## 13. Структури от данни и алгоритми. Анализ на алгоритми. Абстрактни типове от данни. Стек, опашка, списък, дърво. Сортиране.

Изложението по въпроса трябва да включва следните по-съществени елементи:

1. Основи на анализ на алгоритми: Асимптотична нотация на сложността. Осннновни рекурентни формули. Примери за анализ на алгоритми.
2. Абстрактни типове от данни. Интерфейс и реализация.
3. Свързани списъци. Обработка на списъци.
4. Структура от данни стек. Реализация.
5. Структура от данни опашка. Реализация.
6. Дървета. Типове дървета.
7. Сортиране. Елементарни методи за сортиране.
8. Сортиране – QuickSort.

### 1. Анализ на алгоритми

#### 1.1. Сложност на алгоритми

„Сложност на алгоритъм“ е мярка за това, колко ресурси използва този алгоритъм. „Сложен“ в този смисъл означава „използва много ресурси“. За какви ресурси става на въпрос?

- Ресурсът, който най-често имаме предвид, е **времето**. Естествено е да искаме алгоритмите ни да работят максимално бързо. В този смисъл, „качествен алгоритъм“ е алгоритъм, който работи бързо върху множеството от всички входове. Но „бързо“ е разговорен термин. Прецизните разсъждения се базират на понятието „сложност по време“ (**Time Complexity**). Алгоритъм с висока сложност по време е бавен алгоритъм. Времето е функция на големината на входа, която се оценява асимптотично. С други думи, искаме да знаем в какъв порядък броя на елементарните операции нараства, спрямо големината на входа.
- Следващият по важност ресурс е **паметта**, която използва алгоритъмът. Естествено е да искаме алгоритмите ни да използват минимална памет. В този смисъл, „качествен алгоритъм“ е алгоритъм, който използва малко памет върху всички входове. Прецизните разсъждения се базират на понятието „сложност по памет“ (**Space Complexity**). Алгоритъм с висока сложност по памет е такъв, който използва много памет.

#### 1.2. Асимптотични нотации за сложността

С $n$ бележим големината на входа.

- Тета. $\Theta\left(g(n)\right) \overset{\text{def.}}{=} \left\{f(n) \mid \exists c_1, c_2 > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\right\}$

  Аналогичен запис: $g(n) \asymp f(n)$.

  $\Theta(1)$ означава анонимна константа и служи за синоним на „константна сложност“. Примери за алгоритми с константни сложности са такива, чиито вход има константна големина, например сумиране на две числа или такива, който винаги връщат едно и също, игнорирайки входа.
- О-голямо, Омега-голямо, о-малко, омега-малко.

$$O\left(g(n)\right) \overset{\text{def.}}{=} \left\{f(n) \mid \exists c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\right\}, \qquad f(n) \preceq g(n)$$

$$\Omega\left(g(n)\right) \overset{\text{def.}}{=} \left\{f(n) \mid \exists c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\right\}, \qquad f(n) \succeq g(n)$$

$$o\left(g(n)\right) \overset{\text{def.}}{=} \left\{f(n) \mid \forall c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)\right\}, \qquad f(n) \prec g(n)$$

$$\omega\left(g(n)\right) \overset{\text{def.}}{=} \left\{f(n) \mid \forall c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n)\right\}, \qquad f(n) \succ g(n)$$

#### 1.3. Основни рекурентни формули

Основните рекурентни формули, за които лесно може да намерим сложност по време са представени в каноничен вид в Мастър Теоремата (МТ). Тя гласи следното:

Нека $a \geq 1$ и $b > 1$ са константи и нека $f(n)$ е положителна функция. Нека

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n),$$

където $\frac{n}{b}$ има смисъл или на $\left\lfloor \frac{n}{b} \right\rfloor$, или на $\left\lceil \frac{n}{b} \right\rceil$. Нека $k = n^{\log_b a}$. Тогава асимптотиката на $T(n)$ е седната:

**Случай 1.** Ако $f(n) \preceq k/n^{\varepsilon}$, за някоя положителна константа $\varepsilon$, тогава $T(n) = \Theta(k)$;

**Случай 2.** Ако $f(n) \asymp k$, тогава $T(n) = \Theta(k \cdot \log n)$. С други думи, $T(n) = \Theta\left(f(n) \cdot \log n\right)$;

**Случай 3.** $f(n) \succeq k \cdot n^{\varepsilon}$, за някоя положителна константа $\varepsilon$ и $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$, за някоя положителна константа $c$, такава че $0 < c < 1$ за всички достатъчно големи $n$, то $T(n) = \Theta\left(f(n)\right)$.

### 1.4. Примери за анализ на алгоритми

Ще разгледаме рекурсивна функция за двоично търсене (A е указател към сортиран масив):

```
int bin_search(int * A, int l, int r, int x){
        if(l > r) return -1;
        int m = l + (r - l) / 2;
        if(x == A[m]) return m;
        else if(x < A[m]) return bin_search(A, l, m-1, x);
        else return bin_search(A, m+1, r, x);
}
```

Ако $r - l = n$, то сложността по време се определя от рек. зависимост $T(n) = T\left(\frac{n}{2}\right) + 1$.

Според МТ, $a = 1$, $b = 2$, $k = n^{\log_b a} = n^{\log_2 1} = n^0 = 1$ и $f(n) = 1$. Следователно $k = 1 = f(n)$ и влизаме във втория случай на МТ, който ни дава, $T(n) = \Theta\left(\log(n)\right)$.

Други примери:

| Формула | Сложност | Примерен алгоритъм |
|---|---|---|
| $T(n) = T(n-1) + n$ | $\Theta(n^2)$ | Наивна сортировка |
| $T(n) = T\left(\frac{n}{2}\right) + n$ | $\Theta(n)$ | Среден случай на търсене на $k$-ти по големина елемент с **Quick Select** |
| $T(n) = 2T\left(\frac{n}{2}\right) + n$ | $\Theta\left(n \log(n)\right)$ | Бърза сортировка |
| $T(n) = 2T\left(\frac{n}{2}\right) + 1$ | $\Theta(n)$ | Търсене на минимален елемент по схемата „разделяй и владей“ |
| $T(n) = T\left(\sqrt{n}\right) + 1$ | $\Theta\left(\log\log(n)\right)$ | Основни операции в дървета на ван Емде Боас (вмъкване, изтриване, търсене) |

## 2. Абстрактни типове от данни

### 2.1. Дефиниция и идея

Често при програмирането използваме отделни компоненти наготово, без да се интересуваме как точно си вършат работата. Казваме, че такива компоненти са **абстракции**. Абстракциите улесняват писането на софтуер, като не ни карат да мислим за ненужни детайли и всеки път да преоткриваме топлата вода.

Абстрактен тип данни е такъв тип, който единствено дефинира какви операции поддържа и какъв **underlying** тип данни използва.

### 2.2. Интерфейс и реализация

Поддържаните операции от някой абстрактен тип се наричат негов **интерфейс**. Интерфейсът е това, което клиентския код вижда и ползва. **Реализацията** е конкретен начин, по който е написан даден интерфейс – и съответно, абстрактен тип. Тя остава скрита от клиентския код. За един интерфейс може да има много реализации.

## 3. Свързани списъци

### 3.1. Описание

Свързаният списък е линейна структура от данни, която се състои от върхове (или наричани още възли) и ограничени връзки между тях. В едносвързания списък всеки връх пази указател към непосредствено следващия го връх, а в двусвързания – всеки връх има и указател и към предходния връх. Хубавият списък пази указател освен към първия връх (наричан още глава) и към последния възел (наричан още опашка). По този начин ще може да осигури константна сложност за добавяне и изтриване на върхове от главата и добавяне към опашката на списъка. Ако е двусвързан и изтриване в края на списъка. Всички останали операции като търсене, вмъкване или изтриване на позиция имат линейна сожност по броя на върховете в списъка (тъй като в най-лошия случай ще трябва да се обходи целия списък). В слуай, че имаме указател към даден връх от списъка, ще имаме възможност да добавиим или изтрием с константна сложност следващия елемент след този, към който указателят, който имсме – сочи или следващия, текущия и предходния при двусвързан. Конкатениране с оригинал на друг списък също е с константна сложност.

### 3.2. Реализация Singly Linked List

## 4. Стек

### 4.1. Описание

Стекът е линейна структура от данни, която представлява колекция наподобяваща шиш като от играта „Кулите на Ханой“. На този шиш се наслагват елементите на колекцията, като се поддържат операциите добавяне най-отгоре, премахване най-отгоре и разглеждане на най-горния елемент. Всички тези операции се изпълняват за константна сложност по време. Операциите описват LIFO (Last In, First Out) структура – последният добавен елемент е и първия изваден. Стековете са много удобни за заместване на рекурсия, обръщане на линейни структури или за обхождане в дълбочина на нелинейни структури от данни. Могат да се използват и при множество алгоритми като алгоритъма на Таржан за намиране на срязваши точки и мостове в граф. Намиране на диаметър в граф и други. Използват се и в множество персистентни структури от данни (като например за реализиране на Undo команди в текстови редактори). Пресмятане на математически изрази с обратна полска нотация и други.

### 4.2. Реализация Stack

Стек може да се реализира по няколко начина:

- Чрез статичен масив. При тази реализация стекът има краен размер, за да бъде забранено преоразмеряването на масива, което е скъпа (линейна) операция и може да наруши константната сложност при добавяне (при удвояване на размера) или изтриване (при свиване – shrink). Пази се единствено индекса на най-горния елемент.
- Чрез свързан списък. При едносвързан списък, най-горният елемент е първия възел в списъка, а добавянето става само откъм главата на списъка.

## 5. Опашка

### 5.1. Описание

Опашката е линейна структура от данни, която представлява колекция наподобяваща тръба, в която се вкарват елементи от единя край, а от другия край се изваждат. Поддържат се операциите добавяне на елемент в единя край и изваждане или поглеждане на елемента от другия край. Тези операции се изпълняват за константна сложност по време. Операциите описват FIFO (First In, First Out) структура – първият добавен елемент е и първия изваден. Опашките се използват при обхождане на нелинейни структури в ширина, метода на вълната, за намиране на най-кратки пътища, при $A*$ алгоритъма за търсене на оптимално решение, за намиране на диаметър на граф и при реализиране на комуникационна тръба чрез семафори, както и при реализиране на силен семафор и други. Комбинация от опашка и стек може да се използва за конвертиране на инфиксна нотация в постфиксна. Опашките се използват в множество входно-изходни операции.

Съществуват и циклични опашки, които представляват примка. При тях последният елемент сочи към първия.

### 5.2. Реализация Queue

Възможните реализации са както при стека с тази разлика, че ако се използват масиви трябва да се пази индекса на първия и последния елемент. При циклични опашки се използва модулно деление и нетривиална логика.

При реализация със свързан списък се добавят елементи в единя край и се вадят от другия (обикновено добавянето е към опашката на свързания списък, а изваждането от главата на свързания списък, за да може да се реализират и чрез едносвързан списък).

## 6. Дървета

### 6.1. Описание

Дърветата са нелинейна, рекурсивна и йерархична структура от данни и представляват частен случай на графите, в който няма цикли. Дървото има корен (връх без родител) и всеки негов връх може да има нула или много на брой деца, но само един родител (без корена). Върховете без деца се наричат листа. Най-дълъг път от корен към листо се нарича височина на дървото. Дълбочина на връх наричаме броя на върховете от корена до този връх (без самия връх). Разклонение на дървото наричаме максимален брой деца на връх от дървото. Поддърво на дървото $t$, наричаме всяко дърво имащо за корен връх от $t$.

### 6.2. Видове дървета

- **Двоични дървета**. Това са дървета, които имат максимална разклоненост 2. Ако всеки връх, който не е листо има точно 2 деца, то дървото се нарича **пълно двоично дърво**. Ако разликата $d$ между минималната и максималната дълбочина на листо е не повече от 1, то дървото е **балансирано двоично дърво**.
- **Двоични дървета за търсене**. Това са двоични дървета, в които правим разлика между ляво и дясно дете. При тях лявото поддърво на всеки връх $v$ има ключове с по-малка стойност от тази на $v$ и дясно поддърво има ключове с по-големи (или равни) стойности от тази на $v$. Лявото и дясното поддърво на всеки връх отново са двоични дървета за търсене.
- **Балансирани дървета за търсене**. Представителите на тези структури от данни са **AVL** дърво и „**червено-черно**“ дърво. Те са подобни на двоичните дървета за търсене с допълнението, че използват някаква нетривиална логика за балансиране, което ги поддържа с $d \leq 1$. Това гарантира максимална сложност по време $O\left(\log(n)\right)$ за добавяне, изтриване и търсене на елемент.
- **Пирамида (Heap)**. Пирамидите са почти пълни и частично наредени балансирани дървета. Позволено е да липсват върхове единствено на последното ниво. Тази тяхна характеристика им позволява да се съхраняват много удобно в масив като връх с индекс $i$ има родител с индекс $\lfloor (i-1)/2 \rfloor$ и ляво и дясно дете съответно на индекси $2i+1$ и $2i+2$. Пирамидите са два вида: **min-heap** и **max-heap**, като името индикира за това кой елемент се намира в корена на пирамидата. Извличането на този елемент изисква константна сложност, а операции като добавяне и премахване на елемент изискват логаритмична сложност. Тази структура от данни се използва за имплементиране на приоритетна опашка. Съществуват и нейни разновидности като рандомизирана пирамида. Пирамидата се използва като помощна структура от данни в алгоритъма на Дейкстра, както и в множество други алгоритми. Пирамидата може да се построи за линейно време чрез алгоритъма на Флойд.
- **Treap (Дерамида от руски)**. Както показва името на тази дървовидна структура – тя представлява смесица от пирамида и бинарно дърво за търсене. Среща се още като Декартово дърво и е изключително интересна структура. Всеки връх се характеризира с ключ и приоритет. Без приоритетите дървото ще е обикновено дърво за търсене спрямо ключовете. По отношение на приоритетите, дървото спазва правилата на пирамида – по всеки клон има наредба. Приоритетите, ако не се повтарят, позволяват еднозначното построяване на това дърво. Това води до дърво, което не е дегенерат в очакване. Поддържат логаритмична сложност по време за основните операции – добавяне, изтриване и търсене. Сечение и обединение на две декартови дървета се изпълняа за $O\left(M \log(N/M)\right)$, което е впечатляващо бързо. Това е и една от

причините те да бъдат удачна помощна структура за построяване на суфиксен автомат. Много удобна структура за заявки за намиране на $k$-ти по големина елемент.

- **Сегментни дървета**. Тези дървета позволяват да отговаряме ефективно на заявки върху интервали от масив. Те са достатъчно гъвкави (за разлика от разредената таблица – Sparse Table), за да позволяват и добавяне и изтриване на елементи. Позволяват т.нар. техника на мързеливо пропагандиране (разпространение), което допринася за бързината им в множество случаи като актуализиране на цели сегменти, обръщане на подинтервали и други.

Други дървета: X-бързо дърво на Willard (двоично дърво, където всяко поддърво съхранява стойности, имащи двоични представяния с общ префикс), дърво на ван Емде Боас (използва коефициент на свиване $\sqrt{n}$, което прави операциите по добавяне, изтриване, търсене и намиране на предшественик и наследник за сложност по време $O\left(\log\log(n)\right)$ на цената на някой ограничения на домейна/вселената), дърво на Уконен (изключително ефективно за търсене в текст, известно е още като суфиксно дърво), k-дименсни дървета (k-d дървета, това е структура от данни за организиране на точки в k-мерното пространство), DSU (Disjoint Set Union, ефективно търсене на множество, в което се намира елемент и обединение на елементи в дървовидна структура със сложност обратната функция на Акерман, което практически е константна сложност), дърво на Фенуйк и много други.

### 6.3. Реализация Binary Search Tree, Tree

## 7. Бавни сортировки Selection Sort, Insertion Sort, Bubble Sort

Бавните сортировки се характеризират с квадратична сложност по време, спрямо броя на елементите, които сортираме.

### 7.1. Сортиране с пряка селекция (Selection sort)

Идеята тук е всеки път да намираме минимален елемент и да го поставяме на първа позиция в текущия масив. След всяка итерация текущия масив се смалява с една позиция от ляво. Предимството на този алгоритъм е, че е прост и извършва малко на брой размени. Алгоритъмът не е стабилен (ако има два или повече елемента с еднаква стойност – след сортировката не се гарантира, че реда им ще е такъв какъвто е бил преди сортировката).

```
void selection_sort(int * arr, const int size){
        for (int i = 0; i < size; i++){
                int min_ind = i;
                for(int j = i + 1; j < size; j++)
                        if(arr[ j ] < arr[min_ind]) min_ind = j;
                std::swap(arr[ i ], arr[min_ind]);
        }
}
```

### 7.2. Сортиране с вмъкване (Insertion sort)

Познато още като сортирането на картоиграча. Инвариантата му е, че на $i$-тата стъпка (броейки от 0), подмасивът с индекси $0...i-1$ сортиран. Следователно на всяка стъпка се намира мястото на елемента $A[i]$. Предимството идва от факта, че това може да отнеме по-малко от $i$ стъпки, следователно при почти сортирани масиви, времето за работа може да е линейно. Алгоритъмът е стабилен.

```
void insertion_sort(int * arr, const int size){
        for (int i = 1; i < size; i++){
                const int new_val = arr[ i ];
                int ind = i;
                while(ind > 0 && arr[ind - 1] > new_val){
                        arr[ind] = arr[ind - 1];
                        ind--;
```

```
            }
            arr[ind] = new_val;
        }
    }
```

### 7.3. Сортиране с метода на балончето (Bubble sort)

Този метод сравнява всеки елемент със съседния му и прави размени, като по този начин при всяко обхождане изпраща един елемент на точното му място (в края на текущия обхождан интервал). Алгоритъмът за сортиране по метода на балончето е стабилен.

```
void bubble_sort(int * arr, const int size){
    bool swapped;
    for (int i = 0; i < size - 1; i++){
        swapped = false;
        for(int j = 0; j < size - i - 1; j++){
            if(arr[ j ] > arr[j + 1]){
                std::swap(arr[ j ], arr[j + 1]);
                swapped = true;
            }
        }
        if(!swapped) break;
    }
}
```

Бавните сортировки може да се използват от бързите в дъното на рекурсията, тъй като за малък на брой елементи бавните сортировки се държат по-добре от бързите. Обикновено за тези цели се използва сортиране с вмъкване, тъй като се очаква да е най-бърз за частично наредени входни данни. И трите разгледани бавни сортировки използват константна памет, т.е. $\Theta(1)$ допълнителна памет (inplace).

### 8. Бързо сортиране (QuickSort) Quick sort

Доазано е, че всеки сортиращ алгоритъм базиран на сравнения, ще извърши $\Omega\left(n\log(n)\right)$ сравнения в най-лошия случай. Всеки алгоритъм, който извършва по-малко сравнения използва допълнителна информация за входните данни, които сортира.
QuickSort е типичен алгоритъм за схемата Разделяй-и-Владей. Идеята е да бъде избран елемент от масива, който наричаме **pivot** и спрямо него масивът да бъде пренареден така, че в лявата част (може и да е празна, това зависи от pivot) да са само елементи, по-малки или равни на pivot, а вдясно от тях, само елементи, по-големи от pivot. Каква точно е наредбата в лявата и дясната част няма значение, но ние знаем, че поне един (pivot-ния) елемент ще е на правилната си позиция (позицията, на която би се намирал при сортиран масив). След това повтаряме процедурата в лявата и дясната половина рекурсивно. Изборът на pivot е от огромно значение за сложността по време. Не е задължително pivot да е елемент от масива, но в нашия случай ние ще го избираме да е от масива. Ако pivot е елемент от масива, то идеалният pivot би бил медианата.

Алгоритъмът използва помощна функция **partition**, която използва константна памет и се изпълнява за линейна сложност по време. В нашия пример ще използваме **Lomuto-Partition**. Алгоритъма не е стабилен.

Сложността на QuickSort се описва от рекурентното уравнение: $T(n) = 2T\left(\frac{n}{2}\right) + n$, като допускаме , че pivot-а разбива масива на две равни части в очакване. Тогава от МТ лесно се вижда, че влизаме във втория случай и $T(n) = \Theta\left(n\log(n)\right)$ (в очакване).