1. **Стандартни типове данни в С#. Масиви в С#**

**Типове данни:**

**-целочислен тип**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тип** | **Размер (битове)** | **Signed/Unsigned** | **Диапазон** | **Default value** |
| byte | 8 | Unsigned | 0 до 255 | 0 |
| sbyte | 8 | Signed | -128 до 127 | 0 |
| short | 16 | Signed | -32768 до 32767 | 0 |
| ushort | 16 | Unsigned | 0 до 65535 | 0 |
| int | 32 | Signed | -231 до 231 - 1 | 0 |
| uint | 32 | Unsigned | 0 до 232 | 0 |
| long | 64 | Signed | -263 до 263-1 | 0L |
| ulong | 64 | Unsigned | 0 до 264 | 0 |

При signed типовете един бит се отделя за знака (+/-) =>

-при int (signed) -> 231

-при uint (unsigned) -> 232

При signed типовете има -1 на максимума заради нулата (не е нито положителна, нито отрицателна).

**-тип с плаваща запетая**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Размер (битове)** | **Точност след запетаята** | **Default value** |
| float | 32 | 6 до 9 знака | 0.0F |
| double | 64 | 15 до 16 знака | 0.0D |
| decimal | 128 | 28 до 29 знака | 0.0M |

**-символи**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Размер (битове)** | **Диапазон** | **Default value** |
| char | 16 | U +0000 to U +ffff | '\0' |

**-булеви типове**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Размер (битове)** | **Диапазон** | **Default value** |
| bool | 8 | true/false | false |

**-низове**

|  |  |  |
| --- | --- | --- |
| **Тип** | **Размер (битове)** | **Default value** |
| string | Зависи от какво съдържа в себе си | null |

**-обекти**

|  |  |  |
| --- | --- | --- |
| **Тип** | **Размер (битове)** | **Default value** |
| object | Зависи от какво съдържа в себе си | null |

**Масиви:**

Колекция, която пази елементи от един и същ тип. Достъпва се чрез индекси. Има фиксиран размер.

**Начини за деклариране:**

int[] arr; - деклариране без инициализация

int[] arr = new int[5]; - деклариране и инициализиране на празен масив с 5 елемента, които приемат стойност по подразбиране

int[] arr = new int[] { 2, 3, 5, 7, 11 }; - деклариране на масив и инициализиране с елементи

int[] arr = { 2, 3, 5, 7, 11 }; - деклариране на масив и инициализиране с елементи (не може да се използва след деклариране на масива)

**Видове масиви:**

* Обикновен масив
* Матрици:

Масив състоящ се от масиви

int[,] matrix = new int[,]

{

{ 1, 2, 3, 4 },

{ 5, 6, 7, 8 }

};

**Използване на масиви:**

|  |  |
| --- | --- |
| **Пример** | **Описание** |
| arr[0] | Взима първи елемент от масива |
| arr[arr.Length - 1] | Взима последен елемент от масива |
| arr.Length | Връща броят на елементите |
| matrix.GetLength(0) | Връща броя на редовете в матрицата |
| matrix.GetLength(1) | Връща броя на колоните в матрицата |

1. **Сортировки**

**Bubble Sort –** проверява кой от двата съседни елемента е по-голям и им разменя местата, докато не се подреди масива. При всяко обхождане най-големият елемент отива в края на масива.

**Selection Sort –**  обхожда всеки елемент, като на всяка итерация намира най-малкият елемент и го поставя на съответното място.

**Insertion Sort –** при тази сортировка приемаме, че първият елемент винаги е на мястото си и започваме от втория. Сравняваме втори с първи и ако трябва им сменяме местата. След това минаваме на трети елемент. Сравняваме го с първи и втори като му намираме съответната позиция. Обхождаме докато не се подреди масива.

**Quick Sort –** избираме елемент от масива, който да служи за ориентир. Ще използваме средният елемент от масива. Дефинираме си променливи за индекс, който следи елементите от лявата страна и едина, която следи за тези от дясната. С цикъл обхождаме докато левият индекс е по-малък от десния. В този цикъл имаме два други цикъла. Единият обхожда докато левият индекс е по-малък или равен от дясната граница на масива и докато е по-малък от ориентирът. Докато това условие е спазено увеличаваме левият индекс. За дясната страна обхождаме докато десният индекс е по-голям или равен от лявата граница и докато е по-голям от ориентирът. Докато това условие е спазено трябва да намалим десният индекс. Ако левият индекс е по-малък или равен от десния то ние разменяме елементите с ляв и десен индекс, за да поставим всички по-малки елементи от ориентирът отляво, а всички по-големи отдясно. Пак увеличаваме и намаляме индексите. След това проверяваме дали левият индекс е по-малък от дясната граница на масива и рекурсивно извикваме метода, като му подаваме левият индекс за лява граница и дясната граница. След това го извикваме още веднъж и проверяваме дали десният индекс е по-голям от лявата граница. Рекурсивно го извикваме и му подаваме лява граница и десен индекс като дясна граница.

**Merge Sort –** тук първо трябва да намерим средата на масива. След като я намерим извикваме същият метод два пъти. Първия път от началото на масива до средата, втория път от средата + 1 до края на масива. Рекурсивно той бива разделян на 2 части, докато левия индекс не надвиши десния. След като останат точно 2 елемента. След това се извиква функция която да ги съедини, като преди това ги записва във временни масиви. Следи за кой е по-голям и така им определя позициите, като следи за позиция на ляв масив, десен масив и позиция в главния масив.

|  |  |
| --- | --- |
| **Алгоритъм за сортиране** | **Сложност** |
| Bubble Sort | O(n2) |
| Selection Sort | O(n2) |
| Insertion Sort | O(n2) |
| Quick Sort | O(n log n) |
| Merge Sort | O(n log n) |

1. **Двоично търсене в сортиран масив**

При това търсене се взима число което трябва да бъде намерено. Взима се средният елемент от масива. Ако числото което търсим е по-голямо от елементът в средата се взима втората половина + 1, за да не се включи предишната среда, ако е по-малко се взима първата – 1, за да се изключи предишната и така процесът продължава докато не се намери числото или докато началото не стане по-голямо от краят.

1. **Свързан списък**

Структура от данни състояща се от node-ове, където всеки node съдържа поле за стойността, която се съдържа в него и референция към следващ node. Когато текущият node е последен то референцията към следващ node е null. Node-ът който е първи се нарича head, а последният се нарича tail.

1. **Хеш таблица**

Структура, която се състои от ключ и стойност. Ключовете трябва да са уникални. Нуждае се от структура, която да пази елементите и структура, която да се състои от ключ и стойност.   
За създаване на тази структура трябва:

Хешираща функция, която да определя на кой индекс се намира двойката ключ стойност.

За колекция ползваме масив от свързани списъци, които се състоят от HashEntry (ключ и стойност). За да премахнем/добавим елемент по ключ трябва да намерим свързаният списък от масива и след това да премахнем/добавим елемента.

1. **Дървета**

Всяко дърво се състои от TreeNode-ове, всеки TreeNode съдържа в себе си стойност и колекция от наследници. Дърветата имат корен от който започва всяко разклонение.

Node-ове, които нямат наследници се наричат листа. Част от дървото се нарича поддърво. Наследници, които произлизат от 1 родител и са на едно ниво се наричат siblings. Връзката между родител и наследник се нарича edge. Node с поне един наследник се нарича Internal Node.

Начини на обхождане:

A

/ \

B C

/ \ \

D E F

Preorder – node, ляво поддърво, дясно поддърво (A B D E C F)

Inorder – ляво поддърво, node, дясно поддърво (D B E A C F) (само за двоични дървета)

Postorder – ляво поддърво, дясно поддърво, node (D E B F C A) (използва се рекурсия)

Видове обхождания:  
DFS (Depth-First Search) – с изброените варианти по-горе. Тръгва надолу по левият наследник и така докато не стигне края. След това се връща нагоре и тръгва в следващия клон. Използва стек.

BFS (Breadth-First Search) – тръгва от корена, след това посещава наследник 1, 2, 3… и след това минава още едно ниво надолу и обхожда всички наследници на това ниво. Използва се опашка. При него няма ордъри.