*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.

Един обект наричаме **рекурсивен**, ако съдържа себе си или е дефиниран чрез себе си.

**Рекурсия** е програмна техника, при която даден **метод извиква сам себе си** при ре­ша­ването на определен проблем. Такива методи наричаме **рекурсивни**.

Рекурсията е програмна техника, чиято правилна употреба води до еле­гант­ни решения на определени проблеми. Понякога нейното използване може да опрости значително кода и да подобри четимостта му.

**Пряка и косвена рекурсия**

Когато в тялото на метод се извършва извикване на същия метод, казваме, че методът е **пряко рекурсивен**.

Ако метод A се обръща към метод B, B към C, а С отново към А, казваме, че методът А, както и методите В и C са **непряко (косвено) рекур­сивни**или **взаимно-рекурсивни**.

Веригата от извиквания при косвената рекурсия може да съдържа множество методи, както и разклонения, т.е. при наличие на едно условие да се извиква един метод, а при различно условие да се извиква друг.

**Дъно на рекурсията**

Реализирайки рекурсия, трябва да сме сигурни, че след краен брой стъпки ще получим конкретен резултат. Затова трябва да имаме един или няколко случаи, чието решение можем да намерим директно, без рекур­сивно извикване. Тези случаи наричаме **дъно на рекурсията**.

В примера с числата на Фибоначи, дъното на рекурсията е случаят, когато **n** e по-малко или равно на 2. При него можем директно да върнем резул­тат, без да извършваме рекурсивни извиквания, тъй като по дефиниция първите два члена на редицата на Фибоначи са равни на 1. Ако даден рекурсивен метод няма дъно на рекурсията, тя ще стане **безкрайна** и резултатът ще е **StackOverflowException**.

**Създаване на рекурсивни методи**

Когато създаваме рекурсивни методи, трябва разбием задачата, която решаваме, на подзадачи, за чието решение можем да използваме същия алгоритъм (рекурсивно).

Комбинирането на решенията на всички подзадачи, трябва да води до решение на изходната задача.

При всяко рекурсивно извикване, проблемната област трябва да се огра­ничава така, че в даден момент да бъде достигнато дъното на рекурсията, т.е. разбиването на всяка от подзадачите трябва да води рано или късно до дъното на рекурсията.

**Реализация на алгоритъма**

Дъното на нашата рекурсия е най-простият случай n = 0, при който стойността на факториел е 1. В останалите случаи, решаваме задачата за n-1 и умножаваме получения резултат по n. Така след краен брой стъпки, със сигурност ще достигнем дъното на рекурсията, понеже между 0 и n има краен брой естествени числа.

След като имаме налице тези ключови условия, можем да реализираме метод изчисляващ факториел:

**Алгоритми за търсене**

**Линейното** търсене и **двоичното** търсене са двата метода, които се използват в масиви за **търсене**елементите.

**Търсенето** е процес на намиране на елемент в списъка с елементи, съхранени в произволен ред или на случаен принцип.

**Основната разлика** между линейното търсене и двоичното търсене е, че бинарното търсене отнема по-малко време за търсене на елемент от списъка на подредените елементи. Така че се заключава, че ефективността на метода на двоично търсене е по-голяма от линейното търсене.

**Друга разлика** между двете е, че има предпоставка за двоичното търсене, т.е. елементите трябва да бъдат **сортиран** докато при линейното търсене няма такава предпоставка. Въпреки че и двата метода на търсене използват различни техники.

**Определение за линейно** **търсене -** При линейно търсене всеки елемент от масива се извлича един по един в логически ред и се проверява дали е желаният елемент или не. Търсенето ще бъде неуспешно, ако всички елементи са достъпни и желаният елемент не бъде намерен. В най-лошия случай, броят на средния случай, може да се наложи да сканираме половината от размера на масива (n / 2).

Следователно линейното търсене може да бъде определено като техника, която последователно обикаля масива, за да намери дадения елемент.

**Определение на двоично търсене -** Двоичното търсене е изключително ефективен алгоритъм. Тази техника на търсене отнема по-малко време при търсене на дадения артикул при минимални възможни сравнения. За да извършим двоичното търсене, първо трябва да сортираме елементите от масива.

Логиката зад тази техника е дадена по-долу:

* Първо намерете средния елемент от масива.
* Средният елемент на масива се сравнява с елемента за търсене.

Може да възникнат три случая:

1. Ако елементът е необходимия елемент, тогава търсенето е успешно.
2. Когато елементът е по-малък от желания елемент, тогава търсете само първата половина на масива.
3. Ако той е по-голям от желания елемент, тогава търсете във втората половина на масива.

Повторете същите стъпки, докато елемент не бъде намерен или изчерпан в областта на търсене. В този алгоритъм всеки път, когато търсенето се намалява. Следователно броят на сравненията е най-много log (N + 1). В резултат на това е ефективен алгоритъм в сравнение с линейното търсене, но масивът трябва да бъде сортиран, преди да извършите двоичното търсене.

**Двоично търсене**(Binary search) При несортиран масив няма как да търсим със сложност по-малка от О(n), в случай, че ще извършваме много търсения върху един масив то за да ускорим търсенето можем първоначално да го сортираме ( за О(nlog(n)) ) и после да търсим със по-ниска сложност O(log(n))

**Алгоритми за сортиране**

Дефиниция на **алгоритъм за сортиране -** Алгоритъм, който подрежда елементите на списък, В ненамаляващ ред, Елементите трябва да са сравними. Или **по-формално** обяснение: Входът е последователност / списък от елементи; Изходът е пренареждане / пермутация на елементи; В ненамаляващ ред

**Видове алгоритми за сортиране**

Видове: Известни са стотици методи (алгоритми) за сортиране които подреждат елементите на място (без да се използва помощен масив, което е доста неефективно от гледна точка на заеманата памет), но всеки един от тях може да се отнесе към една от три основни групи, а именно методи за сортиране чрез:

1. Сортиране чрез размяна:

• метод на мехурчетата (пряка размяна);

• метод на шейкър сортиране (“чрез клатене”);

• метод на бърза сортировка на Hoare.

1. Сортиране чрез селекция (избор):

• метод на пряка селекция;

• метод на пирамидална сортировка.

1. Сортиране чрез вмъкване:

• метод на пряко вмъкване;

• метод на двуично вмъкване;

• метод на вмъкване с намаляваща стъпка.

Метод на СОРТИТАНЕ с ПРЯКА СЕЛЕКЦИЯ

ИДЕЯ на метода: Масивът се обхожда n-1 пъти, като на стъпката (обхождането) с номер i се избира най-големият измежду първите n-i+1 елемента и той се поставя на n-i+1 -во място в масива.

Би могло да се избира най-малкият елемент и съответно – да се поставя на такова място в началото на масива, че масивът да се сортира възходящо.

// Алгоритъмът за сортиране чрез пряка селекция (на [английски](https://bg.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8_%D0%B5%D0%B7%D0%B8%D0%BA): *Selection sort*) е неефективен с изчислителна сложност O*n*2. Подобен алгоритъм, който има по-добра производителност, е алгоритъмът за [сортиране чрез вмъкване](https://bg.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%B0%D0%BD%D0%B5_%D1%87%D1%80%D0%B5%D0%B7_%D0%B2%D0%BC%D1%8A%D0%BA%D0%B2%D0%B0%D0%BD%D0%B5). Сортирането чрез пряка селекция впечатлява с простотата си, а също така в дадени ситуации има предимства пред някои сложни алгоритми.

Алгоритъмът намира най-малкия елемент в списъка и го разменя с първия елемент. След това търси втория най-малък и го поставя на второ място. Действието се повтаря, докато се стигне до края на списъка. При този алгоритъм не се правят повече от *n* размени. Този алгоритъм е поле-зен в случаи, когато размяната на елементи е тежка операция и трябва да се минимизира.//

Сортиране с метод на мехурчето

Основна идея: Започвайки последователно от началният елемент до крайният сравняваме всеки елемент със следващия като ги разменяме ако не са подредени. По този начин на всяка стъпка изкарваме най-големият в края на правилното за него място. Повтаряйки този процедура толкова пъти колкото е големината на масива постигаме правилно подреждане на целият масив.

// Методът на мехурчето (на [английски](https://bg.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8_%D0%B5%D0%B7%D0%B8%D0%BA): Bubble sort) е прост сортиращ алгоритъм. Алгоритъмът започва в началото на сортиращия се списък. Той сравнява първия и втория елемент, и ако първият е по-голям от втория, ги разменя. Продължава да прави с това с всички съседни двойки елементи до края на сортиращия се списък. След това повтаря същото действие още толкова пъти, докато накрая при обхождането на целия списък не е направено нито една размяна на два съседни елемента. Средният и най-лош случай на този алгоритъм е O(n2) и не се използва за сортиране на големи неподредени множества от данни. Методът на мехурчето може да се използва за малки множества или за множества, където има елементи близо до очакваното си място. Например, ако някой елемент не е на мястото си, на разстояние един елемент (например 112 и 111), методът на мехурчето ще обходи един път множеството и ще направи размяната, а на второто обхождане няма да направи размяна и ще завърши сортирането, и това ще отнеме само 2n.//

Сортиране с пряка селекция

Намаля броя на разменянията в сравнение с метода на мехурчето! Основна идея: Търсим най-големият елемент в масива и директно го поставяме на последно място. След това следващият по големина и отново го поставяме на място и т.н. Докато всички се подредят.

Сортиране с вмъкване

Основна идея: сортиране постепенно на все по-голяма част от масива, като обхождайки несортираната част всеки един елемент го поставяме в сортираната част с намиране на правилното за него място на което сортираният масив остава сортиран.

// Сортирането чрез вмъкване (на [английски](https://bg.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8_%D0%B5%D0%B7%D0%B8%D0%BA): Insertion sort) е прост алгоритъм, който е относително ефективен при малки и почти сортирани списъци, като често се използва в комбинация с по-усъвършенствани алгоритми. Алгоритъмът работи като взема всеки елемент един по-един от списъка и го вмъква на съответното си място в нов сортиран списък. При сортиране на масиви елементите могат да споделят базовата памет (пространство), но се извършват прекалено много премествания на елементи което е скъпа операция. Шел сортирането е по-усъвършенстваният вариант за по-големи списъци.//

[Шел сортиране](https://bg.wikipedia.org/wiki/%D0%A8%D0%B5%D0%BB_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%B0%D0%BD%D0%B5)

Шел сортирането е открито от Доналд Шел през 1959 година. То подобрява метода на мехурчето и сортирането чрез вмъкване чрез преместването на елементите през повече от една позиция в даден момент. Може да бъде описано като нареждане на последователни елементи в двумерен масив и след това сортирането на колоните на масива чрез сортиране чрез вмъкване.

Сложност на сортиращите алгоритми Бавни сортировки: Bubble, Selection, Insertion - O(n^2) Бързи сортировки: Merge, Quick - О(n\*log(n))

Сортиране чрез сливане(merge sort)

Основна идея: Ако имаме два сортирани масива то със линейна сложност може да ги влеем в един масив. Тогава ако разделим масива който искаме да сортираме на по-малки масиви и на всяка стъпка сливаме два по-малки масива в един голям то за log(N) стъпки ще слеем всички масиви до един масив, като всяка от стъпките е била линейна.

// Сортирането чрез сливане (на [английски](https://bg.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8_%D0%B5%D0%B7%D0%B8%D0%BA): Merge sort) използва принципа на сливане на вече сортирани списъци. Започва с обединяването на всяка двойка елементи (например 1 и 2, после 3 и 4 ...) и ги разменя ако първата двойка трябва да е след втората. След това слива всички списъци от два елемента в списъци от четири, отново ги пренарежда и ги слива в списъци от осем и т.н., докато останат само два списъка, които се сливат последно за да приключи сортирането. Този алгоритъм се представя добре при списъци с големи мащаби и при най-лош случай има ефективност O(n log n). Също така се прилага лесно както за масиви, така и за списъци, стига да предоставя последователен достъп към елементите. Минус е обаче ефективността на памет O(n), както и прекалено многото използване на прости операции като присвояване и копиране.//

Бързо сортиране ( quick sort)

Основна идея: Ако вземем едно произволно число от масива, то с линейна сложност можем да прехвърлим всички по-малки числа от масива да са в ляво на числото, а всички по-големи в дясно. При бързото сортиране избираме число от масива прехвърляме по-малите отляво, по-големите от дясно и след това изпълняваме същата процедура за лявата и дясната половина. По тази процедура получаваме сложност в средният случай О(n\*log(n))

// Алгоритъмът за бързо сортиране (на [английски](https://bg.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8_%D0%B5%D0%B7%D0%B8%D0%BA): Quicksort) е един от най-добрите алгоритми за търсене на голям брой елементи. Нарича се още алгоритъмът на Хор, заради името на човека, който го е измислил – [Тони Хор](https://bg.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BD%D0%B8_%D0%A5%D0%BE%D1%80). Този алгоритъм прави O(n log n) сравнения за да сортира n на брой елемента. В практиката алгоритъмът за бързо сортиране е по-бърз от другите O(n log n) алгоритми. Неговата имплементация се извършва с рекурсия.//

## Абстрактни структури от данни

## Какво е структура 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.

### **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.

### **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

В настоящата тема ще разгледаме линейните (списъчни) структури от данни, а в следващите няколко теми ще обърнем внимание и на по-сложните структури като дървета, графи, хеш-таблици и множества и ще обясним кога се използва и прилага всяка от тези структури.

Овладяването на основните от структури от данни в програмирането е от изключителна важност, тъй като без тях не може да се програмира ефективно. Те, заедно с алгоритмите, са в основата на програмирането и в следващите няколко глави ще се запознаем с тях.

## Списъчни структури

Най–често срещаните и използвани са линейните (списъчни) структури. Те представляват абстракция на всякакви видове редици, последовател­ности, поредици и други подобни от реалния свят.

**Списък**

Най–просто можем да си представим списъка като наредена последователност (редица) от елементи. Нека вземем за пример списък за покупки от магазин. В списъка можем да четем всеки един от елементите (покупките), както и да добавяме нови покупки в него. Можем също така и да задраскваме (изтрием) покупки или да ги разместваме.

**Абстрактна структура данни "списък"**

Дефиниция на структурата списък: **Списък** е линейна структура от данни, която съдържа поредица от елементи. Списъкът има свойството дължина (брой елементи) и елемен­тите му са наредени последователно.

Списъкът позволява добавяне на елементи на всяко едно място, премахването им и последователното им обхождане. Както споменахме по-горе, един АТД може да има няколко реализации. Пример за такъв АТД е интерфейсът **System.Collections.IList**.

Интерфейсите в C# изграждат една "рамка" за техните имплементации – класовете. Тази рамка представлява съвкупност от методи и свойства, които всеки клас, имплементиращ интерфейса, трябва да реализира.

Всеки АТД реално определя някакъв интерфейс. Нека разгледаме интер­фейса **System.Collections.IList**. Основните методи, които той декларира, са:

-     **int 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

-     **void Remove(object) –**премахва съответния елемент

-     **void RemoveAt(int) –**премахва елемента на дадена позиция

-     **int IndexOf(object) –**връща позицията на елемента

-     **this[int] –**индексатор, позволява достъп на елементите по подадена позиция

Нека видим няколко от основните реализации на АТД списък и обясним в какви ситуации се използва всяка от тях.

**Статичен списък (реализация чрез 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 (по подобие на класа **StringBuilder**). Такъв списък се нарича **статичен.**

Първо си създаваме масива, в който ще пазим елементите, както и брояч за това колко елемента имаме в момента. След това добавяме и конструктора, като инициализираме нашия масив с някакъв начален капацитет, за да не се налага да го преоразмеряваме, когато добавим нов елемент. Реализирахме операцията добавяне на нов елемент, както и вмъкване на нов елемент. Тъй като едната операция е частен случай на другата, методът за добавяне вика този за вмъкване. Ако масивът ни се напълни заделяме два пъти повече място и копираме елементите от стария в новия масив.

***Свързан списък (динамична реализация)***

Както видяхме, 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 елементите съдържат само конкретния обект, при динамичния списък елементите пазят 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)

За динамичната реализация ще са ни необходими два класа – класът **Node** – който ще представлява един отделен елемент от списъка и главният клас **DynamicList**:

***Класът ArrayList***

След като се запознахме с някои от основните реализации на списъците, ще се спрем на класовете в C#, които ни предоставят списъчни структури "на готово". Първият от тях е класът **ArrayList**, който представлява динамично-разширяем масив.

**ArrayList** дава възможност да добавяме, премахваме и търсим елементи в него. Някои по-важни членове, които можем да използваме са:

-     **Add(object)** – добавяне на нов елемент

-     **Insert(int, object)** – добавяне на елемент на определено място (индекс)

-     **Count** – връща броя на елементите в списъка

-     **Remove(object)** – премахване на определен елемент

-     **RemoveAt(int)** – премахване на елемента на определено място (индекс)

-     **Clear()** – изтрива елементите на списъка

-     **this[int]** – индексатор, позволява достъп на елементите по подадена позиция

Както видяхме, един от основните проблеми при тази реализация е прео­размеряването на вътрешния масив при добавянето и премахването на елементи. В класа **ArrayList** проблемът е решен чрез предварително създаване на по-голям масив, който ни предоставя възможност да доба­вяме елементи, без да преоразмеряваме масива при всяко добавяне или премахване на елементи.

## Пример на C# за сортиране на числата, чрез алгоритъма на мехурчето

using System;

class BubbleSort

{

static void Main()

{

int[] array = new int[] { 6, 9, 4, 3, 5, 1, 42, -2 };

for (int i = 0; i < array.Length - 1; i++)

{

for (int j = 0; j < array.Length - 1; j++)

{

if (array[j] > array[j + 1]) // swap the elements

{

int tmp = array[j];

array[j] = array[j + 1];

array[j + 1] = tmp;

}

}

}

for (int i = 0; i < array.Length; i++) // print the elements

{

Console.Write(array[i] + " ");

}

}

}