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

Основните алгоритми, използвани в практиката: рекурсия и рекурсивни алгоритми, алгоритми за търсене (линейно, двоично, интерполационно) и сортиране (метод на мехурчето, метод на пряката селекция, сортиране чрез вмъкване и др.) – видове, разлики между алгоритми за търсене и сортиране, имплементация на алгоритми за търсене и сортиране, разработка на рекурсивни алгоритми. Линейна структура от данни – понятие. Статична и динамична реализация на имплементация на структури от данни – разтеглив масив, свързан списък, двойно свързан списък, стек и опашка.

Създаване и/или поправка/допълване на вече съществуващи компютърни програми, описващи линейни структури от данни, рекурсивни, търсещи или сортиращи алгоритми със средствата на програмен език.

**Дидактически материали:** *Компютър с подходяща версия на софтуерното обезпечение и интегрирана среда за разработка от изучаваните. Например Visual Studio, IntelliJ IDEA, PyCharm, Eclipse и др. Задачи и фрагменти от код на програмен език от изучаваните.*

|  |  |
| --- | --- |
| **Критерии за оценяване на изпитна тема № 5** | **Максимален брой точки** |
| 1. Дефинира понятието линейна структура от данни. Различава алгоритми за търсене и сортиране. | 6 |
| 2. Посочва видове алгоритми за търсене. Демонстрира имплементация на избран алгоритъм за търсене. | 8 |
| 3. Различава статична и динамична реализация на линейна структура от данни. 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. Прави заключения за бързодействието на операциите при статична и динамична реализация. | 18 |
| 4. 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. Демонстрира имплементацията на избрани ал- горитми за сортиране. | 18 |
| 5. Обяснява понятието рекурсия и разработва програми с използването на рекурсивни алгоритми. | 10 |
| 6. Анализира фрагмент/и от код и идентифицира и поправя правилно грешките в написания програмен код, така че да реши поставената задача. Допълва кода, ако и когато това е необходимо. | 40 |
| **ОБЩ БРОЙ ТОЧКИ:** | **100** |

**I. Дефинира понятието линейна структура от данни. Различава алгоритми за търсене и сортиране.**

Много често, за решаване на дадена задача се нуждаем да работим с последователности от елементи. Например, за да прочетем тази книга, трябва да прочетем последователно всяка една страница т.е. да обходим последователно всеки един от елементите на множеството от нейните страници. В зависимост от кон­крет­ната задача се налага да прилагаме различни операции върху тази съвкупност от данни. В настоящата тема ще се запознаем с някои от основните представяния на данните в програмирането. Ще видим как при определена задача една структура е по-ефективна и удобна от друга. Ще разгледаме структурите "списък", "стек" и "опашка" както и тяхното приложе­ние. Също така подробно ще се запознаем и с някои от реализациите на тези структури.

1. **Абстрактни структури от данни**

Преди да започнем разглеждането на класовете в C#, имплементиращи някои от най-често използваните структури от данни (като списъци и речници), ще разгледаме понятията **структури от данни** и **абстрактни структури от данни**.

**2. Какво е структура данни?**

Много често, когато пишем програми ни се налага да работим с множество от обекти (данни). Понякога добавяме и премахваме елементи, друг път искаме да ги подредим или да обработваме данните по друг специфичен начин. Поради това са изработени различни начини за съхранение на данните в зависимост от задачата, като най-често между елементите съществува някаква наредба (например обект А е преди обект Б).

В този момент на помощ ни идват **структурите от данни** – **множество от данни организирани на основата на логически и математически закони.** Много често изборът на правилната структура прави програмата много по-ефективна – можем да спестим памет и време за изпълнение.

**3. Какво е абстрактен тип данни?**

Най-общо **абстрактният тип данни (АТД)**дава определена дефиниция (абстракция) на конкретната структура т.е. **определя допустимите опера­ции и свойства**, без да се интересува от конкретната реализация. Това позволява един тип абстрактни данни да има различни реализации и респективно различна ефективност.

**4. Основни структури от данни в програмирането**

Могат ясно да се различат няколко групи структури:

-     **Линейни** – към тях спадат списъците, стековете и опашките

-     **Дървовидни** – различни типове дървета

-     **Речници** – хеш-таблици

-     **Множества**

**5. Линейни структури от данни**

При линейните структури от данни всички елементи са подредени последователно (не е задължително да са последователно наредени в паметта). Всеки елемент в линейна структура от данни има предшественик (предходен елемент) и наследник (следващ елемент).

Линейните структури от данни се разделят допълнително на статични и динамични структури от данни. **Статичните структури от данни** използват за съхранение на елементите масив с фиксиран размер. Някои езици като C# имат възможност да променят динамично размера на масива (всъщност създават нов, по-голям масив и копират елементите в него). В C++ това не е възможно. **Динамичните структури от данни** използват динамичната памет на компютъра (HEAP) и могат да променят динамично своя размер.

Масивът е най-простата линейна структура от данни.

**6. Алгоритми за търсене и сортиране**

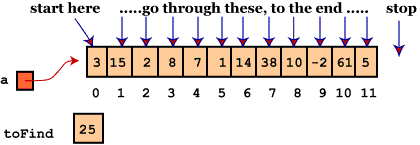
**6.1 Алгоритъм за търсене** e алгоритъм за намиране на елемент с указани свойства в колекция от елементи (в някаква структура от данни, например масив).

Съществуват различни типове алгоритми за търсене.

Критерий за намиране на елемента:  
- търсене на число или текст (пълно съвпадение)  
- търсене на част от текст (непълно съвпадение)

**Ако структурата не е подредена, за намирането на определена стойност в нея се използва последователното (линейно) търсене** [**(linear search)**](https://en.wikipedia.org/wiki/Linear_search)**.**

Проверяват се всички елементите, един по един, от началото до края, последователно, докато открием желания (който отговаря на критерия за търсене).

****

**Ако структурата Е подредена се използват методи за по-бързо търсене.**

**Търсене на елемент в едномерен масив (Теоретична постановка)**

Търсенето на елемент в масив означава да намерим дали зададена стойност се среща като елемент на даден масив. Необходимо е да обходим всички елементи от масива с цикъл (най-често с for) и всеки път да проверяваме дали текущия елемент е равен на търсената стойност.

**Търсене с използване на цикъл for**

int[] array = {19, 17, 1, 83, 23, 33, 2, 3}; //деклариране и инициализация на масив

int length = array.Length; // получаване на дължината на масива

int searchFor = 2; // стойността, която търсим

int foundAtPos = -1; // позицията, на която сме открили стойността

// -1 ако не е открита

// обхождане на масива с цикъл от първия до последния елемент

for (int index = 0; index < length; index++)

{ // проверка дали текущия елемент array[index] е равен на търсения

if (array[index]== searchFor) {

// позицията, на която сме открили търсената стойност

foundAtPos = index;

break; // излизане от цикъла

}

}

if (foundAtPos!=-1) {

Console.WriteLine("Стойността {0} е открита на позиция {1}", searchFor, foundAtPos);

} else {

Console.WriteLine("Стойността {0} не е открита!", searchFor);

}

**Търсене с използване на цикъл while**

int[] array = {19, 17, 1, 83, 23, 33, 2, 3}; //деклариране и инициализация на масив

int length = array.Length; // получаване на дължината на масива

int searchFor = 2; // стойността, която търсим

int foundAtPos = -1; // позицията, на която сме открили стойността

// -1 ако не е открита

// обхождане на масива с цикъл от първия до последния елемент

int index = 0; // инициализация на цикъла

while (index < length)

{

// проверка дали текущия елемент array[index] е равен на търсения

if (array[index]== searchFor) {

foundAtPos = index; // позицията, на която сме открили търсената стойност

break; // излизане от цикъла

}

index++;// актуализация,увеличаване на променливата на цикъла index с 1

}

Ако масивът е от низове, вместо

if (array[index]== searchFor) {

трябва да използваме

if (searchFor.**CompareTo(**array[index]**)==0**) {

**first.**CompareTo**(second)** сравнява два низа и връща integer стойност:   
0 ако са равни;  
<=-1 ако първият низ е по-малък от втория;  
>=1 ако първият низ е по-голям от втория или вторият е null.

**Следният пример илюстира търсенето в масив от низове:**

string[] array = { "Стоян", "Иван", "Ана", "Георги", "Дани", "Мария", "Петко"};

int length = array.Length;

string searchFor = "Мария"; // стойността, която търсим

int foundAtPos = -1; // позицията, на която сме открили стойността

// -1 ако не е открита

// обхождане на масива с цикъл от първия до последния елемент

for (int index = 0; index < length; index++)

{

// проверка дали текущия елемент array[index] е равен на търсения

if (searchFor.CompareTo(array[index]) == 0)

{

// позицията, на която сме открили търсената стойност

foundAtPos = index;

break; // излизане от цикъла

}

}

if (foundAtPos != -1)

{

Console.WriteLine("Стойността {0} е открита на позиция {1}", searchFor, foundAtPos);

}

else

{

Console.WriteLine("Стойността {0} не е открита!", searchFor);

}

Търсене на всички съвпадения:

int[] array = { 19, 17, 1, 83, 23, 33, 2, 3, 2, 2 };

int length = array.Length;

int[] cnt = new int[length];

int currind = 0;

int searchFor = 2; // стойността, която търсим

for (int index = 0; index < length; index++)

{

// проверка дали текущия елемент array[index] е равен на търсения

if (array[index] == searchFor)

{

// позицията, на която сме открили търсената стойност

cnt[currind] = index;

currind++;

}

}

if (currind>0)

{

Console.Write("Стойността {0} е открита на позиции: ", searchFor);

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

{

if (i!=0)

{

Console.Write(", ");

}

Console.Write(cnt[i]);

}

Console.WriteLine();

}

else

{

Console.WriteLine("Стойността {0} не е открита!", searchFor);

}

**6.2 Алгоритъм за сортиране** e алгоритъм за подреждане на елементите в колекция от елементи (в някаква структура от данни) в определена последователност – примерно нарастващ или намаляващ ред.

**II. Посочва видове алгоритми за търсене. Демонстрира имплементация на избран алгоритъм за търсене.**

**Ще илюстрираме три метода за търсене в линейната структура МАСИВ.**

**1. Linear Search = Линейно търсене = O(n)**

Последователно (или линейно) търсене намира определена стойност в списък. Проверява последователно всеки от елементите, един по един, от първия до последния, докато открие желания.

using System;

namespace SearchingMethods

{

class Program

{

// Метод (функция) за намиране на индекса на търсения елемент

public static int Linear<T>(T[] elements, T key) where T : IComparable

{

for (int index = 0; index < elements.Length; index++)

{

// дали поредният елемент е търсеният

if (elements[index].CompareTo(key) == 0)

{

return index; // стойността Е намерена

}

}

return -1; // стойността НЕ Е намерена

}

static void Main(string[] args)

{

int[] arr = {16, 2, 1, 288, 53}; // декларира масив от числа

int ind = Linear<int>(arr, 288); // извиква метода за търсене

if (ind!=-1)

{

Console.WriteLine("Елементът е намерен на позиция {0}", ind);

} else

{

Console.WriteLine("Елементът не е намерен!", ind);

}

}

}

}

**2. Binary Search = Двоично търсене = O(log(n))**

Методът **двоично търсене** се нарича *Binary Search* на английски.

В информатиката, **двоично търсене** е алгоритъм, който се използва за намиране на позицията на елемент или **стойност в подредена структура от данни**, например в предварително сортиран масив. За разлика от последователното търсене, при двоичното търсене са необходими много по-малко стъпки за намиране на търсения елемент. При двоичното търсене, размерът на претърсваното пространство намалява на половина след всяка стъпка.

Когато искаме многократно да търсим различни елементи в даден масив, е по-добре първо да го сортираме и после да използваме метода на двоичното търсене. Това е бърз метод за претърсване на вече сортиран масив.

**Алгоритъм:**

(1) разделянето на претърсваната структура от данни на две равни части;

(2) сравняване на търсената стойност с елемента по средата;

(3) ако са равни => елементът е намерен;

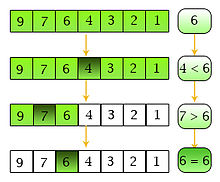
(4) ако не са равни :

(5) ако търсената стойност е по-малка от стойността на средния елемент, търсенето продължава в първата половина на структурата;

(6) ако търсената стойност е по-голяма от стойността на средния елемент, търсенето продължава във втората половина на структурата;

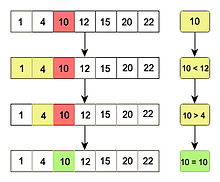
Максималният брой стъпки, необходими на алгоритъма за намирането на елемент в сортиран масив, е O(log(n)). За намиране на елемент в сортиран масив от 1 милион елемента, алгоритъмът ще извърши най-много 20 стъпки.

### Пример 1



Ако в дадена поредица имаме седем числа { 9, 7, 6, 4, 3, 2, 1} и търсим числото 6, на първата стъпка средният елемент ще е 4. Тъй като 4 не съвпада с търсената от нас стойност, алгоритъмът продължава да търси в лявата половина, т.е. 9, 7, и 6. На следващата стъпка отново се определя средният елемент, който е числото 7. Той е по-голям от търсената стойност. Всички числа вляво се елиминират и остава елементът 6. Тъй като той съвпада с търсената от нас стойност, алгоритъмът приключва.

#### Пример 2



Масивът, в който ще се търси, е: {1, 4, 10, 12, 15, 20, 22 }.

Стойността, която трябва да намерим е: 10.

Първо 10 се сравнява с числото, намиращо се в средата на масива – 12. 10 е по-малкото от 12. Търсенето продължава с числата, намиращи в лявата от 12 част от масива: 1, 4, 10.

Първо се сравнява 10 с 1. Търсеното число не е намерено.

Преминава се към следващото число от масива – 4. Числото е по-голямо от числото от масива. Преминава се към следващата стойност.

Сравнява се числото *10* със следващото число – 10. Числото *10* e равно на числото от масива. Така се намира търсеното *число*.

using System;

namespace SearchingMethods

{

class Program

{

public static int Binary<T>(T[] elements, T key) where T : IComparable

{

int start = 0, end = elements.Length - 1; // индекси на първия и последния

while (end >= start)

{

// намира елементът по средата на структурата

int mid = (start + end) / 2;

// сравнява средният елемент с търсената стойност

if (elements[mid].CompareTo(key) > 0)

{

// ако среднуият елемент е по-голям от търсената стойност

// това означава, че търсеният елемент може да бъде само в

// лявата половина, затова:

end = mid - 1;

}

else if (elements[mid].CompareTo(key) < 0)

{

// ако среднуият елемент е по-малък от търсената стойност

// това означава, че търсеният елемент може да бъде само в

// дясната половина, затова:

start = mid + 1;

}

else

{

// елементът Е намерен

return mid;

}

}

// елементът НЕ Е намерен

return -1;

}

static void Main(string[] args)

{

int[] arr = { 1, 2, 16, 53, 288 }; // декларира подреден масив от числа

int ind = Binary<int>(arr, 288); // извиква метода за търсене

if (ind!=-1)

{

Console.WriteLine("Елементът е намерен на позиция {0}", ind);

} else

{

Console.WriteLine("Елементът не е намерен!", ind);

}

}

}

}

**Алгоритъм:**

<1> Намира се индекса на средния елемент (по индекс, т.е. дели се точно на две) в частта от масива, в която се търси (в началото това е целият масив).

<2> Сравнява се стойността на този елемент с търсената.  
Ако тя е по-голяма от търсената, то продължаваме да търсим **в първата половина** на частта от масива, в която търсим в дадения момент.

Ако тя е по-малка от търсената, то продължаваме да търсим **във втората половина** на частта от масива, в която търсим в дадения момент.

И така докато намерим търсения елемент или частта, в която търсим стане 1 елемент – тогава не сме намерили елемента.

**Други имплементации на метода на двоично търсене**

**(за конкретен тип** int**)**

// ВАРИАНТ 1 : като параметри се подават

// array – подреденият масив, в който се търси

// value – стойността, която се търси

// left – първият индекс на частта от масива, в която се търси

// right– последният индекс на частта от масива, в която се търси

public static int binarySearch1(int[] array, int value, int left, int right)

{

while (left <= right)

{

// намиране на индекса на елемента по средата на частта,

// в която се търси (от индекс left до индекс right)

int middle = (left + right) / 2;

// ако средният елемент е равен на търсената стойност

if (array[middle] == value)

{

return middle;

}

else if (array[middle] > value)

{

// ако средният елемент е по-голям от търсената стойност

// търсенето продължава в първата половина

right = middle - 1;

}

else

{

// ако средният елемент е по-малък от търсената стойност

// търсенето продължава във втората половина

left = middle + 1;

}

}

return -1; // ако търсената стойност не е намерена

}

// ВАРИАНТ 2 : като параметри се подават само масивът и търсената стойност

public static int binarySearch2(int[] array, int value)

{

int left = 0, right = array.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

if (array[mid] == value)

{

return mid;

} else if (array[mid] < value)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

static void Main(string[] args)

{

int[] array = { 1, 3, 4, 17, 19, 17, 83, 123, 133, 212 };

int searchFor = 83; // стойността, която търсим

int ind;

ind = binarySearch1(array, searchFor, 0, array.Length-1);

//ind = binarySearch2(array, searchFor);

if (ind >= 0)

{

Console.WriteLine("Стойността {0} в масива е на позиция {1}", searchFor, ind);

}

else

{

Console.WriteLine("Стойността {0} не е открита!", searchFor);

}

}

**3. Interpolation Search = Интерполационно търсене = O(log(log(N))**

Търсене чрез интерполация също е алгоритъм за търсене по даден ключ **в подреден индексиран масив (структура от данни)**.

Методът е подобен на двоичното търсене, но масивът не се разделя наполовина, а стойността, в която се разделя масива на две се намира по интерполационна формула, която връща число, което е осреднено в съответната част от масива.

Формулата е:

**(sortedArray[high] - sortedArray[low])** е разликата между най-голямата и най-малката стойност в съответната част от масива.

Той е по-бърз от двоичния, особено при големи структури от данни.

using System;

namespace SearchingMethods {

class Program

{

public static int Interpolational<T>(T[] elements, T key) where T : IComparable

{

int low = 0, high = elements.Length - 1;

// проверка дали търсената стойност е по-малка от първия или по-голяма

// от последния елемент в претърсваната част от масива (тогава

// търсената стойност не е намерена)

while (elements[low].CompareTo(key) <= 0 && elements[high].CompareTo(key) >= 0)

{

// намира по интерполационна формула индекса на елемента

// който се предполага, че има средна стойност в интервала от

// стойности в масива

int mid = low + (((dynamic)key - (dynamic)elements[low]) \* (high - low)) / ((dynamic)elements[high] - (dynamic)elements[low]);

// сравнява получения среден по стойност елемент с търсената стойност

if (elements[mid].CompareTo(key) > 0)

{

// ако среднуият елемент е по-голям от търсената стойност

// това означава, че търсеният елемент може да бъде само в

// лявата половина, затова:

high = mid - 1;

}

else if (elements[mid].CompareTo(key) < 0)

{

// ако среднуият елемент е по-малък от търсената стойност

// това означава, че търсеният елемент може да бъде само в

// дясната половина, затова:

low = mid + 1;

}

else

{

return mid; // елементът Е намерен

}

}

return -1; // елементът НЕ Е намерен

}

static void Main(string[] args)

{

int[] arr = { 1, 2, 16, 53, 288 }; // декларира подреден масив от числа

int ind = Interpolational<int>(arr, 288); // извиква метода за търсене

if (ind!=-1)

{

Console.WriteLine("Елементът е намерен на позиция {0}", ind);

} else

{

Console.WriteLine("Елементът не е намерен!", ind);

}

}

}

}

**Друга имплементация (за конкретен** тип**):**

public static int InterpolationSearch(int[] sortedArray, int key) {

int low = 0;

int high = sortedArray.Length - 1;

while (sortedArray[low] <= key && sortedArray[high] >= key) {

int mid = low + ((key - sortedArray[low]) \* (high - low))

/ (sortedArray[high] - sortedArray[low]);

if (sortedArray[mid] < key) {

low = mid + 1;

} else if (sortedArray[mid] > key) {

high = mid - 1;

} else {

return mid;

}

}

if (sortedArray[low] == key) return low;

else return -1;

}

**III. Различава статична и динамична реализация на линейна структура от данни. 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.**

Прави заключения за бързодействието на операциите при статична и динамична реализация.

**Статичните реализации на** линейните структури в C# се реализират чрез т.нар. **разтеглив масив**. Това е масив, който може динамично да променя размера си. В C# това е възможно, за разлика от някои други езици като C++. Всъщност се създава напълно нов масив и той се присвоява на указателя на масива. Връзката със стария масив се прекъсва и Garbage Collector-ът ще го изтрие автоматично.

**Статичните реализации на** линейните структури се имплементират по-лесно, но са по-бавни от динамичните.

**1. Статичен списък (реализация чрез разтеглив масив)**

Масивите изпълняват много от условията на АТД списък, но имат една съществена разлика – списъците позволяват добавяне на нови елементи, докато масивите имат фиксиран размер. Въпреки това е възможна реализация на списък чрез масив, който автоматично увеличава размера си при нужда. Такъв списък се нарича **статичен**. Ето една имплемен­тация на статичен списък, реализиран чрез разширяем масив:

|  |
| --- |
| using System;  namespace CustomStaticList  {  public class CustomArrayList  {        private object[] arr; // деклариране на масив        private int count; // брой на елементите в списъка          // връща дължината на масива        public int Count        {              get              {                    return count;              }        }    // първоначален размер на масива        private static readonly int INITIAL\_CAPACITY = 4;    // инициализиране на списъка (празен)        public CustomArrayList()        {              arr = new object[INITIAL\_CAPACITY];              count = 0;        }  // добавяне на елемент в края на списъка  public void Add(object item)  {        Insert(count, item);  }  // добавяне на елемент на определено място в списъка  public void Insert(int index, object item)  {        if (index > count || index < 0)        {              throw new IndexOutOfRangeException("Invalid index: " + index);        }        object[] extendedArr = arr;  // ако няма място в списъка, го прави 2 пъти по-голям        if (count + 1 == arr.Length)        {              extendedArr = new object[arr.Length \* 2];        }  // копира елементите от масива arr в extendedArr  // до елемента с индекс index        Array.Copy(arr, extendedArr, index);        count++; // увеличава броя на елементите с 1    // копира останалите елементи от масива arr в extendedArr        Array.Copy(arr, index, extendedArr, index + 1, count - index - 1);        extendedArr[index] = item; // добавя новия елемент с индекс index        arr = extendedArr; // arr да сочи новосъздадения масив  }  // намиране на индекса на елемент  public int IndexOf(object item)  {        for (int i = 0; i < arr.Length; i++)        {              if (item == arr[i])   {                    return i;              }        }        return -1;  }    // изтриване на списъка  public void Clear()  {        arr = new object[INITIAL\_CAPACITY];        count = 0;  }  // дали зададения обект е елемент от списъка  public bool Contains(object item)  {        int index = IndexOf(item);        bool found = (index != -1);        return found;  }    // да можем да връщаме обект с неговия индекс  public object 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;        }  }  // изтриване на обект по зададен индекс  public object Remove(int index)  {        if (index >= count || index < 0)        {              throw new ArgumentOutOfRangeException(                    "Invalid index: " + index);        }         object item = arr[index];        Array.Copy(arr, index + 1, arr, index, count - index + 1);        arr[count - 1] = null;        count--;        return item;  }  // изтриване на зададения обект  public int Remove(object item)  {        int index = IndexOf(item); // намиране на индекса му        if (index == -1)        {              return index;        }          Array.Copy(arr, index + 1, arr, index, count - index + 1);        count--;          return index;  }  public static void Main()  {        CustomArrayList shoppingList = new CustomArrayList();        shoppingList.Add("Milk");        shoppingList.Add("Honey");        shoppingList.Add("Olives");        shoppingList.Add("Beer");        shoppingList.Remove("Olives");        Console.WriteLine("Трябва да купим:");        for (int i = 0; i < shoppingList.Count; i++)        {              Console.WriteLine(shoppingList[i]);        }        Console.WriteLine("Да купим ли хляб? " +              shoppingList.Contains("Bread"));  }  }  } |

**2. Свързан списък (динамичен списък)**

Статичният списък има един сериозен недостатък – операциите добавяне и премахване от вътрешността на списъка изискват **пренареждане на елементите**. При често добавяне и премахване (особено при голям брой елементи) това може да доведе до ниска **производителност**.

В такива случаи се използват т. нар. **свързани списъци**. Разликата

при тях е в структурата на елементите – докато при статичния списък всеки от елементите съдържа само конкретния обект, при динамичния списък всеки от елементите пази информация за следващия елемент.

Ето как изглежда един примерен свързан списък в паметта:

[clip_image003](https://introprogramming.info/wp-content/uploads/2011/07/clip_image0037.gif)

Ето една примерна реализация на динамичен списък с указател към следващия елемент.

using System;

public class DynamicList

{

private class Node // класът, съхраняващ данните

{

private object element; // данни за един елемент

private Node next; // указател към следващия елемент

public object Element

{

get { return element; }

set { element = value; }

}

public Node Next

{

get { return next; }

set { next = value; }

}

public Node(object element, Node prevNode)

{

this.element = element;

prevNode.next = this;

}

public Node(object element)

{

this.element = element;

next = null;

}

} // край на класа, съхраняващ данните (обекта)

private Node head; // указател към първия елемент на списъка

private Node tail; // указател към последния елемент на списъка

private int count; // брой елементи в списъка

// Конструктор

public DynamicList()

{

this.head = null;

this.tail = null;

this.count = 0;

}

// добавя елемент в края на списъка

public void Add(object item)

{

if (head == null) // ако списъкът е празен

{

head = new Node(item);

tail = head;

}

else

{

// ако списъкът не е празен добавя елемента най-отзад

Node newNode = new Node(item, tail);

tail = newNode;

}

count++; // увеличава броя на елементите

}

// премахва елемент от списъка с индекс

public object Remove(int index)

{

if (index >= count || index < 0)

{

throw new ArgumentOutOfRangeException("Invalid index: " + index);

}

// намира елемента със зададения индекс чрез обхождане на списъка

int currentIndex = 0;

Node currentNode = head;

Node prevNode = null;

while (currentIndex < index)

{

prevNode = currentNode;

currentNode = currentNode.Next;

currentIndex++;

}

// премахва елемента

count--;

if (count == 0)

{

head = null;

}

else if (prevNode == null)

{

head = currentNode.Next;

}

else

{

prevNode.Next = currentNode.Next;

}

// намира последния елемент

Node lastElement = null;

if (this.head != null)

{

lastElement = this.head;

while (lastElement.Next != null)

{

lastElement = lastElement.Next;

}

}

tail = lastElement;

return currentNode.Element;

}

public static void Main()

{

DynamicList shoppingList = new DynamicList();

shoppingList.Add("Мляко");

shoppingList.Add("Мед");

shoppingList.Add("Маслини");

shoppingList.Add("Бира");

shoppingList.Remove("Маслини");

Console.WriteLine("Какво трябва да купим? ");

for (int i = 0; i < shoppingList.Count; i++)

{

Console.WriteLine(shoppingList[i]);

}

Console.WriteLine("Трябва ли да купим хляб? " +

shoppingList.Contains("Хляб")); // в списъка нямаме Хляб

}

}

**3. Стек**

Типичен пример за стек е стойката за CD/DVD дискове на картинката по-долу. Можете да извадите само най-горния диск. За да извадите първия, ще трябва да извадите всички, които са над него.  
Т.е. единствените операции, които можете да правите са:

- добавяне най-отгоре на диск;

- изваждане на най-горния диск;



Точно тази конструкция представлява стекът – можем да добавяме елементи най-отгоре и да извличаме последния добавен елемент, но не и предходните (които са затрупани под него). Стекът е често срещана и използвана структура от данни. Стек се използва и вътрешно от C# виртуалната машина за съхранение на променливите в програмата и параметрите при извикване на метод.

**3.1 Абстрактна структура данни "стек"**

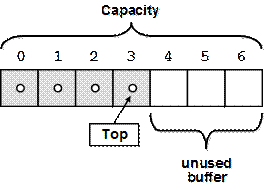
**Стекът** представлява структура от данни с поведение **"последният влязъл първи излиза"** (**LIFO** – **L**ast **I**n **F**irst **O**ut).

Структурата от данни стек също може да има различни реализации, но ние ще се спрем на двете основни – динамичната и статичната реали­зация.

**3.2 Статичен стек (реализация с масив)**

Както и при статичния списък можем да използваме масив за съхраняване на елементите на стека. Ще пазим индекс или указател, който сочи към елемента, който се намира на върха. Обикновено при запълване на масива следва заделяне на двойно повече памет, както това се случва при статичния списък.

Ето как можем да си представим един статичен стек:

[](https://introprogramming.info/wp-content/uploads/2011/07/clip_image0202.gif)

Както и при статичния масив се поддържа свободна буферна памет с цел по-бързо добавяне.

Стекът има **капацитет (Capacity)**. При запълването му се заделя още толкова памет. Има и указател към върха (**Top**)

using System;

namespace CustomStaticStack

{

class StaticStack

{

private object[] arr; // деклариране на масив

private int count; // брой на елементите в стека

// връща дължината на масива

public int Count

{

get

{

return count;

}

}

// първоначален размер на масива

private static readonly int INITIAL\_CAPACITY = 5;

// инициализиране на списъка (празен)

public StaticStack()

{

arr = new object[INITIAL\_CAPACITY];

count = 0;

}

// добавяне на елемент в края на списъка

public void Push(object item)

{

object[] extendedArr = arr;

if (count + 1 == arr.Length)

{ // ако няма място в стека, го прави 2 пъти по-голям

extendedArr = new object[arr.Length \* 2];

// копира елементите от масива arr в extendedArr

Array.Copy(arr, extendedArr, count - 1);

}

count++; // увеличава броя на елементите с 1

extendedArr[count - 1] = item; // добавя новия елемент на последно място

arr = extendedArr; // arr да сочи новосъздадения масив

}

// изтриване на стека

public void Clear()

{

arr = new object[INITIAL\_CAPACITY];

count = 0;

}

// дали зададения обект е елемент от стека

public bool Contains(object item)

{

for (int i = 0; i < arr.Length; i++)

{

if (item == arr[i])

{

return true;

}

}

return false;

}

// изваждане на елемент от стека

public object Pop()

{

if (count == 0) { return -1; }

object item = arr[count-1];

arr[count - 1] = null;

count--;

return item;

}

// връща последния елемент от стека без да го премахва

public object Peek()

{

if (count == 0) { return -1; }

object item = arr[count-1];

return item;

}

public static void Main()

{

StaticStack shoppingList = new StaticStack();

shoppingList.Push("Milk");

shoppingList.Push("Honey");

shoppingList.Push("Olives");

shoppingList.Push("Beer");

Console.WriteLine("На върха на стека е : {0}", shoppingList.Peek());

Console.WriteLine("Да купим ли хляб? " + ((shoppingList.Contains("Bread"))?"Да":"Не"));

Console.WriteLine("Да купим ли мляко? " + ((shoppingList.Contains("Milk")) ? "Да" : "Не"));

while (shoppingList.Count>0)

{

Console.WriteLine(shoppingList.Pop());

}

}

}

}

**3.3 Свързан стек (динамична реализация)**

За динамичната реализация ще използваме елементи, които пазят, освен обекта, и указател към елемента, който се намира "по-долу". Тази реализация решава ограниченията, които има статичната реализация както и необходимостта от разширяване на масива при нужда:

[clip_image022](https://introprogramming.info/wp-content/uploads/2011/07/clip_image0221.gif)

Когато стекът е празен, върхът има стойност **null**. При добавяне на нов елемент, той се добавя на мястото, където сочи върхът, след което върхът се насочва към новия елемент. Премахването става по аналогичен начин.

using System;

namespace CustomLinkedStack

{

class LinkedStack

{

// клас за елементите на стека (в случая са цели числа)

private class Node

{

private int element; // стойност на елемента

private Node prev; // указател към предходния елемент

// свойство за достъп/инициализация на стойност на елемент

public int Element

{

get { return element; }

set { element = value; }

}

// свойство за достъп/инициализация на указателя към предходния

// елемент

public Node Prev

{

get { return prev; }

set { prev = value; }

}

// конструктор (добавяне елемент)

public Node(int item)

{

this.element = item;

prev = null;

}

}

private Node top; // указател към върха на стека

private int count; // брой елементи в стека

// Конструктор на стека

public LinkedStack()

{

this.top = null;

this.count = 0;

}

// свойство, което връща броя на елементите в стека <object>.Count

public int Count()

{

return count;

}

// добавя елемент на върха на стека

public void push(int item)

{

if (top == null) {

top = new Node(item); // ако стекът е празен

}

else

{ // ако стекът не е празен

Node newNode = new Node(item);

// свързва добавения елемент с този на върха на стека

newNode.Prev = top;

top = newNode; // добавеният елемент застава на върха на стека

}

count++;

}

// изважда елемент от списъка и го връща като стойност

public int pop()

{

if (count==0)

{

return -1; // връща 1 ако стека е празен

}

// взема стойността на елемента на върха на стека

int ret = top.Element;

top = top.Prev; // премахва елемента от върха на стека

count--; // намалява броя на елементите в стека

return ret;

}

// връща като стойност най-горния елемент от списъка без да го премахва

public int gettop()

{

if (count == 0)

{

return -1; // връща 1 ако стека е празен

}

return top.Element;

}

// Изчистване на стека

public void emptyStack()

{

top = null;

count = 0;

}

// Разпечатване на стека без да се изтриват елементите му

public void printStack()

{

if (count==0)

{

Console.Write("Списъкът е празен !");

return;

}

Node tmpNode = top; // започва от елемента на върха на стека

bool first = true;

Console.Write("Разпечатване на списъка: ");

while (tmpNode != null)

{

if (first) first = false;

else Console.Write(", ");

Console.Write(tmpNode.Element);

tmpNode = tmpNode.Prev; // следващият елемент от стека

}

Console.WriteLine();

}

}

class Program

{

static void Main(string[] args)

{

var myStack = new LinkedStack();

myStack.push(1); // добавя 1 в стека

myStack.push(2); // добавя 2 в стека

myStack.push(3); // добавя 3 в стека

myStack.push(4); // добавя 4 в стека

myStack.push(5); // добавя 5 в стека

myStack.push(6); // добавя 6 в стека

myStack.printStack(); // разпечатва стека

Console.WriteLine("Елементът на върха е: {0} ", myStack.pop());

myStack.printStack();

Console.WriteLine("Елементът на върха е: {0} ", myStack.gettop());

myStack.printStack();

myStack.emptyStack(); // изчиства стека

myStack.printStack();

}

}

}

**4. Опашка**

Структурата "**опашка**" е създадена да моделира опашки, като например опашка от чакащи документи за принтиране, чакащи процеси за достъп до общ ресурс и други. Такива опашки много удобно и естествено се моделират чрез структурата "опашка". В опашките **можем да добавяме елементи само най-отзад и да извличаме елементи само най-отпред**.

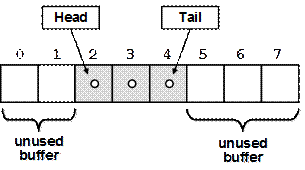
**Абстрактна структура данни "опашка"**

Абстрактната структура опашка изпълнява условието **"първият влязъл първи излиза"** (**FIFO** – **F**irst **I**n **F**irst **O**ut). Добавените елементи се нареждат в края на опашката, а при извличане поредният елемент се взима от началото (главата) й.

Както и при списъка за структурата от данни опашка отново е възможна статична и динамична реализация.

**4.1 Статична опашка (реализация с масив)**

В статичната опашка отново ще използваме масив за пазене на данните. При добавяне на елемент той се добавя на индекса, който следва края, след което края започва да сочи към ново добавения елемент. При премахване на елемент се взима елементът, към който сочи главата, след което главата започва да сочи към следващия елемент. По този начин опашката се придвижва към края на масива. Когато стигне до края, при добавяне на нов елемент той се добавя на първо място. Ето защо тази имплементация се нарича още **зациклена опашка**, тъй като мислено залепяме началото и края на масива и опашката обикаля в него:

[](https://introprogramming.info/wp-content/uploads/2011/07/clip_image0262.gif)

**Зациклената опашка е реализация на опашка с помощта на т.нар. разтеглив масив. Има следните възможности:**

-     **Enqueue(T)** – добавя елемент накрая на опашката

-     **Dequeue()** – връща елемента от началото на опашката и го премахва

-     **Peek()** – връща елементът от началото на опашката без да го премахва

-     **Clear()** – премахва всички елементи от опашката

-     **Contains(Т)** – проверява дали елемента се съдържа в опашката

using System;

namespace CustomStaticQueue

public class CustomArrayQueue

{

      private object[] arr; // деклариране на масив

      private int count; // брой на елементите в опашката

       public int Count // връща дължината на масива

      {

            get

            {

                  return count;

            }

      }

// първоначален размер на масива

      private static readonly int INITIAL\_CAPACITY = 10;

// инициализиране на опашката (празна)

      public CustomArrayList()

      {

            arr = new object[INITIAL\_CAPACITY];

            count = 0;

      }

// добавяне на елемент в края на опашката

public void Enqueue (object item)

{

      object[] extendedArr = arr;

// ако няма място в списъка, го прави 2 пъти по-голям

      if (count + 1 == arr.Length)

      {

            extendedArr = new object[arr.Length \* 2];

// копира елементите от масива arr в extendedArr

      Array.Copy(arr, extendedArr, count);

      }

      count++; // увеличава броя на елементите с 1

      extendedArr[count-1] = item; // добавя новия елемент

      arr = extendedArr; // arr да сочи новосъздадения масив

}

// изтриване на опашката

public void Clear()

{

      arr = new object[INITIAL\_CAPACITY];

      count = 0;

}

// дали зададения обект е елемент от опашката

public bool Contains(object item)

{

if (count==0) { // празна опашка

return  false;

}

// обхождане на опашката

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

      {

            if (item == arr[i]) // ако открие елемента

            {

                  return true;

            }

      }

      return false;

}

// връща елемента от началото на опашката и го премахва

public object Dequeue ()

{

      if (count== 0)

      {

return null;

      }

      object item = arr[0];

// премахва елемента от началото на опашката

      Array.Copy(arr, 0, arr, 1, count - 1);

      count--; // намалява броя на елементите с 1

      return item;

}

// връща елемента от началото на опашката без да го премахва

public object Peek ()

{

      if (count== 0)

      {

return null;

      }

      object item = arr[0];

      return item;

}

public static void Main()

{

      CustomArrayQueue qe = new CustomArrayQueue();

      qe. Enqueue ("Мляко"); // добавя в опашката

      qe. Enqueue ("Мед"); // добавя в опашката

      qe. Enqueue ("Маслини"); // добавя в опашката

      qe. Enqueue ("Бира"); // добавя в опашката

// Разпечатва съдържанието на опашката

      for (int i = 0; i < qe.Count; i++)

      {

            Console.WriteLine(qe.Dequeue());

      }

}

}

**4.2 Свързана опашка (динамична реализация)**

Динамичната реализация на опашката много прилича на тази на свързания списък. Елементите отново съдържат две части – обекта и указател към предишния елемент:

[clip_image028](https://introprogramming.info/wp-content/uploads/2011/07/clip_image0282.gif)

Тук обаче елементите се добавят в края на опашката, а се вземат от главата, като нямаме право да взимаме или добавяме елементи на друго място.

using System;

namespace CustomDynamicQueue

{

public class DynamicQueue

{

private class Node // класът, съхраняващ данните

{

private object element; // данни за един елемент

private Node next; // указател към следващия елемент

public object Element

{

get { return element; }

set { element = value; }

}

public Node Next

{

get { return next; }

set { next = value; }

}

public Node(object element, Node prevNode)

{

this.element = element;

prevNode.next = this;

}

public Node(object element)

{

this.element = element;

next = null;

}

} // край на класа, съхраняващ данните (обекта)

private Node head; // указател към първия елемент на опашката

private Node tail; // указател към последния елемент на опашката

private int count; // брой елементи в опашката

// Конструктор

public DynamicQueue()

{

this.head = null;

this.tail = null;

this.count = 0;

}

public int Count

{

get

{

return this.count;

}

}

// добавя елемент в края на опашката

public void Enqueue(object item)

{

if (head == null) // ако опашката е празна

{

head = new Node(item);

tail = head;

}

else

{

// ако опашката не е празна добавя елемента най-отзад

Node newNode = new Node(item, tail);

tail = newNode;

}

count++; // увеличава броя на елементите

}

// връща първия елемент от опашката и го премахва от опашката

public object Dequeue()

{

if (count == 0)

{

return -1;

}

// намира първия елемент

Node currentNode = head;

head = currentNode.Next;

count--;

return currentNode.Element;

}

// връща първия елемент от опашката без да го премахва от опашката

public object Peek()

{

if (count == 0)

{

return -1;

}

// намира първия елемент

Node currentNode = head;

return currentNode.Element;

}

// дали елемент се съдържа в списъка

public bool Contains(object item)

{

// намира елемента със зададения индекс чрез обхождане на списъка

Node currentNode = head;

while (currentNode.Next != null)

{

if ((currentNode.Element).Equals(item))

{

return true;

}

currentNode = currentNode.Next;

}

return false;

}

public static void Main()

{

DynamicQueue shoppingList = new DynamicQueue();

shoppingList.Enqueue("Мляко");

shoppingList.Enqueue("Мед");

shoppingList.Enqueue("Маслини");

shoppingList.Enqueue("Бира");

// в списъка нямаме Хляб

Console.WriteLine("Да купим ли хляб? " + ((shoppingList.Contains("Хляб")) ? "Да" : "Не"));

// в списъка имаме Мед

Console.WriteLine("Да купим ли мед? " + ((shoppingList.Contains("Мед")) ? "Да" : "Не"));

Console.WriteLine("Какво трябва да купим? ");

while (shoppingList.Count>0)

{

Console.WriteLine(shoppingList.Dequeue());

}

}

}

}

**4. 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. Демонстрира имплементацията на избрани алгоритми за сортиране.**

**4.1 Понятие за сортиране**

“Има много **най-добри** методи, според това какво ще се сортира, на какъв компютър и с каква цел.” ~ D.E.Knuth

Често при работа с големи еднотипни данни се налага въвеждането на някаква наредба с цел по-лесната им обработка. Ако елементите са подредени, това би могло да ни даде значително по-ефективен алгоритъм за търсене в сравнение със случая, когато данните не са подредени.

**Прието е процесът на пренареждане (пермутиране) по подходящ начин на елементите на някакво множество от обекти в определен ред да се нарича *сортиране*.**

Сортиране във възходящ ред (ASCENDING, ASC): 1,2,3,4,5,…

Сортиране във низходящ ред (DESCENDING, DESC): …,5,4,3,2,1

Сортирането е основна дейност с широка сфера на приложение: речници, телефонни указатели, справочни индекси и въобще навсякъде, където се налага бързо търсене и намиране на различни обекти.

Сортирането е изключително широко понятие и в зависимост от типа на сортираните данни може да се реализира по най-разнообразни начини.

**4.2 Видове сортиране**

Съществуват различни класификации на алгоритмите за сортиране.

Може би най-популярната е в зависимост от местонахождението на данните.

**a) Външно и вътрешно сортиране**

Изхождайки от този критерий, различаваме ***вътрешно***(данните се намират и се сортират **в оперативната памет на компютъра RAM** и най-често е възможен директен достъп до произволен елемент на множеството) и ***външно***(данните се намират и сортират във външната памет на компютъра и достъпът до тях става най-често строго последователно, като се започва от първия елемент).

Външното сортиране се използва, когато нямаме достатъчно RAM памет, за да заредим обектите там и да ги подредим.

**б) Според извършваната операция**

В зависимост от операцията, извършвана над елементите, различаваме сортиране чрез ***сравнение***(най-често с помощта на <, > и ==) и чрез ***функция върху елементите***.

**в) Множествено сортиране (по няколко признака)**

**Първо се сортира по най-маловажния критерий (признак), след това по следващия по важност и т.н. Последно се сортира по най-важния критерий.**

**г) Стабилност на сортирането**

* Стабилни сортиращи алгоритми: Запазват подредбата на еднаквите елементи. Ако два елемента след сравнение са равни, редът им един спрямо друг се запазва.
* Нестабилни сортиращи алгоритми: Редът на следване на еднаквите елементи може да се промени спрямо първоначалния.

**4.3 Изисквания (за сравнение на методите)**

Основно изискване към алгоритмите за сортиране е минимален разход на ***допълнителна памет***. Друго важно изискване е за **минимален брой извършвани сравнения и размени на елементи**. Обикновено сортирането се извършва чрез проста размяна на място на два елемента на масива.

* Минимално количество RAM памет и бързодействие на PC
* Бързина (време за изпълнение или брой операции)

**4.4 Метод на пряката селекция (Selection Sort) и имплементация.**

**Алгоритъм на метода на пряката селекция**

1. Намира се най-малкия елемент в масива(списъка) и се сравнява с елемента на първа позиция
2. Разменя го с елемента на първа позиция (ако е по-малък от него)
3. Повтаря горните две стъпки за масива без първия елемент
4. Повтаря горните две стъпки за масива без първите два елемента
5. и т.н.

Масивът(списъкът) се разделя на две половини – подредена и неподредена. При всяко обхождане на неподредената част на масива се намира минималния елемент и той се разменя с първия от неподредената част, след което подредената част нараства с 1 елемент, а неподредената – намалява с 1 елемент.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 31 | 34 | 12 | 22 | 11 | | Най-малкият елемент е 11. Разменят се 11 и 31. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 11 | 34 | 12 | 22 | 31 | | Най-малкият оставащ елемент е 12. Разменят се 12 и 34. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 11 | 12 | 34 | 22 | 31 | | Вече имаме една част от списъка, която е сортирана. Сега се разменят 22 и 34. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 11 | 12 | 22 | 34 | 31 | | Разменят се 31 и 34. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 11 | 12 | 22 | 31 | 34 | | Списъкът е сортиран |

Методът на пряката селекция не е труден да се анализира в сравнение с други сортиращи алгоритми. За да намерим най-малкият елемент се изисква да сканираме всички **n на брой елементи** (това отнема *n* – 1 сравнения) и след това да го разменим с елемента на първата позиция на несортираната част.

За да намерим следващият най-малък елемент се изисква да сканираме оставащите *n* – 1 елементи и така нататък. Общият брой сравнения е равен на (*n* − 1) + (*n* − 2) + ... + 2 + 1 = (*n – 1)*(*n* − 2) / 2 = Θ(*n*2) /сложност по Big-O нотацията/ според формулата за сбор на аритметична прогресия.

Всяко от тези *n* – 1 преминавания по масива изисква една размяна на елементи (последният елемент е вече на правилното място).

**Имплементация (сортиране във възходящ ред)**

using System;

namespace CustomSortingMethods

{

public class CustomSelectionSort

{

// Отпечатване на масив

public static void printArray(int[] array)

{

for (int index = 0; index < array.Length; index++)

{

Console.WriteLine(array[index]);

}

Console.WriteLine();

}

// Сортиране на масив чрез пряка селекция (ВАРИАНТ 1)

// намиране на минималния елемент и размяна с първия

// същото се прави с частта от масива без първия елемент

public static void SelectSort1(int[] array)

{

int k; // индексът на минималния елемент в частта, която обхождаме

int min; // стойността на минималния елемент

int len = array.Length; // дължината на масива

// цикъл за обхождане на масива

for (int i = 0; i < len-1; i++)

{

// намиране на минималния елемент в частта от масива,   
 // започваща от индекс i

// предполагаме, че в началото това е елементът с индекс i

min = array[i]; // текущ минимум

k = i; // индексът на текущия минимум

// започваме да сравняваме текущия минимален елемент

// със следващите елементи с индекси: i+1, i+2, …., len-1

for (int j = i+1; j < len; j++)

{

if (array[j] < min)

{

// ако текущият елемент е по-малък от минималния, той става

// минимален и в k се запомня индекса му

k = j;

min = array[j];

}

}

// Тук приключва поредното обхождане на масива

// Сега трябва да разменим намерения минимален елемент с

// първия от частта на масива, в която търсим (която обхождаме)

// той е с индекс i

array[k] = array[i];

array[i] = min;

}

}

// Сортиране на масив чрез пряка селекция (ВАРИАНТ 2)

// намиране на минималния елемент и размяна с първия

// същото се прави с частта от масива без първия елемент

// НЕ СЪХРАНЯВА МИНИМАЛНИЯ ЕЛЕМЕНТ, А САМО

// ИНДЕКСА МУ

public static void SelectSort2(int[] array)

{

int minind; // индексът на минималния елемент

// допълнителна променлива, използвана при размяна на елементите

int swap;

int len = array.Length; // дължината на масива

// цикъл за обхождане на масива

for (int i = 0; i < len-1; i++)

{

// намиране на минималния елемент в частта от масива,   
 // започваща от индекс i

// предполагаме, че в началото това е елементът с индекс i

minind = i; // индексът на текущия минимум

for (int j = i+1; j < len; j++)

{

if (array[j] < array[minind])

{

// ако текущият елемент е по-малък от минималния, той става

// минимален и в minind се запомня индекса му

minind = j;

}

}

// Тук приключва поредното обхождане на масива

// Сега трябва да разменим намерения минимален елемент с

// първия от частта на масива, в която търсим (която обхождаме)

// той е с индекс i

swap = array[minind];

array[minind] = array[i];

array[i] = swap;

}

}

static void Main(string[] args)

{

int[] arr = { 2, 53, 288, 1, 18, -2}; // декларира масив от числа

SelectSort1(arr); // сортиране на масива по ВАРИАНТ 1

// SelectSort2(arr); // сортиране на масива по ВАРИАНТ 2

printArray(arr); // разпечатване на масива

}

}

}

**Сравнение на обекти в C#**

В горния пример сортирахме числа. В този случай използваме операциите за сравнение (>,<,>=,<=,==, !=) на числа. Понякога обаче се налага да сортираме низове или други обекти. Не можем да използваме операциите за сравнение изброени по-горе. За сравнение на обекти в C# се използва методът **CompareTo()**.

За низове:

**string str1** = "abc", **str2**="bcd";

if (**str1**.**CompareTo**(**str2**)<0) {

Console.WriteLine("str1 е по-малък от str2");

} else

if (str1.CompareTo(str2)>0) {

Console.WriteLine("str1 е по-голям от str2");

} else

if (str1.CompareTo(str2)==0) {

Console.WriteLine("str1 е равен на str2");

}

**4.5 Метод на мехурчето (Bubble Sort).**

**Алгоритъм (сортиране чрез размяна на съседни елементи)**

<1>Масивът се обхожда **n-1 пъти**, където n е размерът на масива, като се сравняват всеки два съседни елемента.   
Ако има два съседни елемента, първият от които е по-голям от втория, им се разменят местата (при сортиране по възходящ ред).  
В края на това първо преминаване през масива на последно място „изплува“ най-големият елемент

<2> Същото се прави с масива без последния елемент и т.н.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 46 | 60 | 56 | 81 | 16 | | Първо се сравняват 46 и 60. 46<60 така че не се прави размяна. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 46 | 60 | 56 | 81 | 16 | | 60>56 им се разменят местата: 46 56 60 81 16 |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 46 | 56 | 60 | 81 | 16 | | Сега се сравняват 60 и 81. 60<81 така че не се прави смяна. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 46 | 56 | 60 | 81 | 16 | | 81>16 така че трябва да се разменят местата им: 46 56 60 16 81 |
| Така след първото цялостно преминаване през масива на последно място  застава най-големият елемент : 81   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 46 | 56 | 60 | 16 | 81 |   Сега трябва да направим същото с елементите на масива без последния 81,  тъй като той вече е на мястото си (на 5-то място). | |
| |  |  |  |  | | --- | --- | --- | --- | | 46 | 56 | 60 | 16 | | 46<56 -> няма размяна |
| |  |  |  |  | | --- | --- | --- | --- | | 46 | 56 | 60 | 16 | | 56<60 -> няма размяна |
| |  |  |  |  | | --- | --- | --- | --- | | 46 | 56 | 60 | 16 | | 60>16 -> трябва да ги разменим |
| |  |  |  |  | | --- | --- | --- | --- | | 46 | 56 | 16 | 60 | | 60 застава на 4-то място |
| **… и така продължаваме до:** | |
| |  |  | | --- | --- | | 46 | 16 | | 46>16 -> трябва да ги разменим |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 16 | 46 | 56 | 60 | 81 | | Масивът е подреден! |

**Имплементация (сортиране във възходящ ред)**

Реализацията може да стане с влагане на два цикъла for.

Това е най-простата реализация, но и най-бавната!

using System;

namespace CustomSortingMethods

{

public class CustomBubleSort

{

// Отпечатване на масив

public static void printArray(int[] array)

{

for (int index = 0; index < array.Length; index++)

{

Console.WriteLine(array[index]);

}

Console.WriteLine();

}

// Сортиране на масив чрез размяна (метода на мехурчето)

public static void bubleSort(int[] array)

{

int swap, len = array.Length; // дължина на масива (брой елементи)

// n-1 пъти обхождане на масива чрез външния цикъл

for (int k = 0; k < len-1; k++)

{

// едно обхождане на масива без последните k елемента

// (всеки път с 1 по-малко)

for (int i = 0; i < len-k-1; i++)

{

// ако предишният елемент е по-голям от следващия

if (array[i] > array[i + 1])

{

// размяна на двата съседни елемента

swap = array[i];

array[i] = array[i + 1];

array[i + 1] = swap;

}

}

}

}

static void Main(string[] args)

{

int[] arr = { 2, 53, 288, 1, 18, -2}; // декларира масив от числа

// сортиране на масива по метода на мехурчето ВАРИАНТ 1

bubleSort(arr);

printArray(arr); // разпечатване на масива

}

}

}

**Подобрен метод на мехурчето**

Идеята на подобрения метод е в една променлива да се съхранява стойност, която показва последно на кое място (т.е. индекс) при поредното обхождане на масива е направена размяна на елементи. След края на обхождането тази стойност ни показва, че след елемента с този индекс масивът е подреден.   
И още един плюс - ако тази стойност е 0, тогава не е имало нито една размяна, и тогава масивът е подреден и няма нужда да се обхода повече.  
**Ето една реализация** : в **k** се съхранява последната размяна на елементи при поредното обхождане на масива.

using System;

namespace CustomSortingMethods

{

public class CustomBubleSort

{

// Отпечатване на масив

public static void printArray(int[] array)

{

for (int index = 0; index < array.Length; index++)

{

Console.WriteLine(array[index]);

}

Console.WriteLine();

}

// Сортиране на масив чрез размяна (подобрен метод на мехурчето)

public static void bestBubleSort(int[] array)

{

int k; // индексът на последния разменен елемент

// допълнителна променлива използвана при размяна на

// стойностите на елементите

int swap;

int len = array.Length; // дължина на масива (брой елементи)

int right = len - 1; // показва до кой индекс да обхожда масива

while (right>0)

{

k = 0; // индексът на последно разменения елемент

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

{

if (array[i] > array[i + 1])

{

// размяна на двата съседни елемента

swap = array[i];

array[i] = array[i + 1];

array[i + 1] = swap;

k = i;

}

}

// следващото обхождане ще бъде

// до последно разменения елемент

right = k;

}

}

static void Main(string[] args)

{

int[] arr = { 2, 53, 288, 1, 18, -2}; // декларира масив от числа

// сортиране на масива по метода на мехурчето ВАРИАНТ 2

bestBubleSort(arr);

printArray(arr); // разпечатване на масива

}

}

}

**4.6** **Сортиране чрез вмъкване (Insertion Sort).**

**Алгоритъм (вмъкване на елементите един по един в подреден масив)**

1. Масивът/списъкът с елементи, които трябва да бъдат сортирани се разделя на две части: част със сортираните елементи и част с несортираните.
2. При всяка стъпка се взема първият елемент от несортирания списък и **се вмъква** на правилната позиция в сортираната част от списъка
3. Сортирането продължава докато елементите от несортираната част на списъка се изчерпят

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 46 | 10 | 56 | 50 | 16 | | Взема се първият елемент от несортирания списък (46) и се поставя на правилното място в сортирания.  Всъщност в началото подреденият списък съдържа само първия елемент на масива, а останалите елементи формират неподредената част. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 46 | 10 | 56 | 50 | 16 | | Взема се първият елемент от несортирания списък (10) и се поставя на правилното място в сортирания. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 10 | 46 | 56 | 50 | 16 | | Взема се първият елемент от несортирания списък (56) и се поставя на правилното място в сортирания. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 10 | 46 | 56 | 50 | 16 | | Взема се първият елемент от несортирания списък (50) и се поставя на правилното място в сортирания. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 10 | 46 | 50 | 56 | 16 | | Взема се първият елемент от несортирания списък (16) и се поставя на правилното място в сортирания. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 10 | 16 | 46 | 50 | 56 | | Списъкът е сортиран |

**Реализация (имплементация)**

using System;

namespace CustomSortingMethods

{

public class CustomInsertionSort

{

// Отпечатване на масив

public static void printArray(int[] array)

{

for (int index = 0; index < array.Length; index++)

{

Console.WriteLine(array[index]);

}

Console.WriteLine();

}

// Сортиране на масив чрез вмъкване ВАРИАНТ 1

// разделяне на масива на подредена и неподредена част

public static void InsertionSort(int[] array)

{

// в началото подредената част е само от първия елемент

// обхождаме масива от втория елемент (първия в неподредената

// част) нататък, т.е. неподредената част

// при всяка итерация на цикъла подредената част се увеличава

// с един елемент, а неподредената – намалява с един елемент

for (int i = 1; i < array.Length; i++)

{

int value = array[i]; // първият елемент от неподредената част

int index = i; // индексът или мястото, където трябва да се вмъкне

// този елемент. В началото е равен на i

// на този първи елемент му намираме мястото в подредената

// част, тръгвайки отзад напред (от последния индекс към

// нулевия) докато срещнем елемент, който е по-малък от него.

// Тогава сме му намерили мястото.

while (index > 0 && array[index - 1] >= value)

{

// преместваме по-големите от value (първия елемент)

// елементи с един надясно!

array[index] = array[index - 1];

index--;

}

array[index] = value; // поставяне на елемента на мястото му

}

}

static void Main(string[] args)

{

int[] arr = { 2, 5, -4, 11, 0, 18, 22, 67, 51, 6 }; // декларира масив от числа

InsertionSort (arr); // сортиране на масива чрез вмъкване ВАРИАНТ 1

printArray (arr); // разпечатване на масива

}

}

}

**Друга реализация:**

// добавяне на елемент в подреден списък

public static void InsertElement(int value, int[] arr, int n)

{

// index - индексът на първия елемент от неподредената част

// който трябва да се добави в подредената част

// първоначално е индексът на елемента, който добавяме

int i, index = n;

// намиране на мястото на елемента в подредения списък

for(i = 0; i < n; i++)

{

if (value<arr[i])

{

index = i; // намерили сме мястото му

break;

}

}

// преместване на елементите по-големи от вмъквания с

// една позиция надясно

for (i = n; i>index ; i--)

{

arr[i] = arr[i - 1];

}

// поставяне на вмъквания елемент на мястото му

arr[index] = value;

}

// друга реализация на метода "сортиране чрез вмъкване"

public static void InsertionSort2(int[] arr)

{

for (int i = 1; i < arr.Length; i++)

{

InsertElement(arr[i], arr, i);

}

}

**4.7** **Бързо сортиране (Quick Sort)** – О\*(N log N))

**Алгоритъм**

Бързото сортиране е добре известен сортиращ алгоритъм, разработен през 1960 година. Основната част на алгоритъма се състои в сравняващо сортиране:  
(1) Избира се “главен” елемент от списъка с елементи, които ще бъдат сортирани (може да бъде първият, последният или средният).  
(2) Списъкът се пренарежда така, че всички елементи, които са по-малки от “главния” се поставят вляво от него, а всички, които са по-големи – вдясно от него.  
(3) Рекурсивно се повтарят горните стъпки върху списъка с по-малките и списъка с по-големите елементи.  
(4) Получените списъци се сливат и се получава сортираният списък.

**Пример 1 (Устойчив метод!)** [5 3 2 8 7 6 1 9 4]  
За “главен” елемент избираме елемента 5 (първият).

Разделяме списъка на три части: елементи, по-малки от “главния”, “главния”, елементи, по-големи от главния.  
[3 2 1 4] | [5] | [8 7 6 9]  
Рекурсивно извършваме действията върху новополучените списъци.  
[2 1] | [3] | [4] | [5] | [7 6] | [8] | [9]  
[1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9]

Свързваме отделните списъци и получаваме сортирания списък.  
[1 2 3 4 5 6 7 8 9]

**Пример 2 (класически, най-често използван, неустойчив метод!)**  
[5 3 2 8 7 6 1 9 4]  
За “главен” елемент избираме елемента 7 (средният).

Разделяме списъка на три части: елементи, по-малки от “главния”, “главния”, елементи, по-големи от главния.  
[5 3 2 8 **7** 6 1 9 4]

Намира се първият елемент **ПРЕДИ** главния (7), който е по-голям от него и първият елемент (отзад-напред) **СЛЕД** главния, който е по-малък от него и им се сменят местата. Ако няма елемент по-малък от главния елементите по-големи от него се слагат непосредствено преди него.

Ако няма елемент по-голям от главния елементите по-малки от него се слагат непосредствено след него.

[5 3 2 4 **7** 6 1 9 8]

[5 3 2 4 1 **7** 6 9 8]

[5 3 2 4 1 6] **7** [9 8]

**След това същото се прави и за** [5 3 2 6 1 4] и за [9 8]

**Реализация (имплементация)**

// сортиране на масив(част от масив) чрез бързо сортиране

public static void QuickSort(int[] arr, int left, int right)

{

int i = left; // индексът на най-левия елемент в частта, която се сортира

int j = right; // индексът на най-десния елемент в частта, която се сортира

// избор на главен елемент - средният в масива [(0+9)/5 = 4]

double pivotValue = ((left + right) / 2); // индексът на главния елемент

// главен елемент mainelem (в случая arr[4]=20)

int mainelem = arr[Convert.ToInt32(pivotValue)];

int swap = 0; // временна променлива за размяна на елементи

while (i <= j)

{

while (arr[i] < mainelem)

{

i++; // пропуска всички елементи по-малки от главния

}

while (mainelem < arr[j])

{

j--; // пропуска всички елементи по-големи от главния

}

if (i <= j)

{

// в този момент i е индексът на първия елемент по-голям от главния

// а j е индексът на първия елемент по-малък от главния

// разменят им се местата

swap = arr[i];

arr[i++] = arr[j];

arr[j--] = swap;

}

} // end of while (i <= j)

if (left < j)

{

QuickSort(arr, left, j); // сортиране на лявата част

}

if (i < right) {

QuickSort(arr, i, right); // сортиране на дясната част

}

}

static void Main(string[] args)

{

int[] arr = { 2, 5, -4, 11, 0, 18, 22, 67, 51, 6 }; // декларира масив от числа

// сортиране на масива по метода на бързото сортиране

QuickSort(arr, 0, arr.Length-1);

printArray (arr); // разпечатване на масива

}

**V. Обяснява понятието рекурсия и разработва програми с използването на рекурсивни алгоритми**

**1. Какво е итерация?**

Итерацията е блок от инструкции, който се повтаря отново и отново, докато дадено условие е истина. Итерацията може да бъде постигната с помощта на цикли. Всяко завъртане на цикъла (всяко изпълнение на тялото на цикъла) се нарича итерация.

**2. Какво е рекурсия?**

При рекурсията отново имаме повтаряне на изпълнението на даден код, но оформен като функция с параметри. Тук не се използват цикли.

С две думи, рекурсията е изпълнението на функция от тялото на същата функция. Такъв вид рекурсия се нарича **пряка рекурсия**.

Има и **косвена рекурсия**. При нея функцията А извиква функцията В, която от своя страна отново извиква функцията А.

**При рекурсията е важното да има условие, при което да се излезе от рекурсията!**

Има задачи, които много трудно се решават с итеративния метод, т.е. чрез цикъл, но се решават лесно с рекурсия. Например задачата за Ханойските кули.

**Задача 1:** Да се напише функция на C# за изчисляване на **n факториел**

**n! = 1\*2\*3\*4\* … \*n**

**С итерация (с цикъл):**

static long IFactorial(int num)

{

long nfact = 1;

for (int i = 1; i <= num; i++) { nfact \*= i; }

return nfact;

}

**С рекурсия (с пряка рекурсия):**

static long RFactorial(int num)

{

if (num == 0) return 1;

return num \* RFactorial(num - 1);

}

Рекурсивните методи имат 3 части:

* **Предварително действие** (преди извикване на рекурсията)
* **Рекурсивни извиквания** (стъпка навътре)
* **Последващо действие** (след връщане от рекурсията)

Тъй като при рекурсията има многократно изпълнение на една функция с различни параметри, а те се пазят в програмния стек, е възможно да има проблеми при изпълнението – запълване на стека (Stack Overflow)!

**Затова, ако можете да решите дадена задача с итерация, използвайте нея!**

**Примерно при изчисление на редицата на Фибоначи, от един момент нататък системата може да увисне поради многократното изчисление на един и същи код**

**Задача 2 : Редица на Фибоначи:** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

**F0 = 0**

**F1 = 1**

**Fn = Fn-1 + Fn-**2 (т.е. всяко число е равно на сумата на предишните две)

**// Рекурсивно изчисление на n-ия член на редицата на Фибоначи**

static decimal Fibonacci(int n)

{

if (n == 0) return 0;

if (n == 1) return 1;

else return Fibonacci(n - 1) + Fibonacci(n - 2);

}

static void Main()

{

Console.WriteLine(Fibonacci(10)); // 55

Console.WriteLine(Fibonacci(50)); // Тук увисва

}

**// Итеративно изчисление на n-ия член на редицата на Фибоначи**

static decimal IFibonacci(int n)

{

if (n == 0) return 0;

if (n == 1) return 1;

int f=0;

int f1 = 0, f2 = 1; // тук се съхраняват последните две стойности

for (int i = 1; i < n; i++)

{

f = f1 + f2; // новата стойност = сумата на предишните две

f1 = f2;

f2 = f;

}

return f;

}

static void Main()

{

Console.WriteLine(Fibonacci(10)); // 55

Console.WriteLine(IFibonacci(40)); // 102334155

Console.WriteLine(Fibonacci(50)); // -298632853 (надхвърля decimal)

}

**Задача 3 : Рекурсивно обръщане на масив**

// обръщане на масив – рекурсивно

static void ReverseArrayRec(int[] arr, int start, int end)

{

// краищата на масива са се разминали

if (end <= start)

{

return;

}

// разменяме крайните елементи

int swap = arr[end];

arr[end] = arr[start];

arr[start] = swap;

// обръщаме подмасива

ReverseArrayRec(arr, start + 1, end-1);

}

// ReverseArrayRec(arr, 0, arr.Length-1); // Извикване

**Итеративно обръщане на масив**

static void ReverseArray(int[] arr)

{

int swap=0, last = arr.Length;

for (int i = 0; i <= last / 2; i++)

{

// разменяме 1-вия и последния, 2-рия и предпоследния и т.н.

swap = arr[i];

arr[i] = arr[last-i-1];

arr[last-i-1] = swap;

}

}

// ReverseArray(arr); // Извикване

**Задача 4: Задача за "Ханойските кули"**

Да се състави програма реализираща играта "Ханойски кули". Играта е следната: Имате три стълба A, B, C и N диска с различни диаметри с отвор в центъра си, така че да могат да се надяват на стълбовете.

Дисковете са наредени на стълб A в следния ред: най-отдолу е диска с най-голям диаметър, върху него е този с по-малък диаметър и т. н.

Целта на играта е дисковете да се наредят в същия ред на стълб B, като се спазват следните правила:

* при едно местене може да се взема само един диск от тези най-отгоре на някой от стълбовете
* и не може да се поставя диск с по-голям диаметър върху диск с по-малък.

    Входни данни: N > 0 - цяло - брой дискове.  
    Изходни данни: поредица от команди към играча, кой диск да премести и къде.

**Алгоритъм**

    Ако броя дискове е 1  
        <1> Тогава го преместваме от стълб A на стълб B.  
    Иначе допускаме, че можем да решим задачата за N-1 диска.  
        <2> Тогава преместваме горните N-1 диска от стълб A на стълб C.  
        <3> Преместваме останалия N-ти диск от A на B.  
        <4> Преместваме N-1 диска от C на A.

**Забележка.** Именно допускането, че можем да решим задачата за N-1 диска означава, че имаме рекурсия.

static void HanoyTower(char A, char B, char C, int n) {

if (n == 1) { // Изход от рекурсията <1>

Console.WriteLine("Премести диск 1 от стълб {0} на стълб {1}", A, B);

} else {

HanoyTower(A, C, B, n - 1); // <2>

// <3>

Console.WriteLine("Премести диск {0} от стълб {1} на стълб {2}", n, A, B);

HanoyTower(C, B, A, n - 1); // <4>

}

}

// Извикване

char a= 'A', b= 'B', c= 'C';

HanoyTower(a, b, c, 3);

**Резултатът е:**

Премести диск 1 от стълб A на стълб B

Премести диск 2 от стълб A на стълб C

Премести диск 1 от стълб B на стълб C

Премести диск 3 от стълб A на стълб B

Премести диск 1 от стълб C на стълб A

Премести диск 2 от стълб C на стълб B

Премести диск 1 от стълб A на стълб B

**Още материали можете да намерите на следната страница:**

[**https://introprogramming.info/intro-csharp-book/read-online/glava10-rekursia/**](https://introprogramming.info/intro-csharp-book/read-online/glava10-rekursia/)

**VI. Анализира фрагмент/и от код и идентифицира и поправя правилно грешките в написания програмен код, така че да реши поставената задача. Допълва кода, ако и когато това е необходимо.**

**Открийте и поправете грешките в следния код:**

Using System;

namespace CustomSortingMethods

{

Public Class CustomInsertionSort

{

// Отпечатване на масив

Public Static Void printArray(int[] Array)

{

for (int index = 0; index < array.Length; index+)

{

Console.WriteLine(array[index]);

}

Console.WriteLine();

}

// Сортиране на масив чрез вмъкване

Public Static Void InsertionSort(int[] array)

{

for (int i = 1; i < array.Len; i+)

int value = array[i];

int index = i;

while (index > 0 && array[index - 1] >= value)

{

array[index] = array[index - 1];

index-;

}

array[index] = value;

}

}

static void Main(string[] args)

{

int[] arr = ( 2, 5, -4, 11, 0, 18, 22, 67, 51, 6 ); // декларира масив от числа

InsertionSort (arr); // сортиране на масива чрез вмъкване

PrintArray(arr); // разпечатване на масива

}

}

}