные "переходные" методы.
    Position AtRank(int rank);
    int RankOf(Position position);
}
```

Реализация последовательности с помощью двусвязного списка

- все методы АТД «список» выполняются за $O(1)$ время.
- Методы же АТД «вектор» реализованы менее эффективно.
- ElemAtRank(***r***) - поиск можно начать с ближайшего конца последовательности, время выполнения составит $O(min(r+1, n-r))$
- Аналогично - InsertAtRank(***r***, ***e***) и RemoveAtRank(***r***)

Реализация последовательности с помощью двусвязного списка 1

```
public class NodeSequence : NodeList, Sequence
{  // проверяем, находится ли разряд в интервале [0,numElt-1];
   protected void checkRank(int rank)  // время O(1).
   {   if (rank<0 || rank>=numElts)
       {  String s = String.Format("Rank {0} is invalid for this sequence of {1} elements.",
          rank, numElts);
          throw new BoundaryViolationException(s);
   }
   public Position ElemAtRank (int rank) // время O(1)
   {   DNode node;
       checkRank(rank);
       if (rank <= Size()/2) // просматриваем последовательность от начала
       {  node = header.GetNext(); for (int i=0; i < rank; i++) node = node.GetNext();
       }
       else // просматриваем последовательность с конца
       {  node = trailer.GetPrev();
          for(int i=1; i<Size()-rank; i++) node = node.GetPrev();
       }
       return node;
   }
```

Реализация последовательности с помощью двусвязного списка 2

```
public void InsertAtRank (int rank, Object element) // время O(n)
   {   if (rank == Size()) // в данном случае не выполняется checkRank
          InsertLast(element);
       else
       {   checkRank(rank);
          InsertBefore(AtRank(rank), element);
       }
   }
   public Object RemoveAtRank (int rank) // время O(n)
   {   checkRank(rank);
       return Remove(AtRank(rank));
   }
   public Object ReplaceAtRank (int rank,Object element) //время O(n)
   {   checkRank(rank);
       return ReplaceElement(AtRank(rank),element);
   }
}
```

Сравнительный анализ различных реализаций последовательности

Операции	Массив	Список
size, isEmpty	O(1)	O(1)
atRank, rankOf, elemAtRank	O(1)	O(n)
first, last, before, after	O(1)	O(1)
replaceElement, swapElements	O(1)	O(1)
replaceAtRank	O(1)	O(n)
insertAtRank, removeAtRank	O(n)	O(n)
insertFirst, insertLast	O(1)	O(1)
insertAfter, insertBefore	O(n)	O(1)
remove	O(n)	O(1)

Каждая из реализаций имеет свои преимущества и недостатки. Выбор того или иного способа обусловлен конкретными требованиями приложения. Поскольку структура АТД «последовательность» не зависит от конкретных условий реализации, применяется наиболее соответствующая реализация с минимальными изменениями в программе.

Векторы, списки, последовательности – иерархия интерфейсов

Задача – оптимизация состава методов

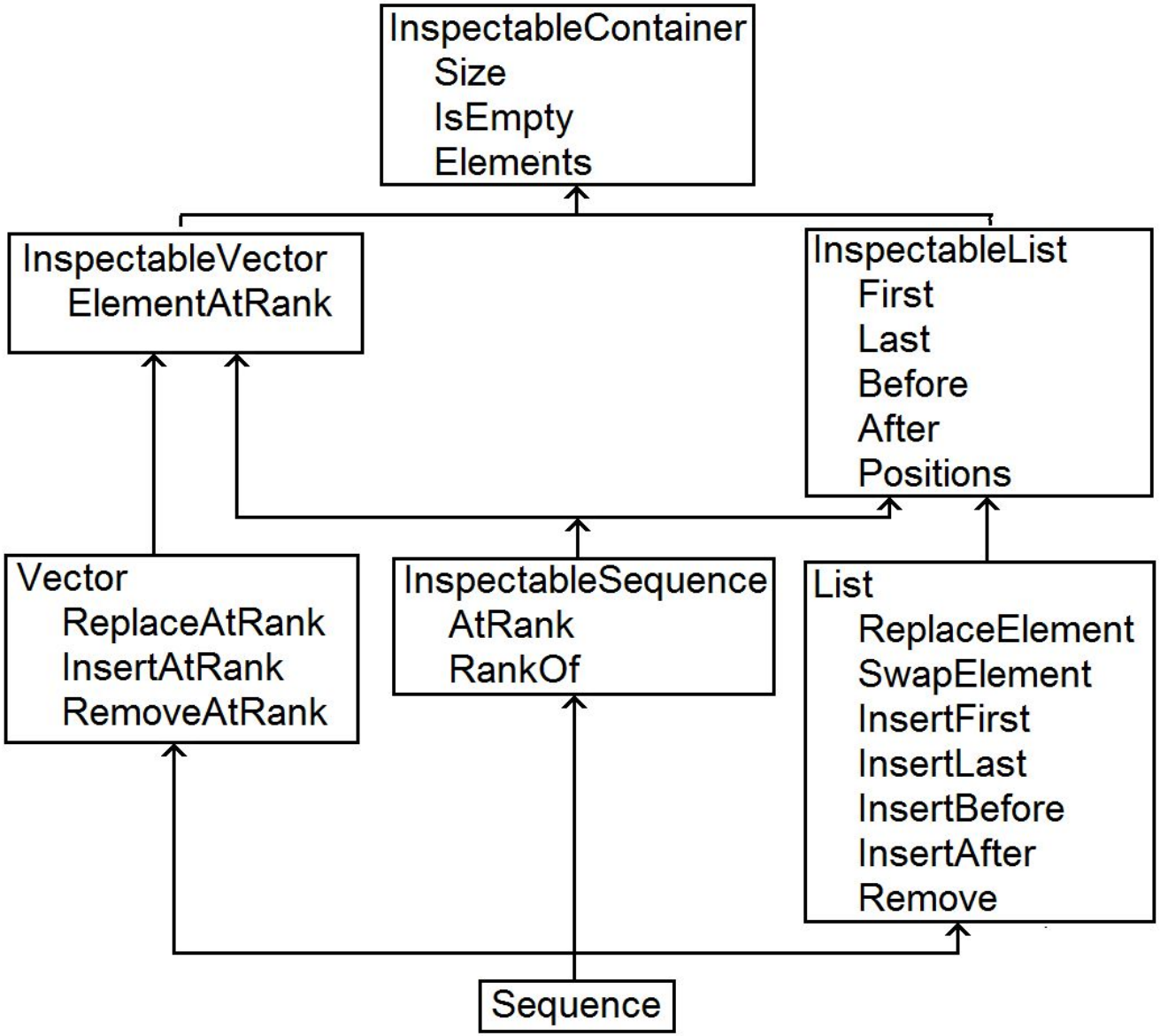
Введем обобщающее понятие «контейнер» («коллекция») и классификацию методов контейнеров:

- ***методы запросов*** возвращают информацию о контейнере;
- ***методы доступа*** возвращают элементы или позиции контейнера;
- ***методы обновления*** изменяют контейнер, добавляя, удаляя элементы или изменяя отношения между элементами.
- ***методы конструктора,*** создающие экземпляр контейнера.

Инспектирующие контейнеры

В таких контейнерах, после их инициализации с помощью конструктора, разрешен доступ «только для чтения». Таким образом, элементы в них защищены от ошибочных или злонамеренных внешних попыток

Структура иерархии последовательностей



Итераторы – АТД «Итератор»

Многие задачи с коллекциями связаны с просмотром всех элементов по порядку.

Итератор - абстрактное представление процесса просмотра коллекции элементов по порядку. Итератор инкапсулирует понятия «место» и «следующий» в коллекциях объектов.

АТД ObjectIterator поддерживает два следующих метода:

HasNext: проверяет наличие оставшихся в итераторе элементов.

Input: нет; Output: логическое значение.

NextObject: возвращает и удаляет следующий элемент итератора.

Input: нет; ***Output:*** объект.

Кроме того, объект-коллекция должен реализовывать метод, который возвращает итератор элементов коллекции (например, GetEnumerator()).

В C# ArrayList реализует поддержку итераторов.

```
public static void printArrayList1(ArrayList aList)
{   IEnumerator iterator = aList.GetEnumerator();
    while (iterator.MoveNext())
    {   Console.WriteLine(iterator.Current);   }
}
public static void printArrayList2(ArrayList aList)
{   foreach(Object o in aList)
    {   Console.WriteLine(o); }
}
```