Изпитна тема № 5: Реализиране на собствени линейни структури от данни

1.1. Дефинира понятието линейна структура от данни (2т.)

Линейна структура от данни представлява начин на съхранение на данните. Данните могат да са числа, букви, обекти и др. Те се съхраняват в някаква структура, т. е. имат някаква наредба едни спрямо други. Линейна е структурата когато данните са наредени в редица (линия). Такива структури от данни са списъци, стекове и опашки, може да се включат и масивите.

1.2. Различава алгоритми за търсене и сортиране (4т.)

Разликата между алгоритмите за търсене и сортиране е, че при търсенето проверяваме елементите един по един докато намерим желания елемент. При сортирането освен търсене има и разменяне на елементи. Например намираме най-малкия елемент и го разменяме с този, който е на първа позиция в редицата. После втория най-малък и го разменяме с този, който е на втора позиция и т. н.

2.1. Посочва видове алгоритми за търсене (2т.)

Има различни алгоритми за търсене. По-известните са: линейно търсене (linear search), двоично търсене (binary search) и интерполационно търсене (interpolation search).

2.2. Демонстрира имплементация на избран алгоритъм за търсене (изберете само един) (6т.)

Линейното търсене

```
int LinearSearch(int[] array, int target)
{
	for(int i = 0; i < array.Length; i++)
	{
		if(array[i] == target)
		{
			return i;
		}
	}
	return -1;
}
```

Линейното търсене има времева сложност O(n). Това означава, че в най-лошия случай ако има 5 елемента в масива ще трябва да се направят 5 проверки. Този вид търсене работи добре при малък брой елементи. Ако има милион елементи ще трябва един по един да провери всеки елемент докато намери търсения. Едно основно предимство на линейното търсене е, че масивът не трябва да е сортиран.

Двоично търсене

```
int BinarySearch(int[] array, int target)
{
	int low = 0;
	int high = array.Length - 1;

	while (low <= high)
	{
		int middle = low + (high - low) / 2;

		if(array[middle] > target)
		{
			high = middle - 1;
		}
		else if(array[middle] < target)
		{
			low = middle + 1;
		}
		else
		{
			return middle;
		}
	}

	return -1;
}
```

Този алгоритъм е с времева сложност O(logn). Колкото повече елементи има масива, толкова по-ефективен е алгоритъмът. Един недостатък е, че масивът трябва да е предварително сортиран преди да започне търсенето.

Интерполационно търсене

```
static int InterpolationSearch(int[] array, int target)
{
	int low = 0;
	int high = array.Length - 1;

	while (target >= array[low] && target <= array[high] && low <= high)
	{
		int probe = low + (high - low) *
		(target - array[low]) / (array[high] - array[low]);

		if(array[probe] > target)
		{
			high = probe - 1;
		}
		else if(array[probe] < target)
		{
			low = probe + 1;
		}
		else
		{
			return probe;
		}
	}
	return -1;
}
```

Този алгоритъм средно е със времева сложност O(log(logn)), но в най-лошия случай е със времева сложност O(n). Може да е по-бърз от двоичното търсене, но може и да е бавен колкото линейното. Работи най-добре когато числата нарастват линейно или са близки едни до други. Ако числата нарастват експоненциално (например всяко следващо число е колкото предишното, умножено на квадрат) тогава алгоритъмът е най-неефективен.

3.1. Различава статична и динамична реализация на линейна структура от данни. (4т.)

Статична реализация на линейна структура от данни представлява реализация, чрез разтеглив масив. Точно така! Масив, който си променя размера. Но масивът не е ли със фиксиран размер? Ами да, но е измислен начин да се разшири чрез използването на втори масив. Динамична реализация представлява свързан списък/стек/опашка. Какво означава това, че са „свързани“. Ще обясним със списък, а стекът и опашката са аналогични. В свързаният списък се пазят възли (nodes). Един възел представлява обект, съдържащ стойност (число, символ, символен низ, др.) и адрес към следващия възел. Сега малко разяснение за паметта на компютъра. Тя се състои от клетки, в които се съхраняват данните. Всяка клетка си има адрес, чрез който тя се достъпва. При заделяне на памет за масив/статичен списък компютърът търси толкова клетки, колкото е размера на масива. Тези клетки, обаче, са последователни. Напр. търсим 5 свободни клетки за масив с 5 елемента. Ако първата клетка е с адрес 426, следващата ще е с адрес 427, третата – адрес 428, четвъртата – адрес 429 и петата – адрес 430. Компютърът търси 5 свободни И последователни клетки, на които да запише масива. При свързаните списъци не е така. Там 5-те свободни клетки може да се намират където си искат в паметта, защото в една клетка се съхранява стойност и адрес към следващата. За това възлите в свързаните списъци са „свързани“.

3.2 Демонстрира имплементация на статична и динамична реализация на избрана линейна структури от данни – разтеглив масив, свързан списък, двойно свързан списък, реализация на стек чрез разтеглив масив и свързан списък, реализация на опашка чрез разтеглив масив и свързан списък. (6т.)

Време е да направим статична и динамична реализация на структурите от данни. Започваме със статичен списък.

Списък

Основни методи на списъка са:
- void Add(object) – добавя елемент в края на списъка
- void Insert(int, object) – добавя елемент на предварително избрана позиция в списъка - void Clear() – изтрива всички елементи от списъка
- bool Contains(object) – проверява дали елементът се съдържа в списъка
- int Remove(object) – премахва съответния елемент и връща индекса му
- T RemoveAt(int) – премахва елемента на дадена позиция и връща премахнатия елемент
- int IndexOf(object) – връща позицията на елемента
- this[int] – индексатор, позволяващ достъп на елементите по подадена позиция

```
public class StaticList<T>
    {
        private const int INITIAL_CAPACITY = 4;

        //полета за масив и за брояч на елементите
        private T[] arr;
        private int count;

        //Създава празен масив с дължина 4
        public StaticList()
        {
            arr = new T[INITIAL_CAPACITY];
            count = 0;
        }

        //this is a getter in C#
        public int Count => count;

        //разширява масива, когато се напълни
        public void GrowIfArrIsFull()
        {
            if (count + 1 > arr.Length)
            {
                T[] extenderArr = new T[arr.Length * 2];
                Array.Copy(arr, extenderArr, count);
                arr = extenderArr;
            }
        }

        //добавя елемент в края на списъка
        public void Add(T item)
        {
            GrowIfArrIsFull();
            arr[count] = item;
            count++;
        }

        //вмъква елемент на дадена позиция
        public void Insert(int index, T item)
        {
            //Проверка за валиден индекс
            if (index > count || index < 0)
            {
                throw new IndexOutOfRangeException("Invalid index: " + index);
            }

            GrowIfArrIsFull();

            //Преместваме част от елементите надясно,
            //за да направим място за новия елемент
            Array.Copy(arr, index, arr, index + 1, count - index);

            //добавяме елемента
            arr[index] = item;
            count++;
        }

        public void Clear()
        {
            arr = new T[INITIAL_CAPACITY];
            count = 0;
```

```
}

//Линейно търсене от началото на масива до броя елементи
public int IndexOf(T item)
{
    for (int i = 0; i < count; i++)
    {
        if (item.Equals(arr[i]))
        {
            return i;
        }
    }

    return -1;
}

//Като IndexOf() но връща true или false
public bool Contains(T item)
{
    int index = IndexOf(item);
    bool found = (index != -1);
    return found;
}

//Връща или задава стойност на зададена от нас позиция
//Работи като при мсив - име_на_списък[индекс]
public T this[int index]
{
    get
    {
        if (index >= count || index < 0)
        {
            throw new ArgumentOutOfRangeException("Invalid index: " + index);
        }

        return arr[index];
    }
    set
    {
        if (index >= count || index < 0)
        {
            throw new ArgumentOutOfRangeException("Invalid index: " + index);
        }

        arr[index] = value;
    }
}

public T RemoveAt(int index)
{
    //Проверка за валиден индекс
    if (index >= count || index < 0)
    {
        throw new ArgumentOutOfRangeException("Invalid index: " + index);
    }

    //запзаваме стойността на елемента, който ще се премахне
    T item = arr[index];
    //Взимаме всички стойности отдясно на елемента за премахване
    //и ги преместваме с едно наляво. Така те закриват елемента за премахване
    Array.Copy(arr, index + 1, arr, index, count - index - 1);
    //последния елемент в опашката се занулява
```

```
            arr[count - 1] = default(T);
            count--;

            return item;
        }

        //Премахва елемент и връща индекса му
        //Използва готовите методи IndexOf() и RemoveAt()
        public int Remove(T item)
        {
            int index = IndexOf(item);
            if (index != -1)
            {
                RemoveAt(index);
            }

            return index;
        }
    }
```

Сега идва ред на динамичната реализация. Тя е малко по-сложна за разбиране.

```
public class DynamicList<T>
    {
        //От този вложен клас - ListNode ще се правят обекти,
        //които представляват възлите с елемент и адрес към следващия възел
        private class ListNode
        {
            //Две полета с автоматични getter и setter (в C# тези полета
            //се наричат свойства)
            public T Element { get; set; } //поле за елемент
            public ListNode NextNode { get; set; }//поле за адрес

            //Конструктор, който се използва, когато списъка е празен
            public ListNode(T element)
            {
                Element = element;
                NextNode = null;
            }
            //Конструктор, който се използва, когато списъка има поне един възел
            public ListNode(T element, ListNode prevNode)
            {
                Element = element; //задава елемент на текущия възел
                prevNode.NextNode = this; //Задава адреса на предишния възел
                                          //да сочи към този
            }
        }

        private ListNode head; //присвоява елемента и адреса на първия възел
        private ListNode tail; //присвоява елемента и адреса на последния възел
        private int count; //брояч колко възела има в списъка

        public int Count => count;

        //Конструктор, който прави празен списък
        public DynamicList()
        {
            head = null;
            tail = null;
            count = 0;
        }

        public void Add(T item)
```

```
{
    if(head == null)
    {
        //Списъка е празен и се задава първия възел
        head = new ListNode(item);
        tail = head;//гогато има само един елемент,
                    //той е първи и последен еднпвременно
    }
    else
    {
        //Списъка не е празен и добавяме възела след последния (tail)
        //Адреса на последния възел сочи към този и този става последен (tail)
        ListNode newNode = new ListNode(item, tail);
        tail = newNode;
    }

    count++;
}

public int IndexOf(T item)
{
    int currentIndex = 0;
    ListNode currentNode = head;

    while (currentNode != null)
    {
        if (ReferenceEquals(currentNode.Element, item))
        {
            return currentIndex;
        }

        currentNode = currentNode.NextNode;
        currentIndex++;
    }

    return -1;
}

//Премахва възел от списъка
//node е възела,който ще се премахва, prevNode е възела преди node
private void RemoveListNode(ListNode node, ListNode prevNode)
{
    count--;
    if (count == 0)
    {
        //списъка става празен - премахват изпразват се полетата,
        //пазещи първия и последния възел
        head = null;
        tail = null;
    }
    else if(prevNode == null)
    {
        //ще се премахне първия възел от списъка - head ще
        //пази в себе си следващия възел
        head = node.NextNode;
    }
    else
    {
        //пренасочваме адресите, така че да се прескача премахнатия възел
        prevNode.NextNode = node.NextNode;
    }
```

```
    //Ако е премахнат последния възел, tail вече ще сочи възела преди него
    if(ReferenceEquals(tail, node))
    {
        tail = prevNode;
    }
}

public T RemoveAt(int index)
{
    if (index >= count || index < 0)
    {
        throw new ArgumentOutOfRangeException("Invalid index: " + index);
    }

    //Започваме от първия възел
    int currentIndex = 0;
    ListNode currentNode = head;
    ListNode prevNode = null;

    //Минаваме към следващия докато не стигнем желания индекс
    while(currentIndex < index)
    {
        prevNode = currentNode;
        currentNode = currentNode.NextNode;
        currentIndex++;
    }

    //Премахваме възела, който е на желания индекс
    RemoveListNode(currentNode, prevNode);

    //Връщаме елемента на премахнатия възел
    return currentNode.Element;
}

public int Remove(T item)
{
    int index = IndexOf(item);
    if(index != -1)
    {
        RemoveAt(index);
    }
    return index;
}

public void Clear()
{
    head = null;
    tail = null;
    count = 0;
}

public bool Contains(T item)
{
    int index = this.IndexOf(item);
    bool found = (index != -1);
    return found;
}

//Връща или задава стойност на зададена от нас позиция
//Работи като при мсив - име_на_списък[индекс]
public T this[int index]
{
```

```
            get
            {
                if(index >= count || index < 0)
                {
                    throw new ArgumentOutOfRangeException("Invalid index: " + index);
                }

                ListNode currentNode = head;

                for(int i = 0; i < index; i++)
                {
                    currentNode = currentNode.NextNode;
                }

                return currentNode.Element;
            }
            set
            {
                if (index >= count || index < 0)
                {
                    throw new ArgumentOutOfRangeException("Invalid index: " + index);
                }

                ListNode currentNode = head;

                for (int i = 0; i < index; i++)
                {
                    currentNode = currentNode.NextNode;
                }

                currentNode.Element = value;
            }
        }
    }
```

Опашка

- void Enqueue(T) – добавя елемент накрая на опашката
- T Dequeue() – взима елемента от началото на опашката и го премахва и връща стойността му
- T Peek() – връща елемента от началото на опашката без да го премахва
- void Clear() – премахва всички елементи от опашката
- bool Contains(T) – проверява дали елемента се съдържа в опашката
- Count – връща броя на елементите в опашката

Статична реализация

```
class StaticQueue<T>
    {
        private const int INITIAL_CAPACITY = 4;
        //Подобно на списък, но имаме 2 нови полета-head и tail
        //Те ще пазят съответно първия и последния индекс на опашката
        private T[] arr;
        private int head;
        private int tail;
        private int count;

        //Празен масив с дължина 4
```

```
public StaticQueue()
{
    arr = new T[INITIAL_CAPACITY];
    head = -1;
    tail = -1;
    count = 0;
}

//this is a getter in C#
public int Count => count;

//разширява масива когато tail сочи към последния индекс
private void GrowIfArrIsFull()
{
    if (tail + 2 > arr.Length)
    {
        T[] extenderArr = new T[arr.Length * 2];
        Array.Copy(arr, extenderArr, tail + 1);
        arr = extenderArr;
    }
}

//Добавя елемент в края на опашката
public void Enqueue(T item)
{
    //разширяваме масива при нужда
    GrowIfArrIsFull();
    //Ако опашката е празна се добавя елемент в началото на масива
    //Той ще е и начало и край на опашката
    if (count == 0)
    {
        arr[count] = item;
        head = count;
        tail = head;
    }
    //Ако не е празна со добавя елемент след последния индекс tail
    //tail ще сочи към индекса на добавения елемент
    else
    {
        arr[tail + 1] = item;
        tail++;
    }

    count++;
}

//изпразва опашката като занулява полетата
public void Clear()
{
    arr = new T[INITIAL_CAPACITY];
    head = -1;
    tail = -1;
    count = 0;
}

//Линейно търсене между head и tail индексите
public bool Contains(T item)
{
    for(int i = head; i <= tail; i++)
    {
        if (item.Equals(arr[i]))
        {
```

```
                return true;
            }
        }

        return false;
    }

    //Връща първия елемент в опашката
    public T Peek()
    {
        if(count == 0)
        {
            throw new InvalidOperationException("Queue is empty!");
        }

        return arr[head];
    }

    //Премахва първия елемент в опашката
    public T Dequeue()
    {
        if (count == 0)
        {
            throw new InvalidOperationException("Queue is empty!");
        }
        //запазваме стойността на първия елемент
        T item = arr[head];

        //ако той е единствения в опашката, опашката става празна
        if(head == tail)
        {
            Clear();
        }
        //Ако не, занулява се стойността на първия елемент
        //head се премества на следващия индекс
        else
        {
            arr[head] = default(T);
            head++;
            count--;
        }

        return item;
    }
}
```

Динамична реализация.

```
class DynamicQueue<T>
    {
        //От този вложен клас - QueueNode ще се правят обекти,
        //които представляват възлите с елемент и адрес към следващия възел
        private class QueueNode
        {
            //Две полета с автоматични getter и setter (в C# тези полета
            //се наричат свойства)
            public T Element { get; set; } //поле за елемент
            public QueueNode NextNode { get; set; }//поле за адрес

            //Конструктор, който се използва, когато опашката е празна
```

```
    public QueueNode(T element)
    {
        Element = element;
        NextNode = null;
    }
    //Конструктор, който се използва, когато опашката има поне един възел
    public QueueNode(T element, QueueNode prevNode)
    {
        Element = element; //задава елемент на текущия възел
        prevNode.NextNode = this; //Задава адреса на предишния възел
                                  //да сочи към този
    }
}

private QueueNode head; //присвоява елемента и адреса на първия възел
private QueueNode tail; //присвоява елемента и адреса на последния възел
private int count; //брояч колко възела има в опашката

//getter
public int Count => count;

//Конструктор, който прави празна опашка
public DynamicQueue()
{
    head = null;
    tail = null;
    count = 0;
}

//добавя възел в края на опашката
public void Enqueue(T item)
{
    if (head == null)
    {
        //Опашката е празна и се задава първия възел
        head = new QueueNode(item);
        tail = head;//гогато има само един елемент,
                    //той е първи и последен еднпвременно
    }
    else
    {
        //Опашката не е празна и добавяме възела след последния (tail)
        //Адреса на последния възел сочи към този и този става последен (tail)
        QueueNode newNode = new QueueNode(item, tail);
        tail = newNode;
    }

    count++;
}

//Показва дали се съдържа възел с даден елемент
//чрез линейно търсене
public bool Contains(T item)
{
    QueueNode currentNode = head;

    while (currentNode != null)
    {
        if (ReferenceEquals(currentNode.Element, item))
        {
            return true;
        }
```

```
                currentNode = currentNode.NextNode;
            }

            return false;
        }

        //Връща елемента на първия възел
        public T Peek()
        {
            if(count == 0)
            {
                throw new InvalidOperationException("Queue is empty!");
            }
            return head.Element;
        }

        //Премахва възел от началото на опашката
        public T Dequeue()
        {
            if (count == 0)
            {
                throw new InvalidOperationException("Queue is empty!");
            }

            //Пазим елемента на възела, който ще премахнем
            T item = head.Element;
            count--;
            if (count == 0)
            {
                //опашката става празна - зануляват се полетата,
                //пазещи първия и последния възел
                head = null;
                tail = null;
            }
            else
            {
                //ще се премахне първия възел от опашката - head ще
                //пази в себе си следващия възел
                head = head.NextNode;
            }

            return item;
        }

        public void Clear()
        {
            head = null;
            tail = null;
            count = 0;
        }
    }
```

Стек

Основни методи:
- void Push(T) – добавя нов елемент на върха на стека
- T Pop() – връща най-горния елемент като го премахва от стека
- T Peek() – връща най-горния елемент без да го премахва
- Count – връща броя на елементите в стека

- void Clear() – премахва всички елементи
- bool Contains(T) – проверява дали елементът се съдържа в стека

Статична реализация

```
class StaticStack<T>
    {
        private const int INITIAL_CAPACITY = 4;

        //полета за масив и за брояч на елементите
        private T[] arr;
        private int count;
        private int top; //Пази индекса на най-горния елемент

        //Създава празен масив с дължина 4
        public StaticStack()
        {
            arr = new T[INITIAL_CAPACITY];
            top = -1;
            count = 0;
        }

        //this is a getter in C#
        public int Count => count;

        //разширява масива, когато се напълни
        public void GrowIfArrIsFull()
        {
            if (count + 1 > arr.Length)
            {
                T[] extenderArr = new T[arr.Length * 2];
                Array.Copy(arr, extenderArr, count);
                arr = extenderArr;
            }
        }

        //добавя елемент на върха на стека
        public void Push(T item)
        {
            GrowIfArrIsFull();
            arr[count] = item;
            top = count;
            count++;
        }

        public void Clear()
        {
            arr = new T[INITIAL_CAPACITY];
            top = -1;
            count = 0;
        }

        //Линейно търсене от началото на масива до броя елементи
        public bool Contains(T item)
        {
            for (int i = 0; i < count; i++)
            {
                if (item.Equals(arr[i]))
                {
                    return true;
                }
            }
```

```
            return false;
        }

        //Връща елемента на върха без да го трие
        public T Peek()
        {
            //Проверка за празен стек
            if (count == 0)
            {
                throw new InvalidOperationException("Stack is empty!");
            }

            return arr[top];
        }

        //Връща и трие елемента на върха
        public T Pop()
        {
            //Проверка за празен стек
            if (count == 0)
            {
                throw new InvalidOperationException("Stack is empty!");
            }

            //запзаваме стойността на елемента, който ще се премахне
            T item = arr[top];

            //най-горния елемент в стека се занулява
            arr[top] = default(T);
            //сега върхът ще е индекса на елемента преди премахнатия
            top--;
            count--;

            return item;
        }
    }
```

Добавя се и се трие само от дясно на масива.

Динамична реализация

```
class DynamicStack<T>
    {
        //От този вложен клас - StackNode ще се правят обекти,
        //които представляват възлите с елемент и адрес към следващия възел
        private class StackNode
        {
            //Две полета с автоматични getter и setter (в C# тези полета
            //се наричат свойства)
            public T Element { get; set; } //поле за елемент
            public StackNode NextNode { get; set; }//поле за адрес

            //Конструктор, който се използва, когато стека е празен
            public StackNode(T element)
            {
                Element = element;
                NextNode = null;
            }
            //Конструктор, който се използва, когато стека има поне един възел
            public StackNode(T element, StackNode nextNode)
            {
```

```
            Element = element; //задава елемент на текущия възел
            NextNode = nextNode; //Задава адреса на текущия възел
                                 //да сочи към следващия
        }
    }

    private StackNode top; //присвоява елемента и адреса на най-горния възел
    private int count; //брояч колко възела има в стека

    //getter
    public int Count => count;

    //Конструктор, който прави празен стек
    public DynamicStack()
    {
        top = null;
        count = 0;
    }

    //добавя възел на върха на стека
    public void Push(T item)
    {
        if (top == null)
        {
            //Стека е празен и се задава първия възел
            top = new StackNode(item);
        }
        else
        {
            //Стека не е празен и добавяме възела след най-горния (top)
            //Адреса на този възел сочи към най-горния и този става най-горен
(top)
            StackNode newNode = new StackNode(item, top);
            top = newNode;
        }

        count++;
    }

    //Показва дали се съдържа възел с даден елемент
    //чрез линейно търсене
    public bool Contains(T item)
    {
        StackNode currentNode = top;

        while (currentNode != null)
        {
            if (ReferenceEquals(currentNode.Element, item))
            {
                return true;
            }

            currentNode = currentNode.NextNode;
        }

        return false;
    }

    //Връща елемента на най-горния възел
    public T Peek()
    {
        if (count == 0)
```

```
            {
                throw new InvalidOperationException("Stack is empty!");
            }
            return top.Element;
        }

        //Премахва и връща елемента на най-горния възел
        public T Pop()
        {
            if (count == 0)
            {
                throw new InvalidOperationException("Stack is empty!");
            }

            //Пазим елемента на възела, който ще премахнем
            T item = top.Element;
            count--;
            //ще се премахне най-горния възел от стека - top ще
            //пази в себе си следващия възел
            top = top.NextNode;

            return item;
        }

        public void Clear()
        {
            top = null;
            count = 0;
        }
    }
```

3.3 Прави заключения за бързодействието на операциите при статична и динамична реализация (8 т.)

Предимството на статичната имплементация е, че като използва масив има по-бърз достъп на елементи (масивът има random access), но при добавяне и премахване на елементи трябва останалите да си изместват позициите и това забавя процеса. Също така статичната имплементация заема повече място в паметта. При динамичната имплементация, за да се достъпи елемент трябва да се започне от първия възел и да се преминава към следващия един по един, докато не се достъпи желания възел. От друга страна при премахване и добавяне на елементи не трябва да се местят възли, а трябва да се пренасочат адресите, което става по-бързо. Също така динамичната имплементация заема по-малко място в паметта.

4.1. Описва и дава примери за основните алгоритми за сортиране – сортиране чрез пряка селекция, сортиране чрез метода на мехурчето, сортиране чрез вмъкване и други сортиращи алгоритми. (2+4 т.)

Има различни методи за сортиране. Трите основни са метода на мехурчето (bubble sort), сортиране с пряка селекция(selection sort) и сортиране чрез вмъкване(insertion sort).

При метода на мехурчето най-големия елемент „изплува“ в най-вдясно на масива, след него „изплува“ втория по големина и т. н. Как става това? Имаме масив [2; 3; 6; 4; 8; 1; 5; 7]. Взимаме първите два елемента(2 и 3) и ги сравняваме. Наредени са правилно. После следващите два (3 и 6). И те са в правилен ред. След това следващите два (6 и 4). Те не са в правилен ред. Разменяме ги. Масива става [2; 3; 4; 6; 8; 1; 5; 7]. Продължаваме с 6 и 8. Те са в правилен ред. 8 и 1 не са. Разменяме ги. Става [2; 3; 4; 6;

1; 8; 5; 7]. Продължаваме по същия начин и така 8 ще изплува на върха. После правим същото нещо отначало без да проверяваме последния елемент, защото той вече е на правилното място. Ще изплува 7 преди 8. Същото нещо отначало без проверка на последните два елемента. И така докато не се сортира масива.
Ето го кода

```
void BubbleSort(int[] array)
        {
            for (int i = 0; i < array.Length - 1; i++)
            {
                for (int j = 0; j < array.Length - 1 - i; j++)
                {
                    if(array[j] > array[j + 1])
                    {
                        int temp = array[j];
                        array[j] = array[j + 1];
                        array[j + 1] = temp;
                    }
                }
            }
        }
```

Този метод е много бавен. Може да се използва за малък брой елементи. Има времева сложност $O(n^2)$.

При метода на пряката селекция се казва, че първия елемент е най-малкия. Сравнява се със всеки след него и ако има някой по-малък, той ще бъде най-малкия. След това най-малкия елемент застава най-вляво в масива. После се казва, че втория елемент е най-малкия. Повтаря се стъпката по-горе и следващия най-малък застава след елемента, който е най-вляво в масива. Така докато масива не се сортира.
Ето го кода

```
void SelectionSort(int[] array)
        {
            for (int i = 0; i < array.Length - 1; i++)
            {
                int min = i;
                for (int j = i; j < array.Length; j++)
                {
                    if(array[j] < array[min])
                    {
                        min = j;
                    }
                }
                int temp = array[i];
                array[i] = array[min];
                array[min] = temp;
            }
        }
```

Въпреки, че е по-бърз от метода на мехурчето, метода на пряката селекция пак е сравнително бавен. Времевата сложност пак е $O(n^2)$.

При метода на вмъкване се взима втория елемент. Той се сравнява със елементите преди него. Ако някой е по-голям се измества наляво. Това продължава докато не се намери някой по-малък или докато не премине всички елементи (тогава никой не е по-малък). После този втори елемент се поставя на мястото, където условието по-горе е приключило. После това се повтаря с третия елемент. После с четвъртия и така докато масива не се сортира.

Ето го кода

```
void InsertionSort(int[] array)
        {
            for (int i = 1; i < array.Length; i++)
            {
                int temp = array[i];
                int j = i - 1;

                while(j >= 0 && array[j] > temp)
                {
                    array[j + 1] = array[j];
                    j--;
                }
                array[j + 1] = temp;
            }
        }
```

Това е най-бързия метод от трите, но пак се счита за бавен. Времевата сложност е $O(n^2)$. По-бързи методи за сортиране са merge sort и quick sort.

Демонстрира имплементацията на избрани алгоритми за сортиране. (6 т.)

5. Обяснява понятието рекурсия и разработва програми с използването на рекурсивни алгоритми (4+6 т.)

Рекурсията най-просто казано е когато метод извиква себе си.

Ето пример за рекурсия с факториел:

```
int Factorial(int num)
{
       if (num == 0)
       {
              return 1;
       }
       else
       {
              return num * Factorial(num - 1);
       }
}
```

Рекурсията се използва като цикъл. Но защо тогава не използваме for или while цикъл? Тя е кратка за писане и лесна за четене. Преди малко споменахме, че при нея може да има разклонения, което е както голям недостатък, така и голямо предимство. Зависи от ситуацията. За четене на данните в дървовидни структури от данни се използва рекурсия. При обикновените цикли разклонения се превят далеч по-трудно. Като цяло рекурсията е мощно средство, но при неправилно използване може да е проблем. Тя не се използва често. Използвайте я, когато е сложно да осъществите дадено действие с обикновен цикъл.