

# Дървовидни структури от данни и алгоритми върху тях

ИТ Кариера



Учителски екип

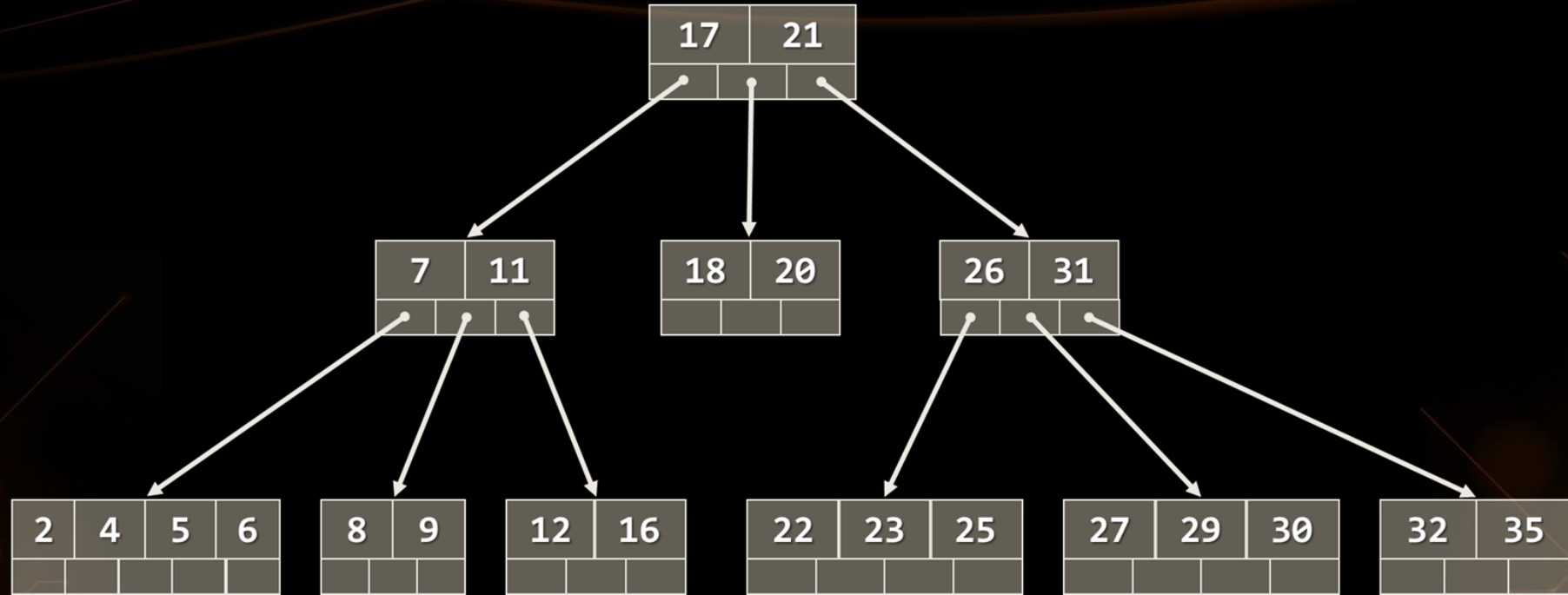
Обучение за ИТ кариера

<https://it-kariera.mon.bg/e-learning>

# Съдържание

- Дървета и дървовидни структури
- Подредени двоични дървета, балансирани дървета, В-дървета
- Упражнения: структура от данни “дърво”, използване на класове и библиотеки за дървовидни структури
- Обхождания в дълбочина и ширина (DFS и BFS)
- Упражнения: обхождане в дълбочина (DFS)
- Упражнения: обхождане в ширина (BFS)



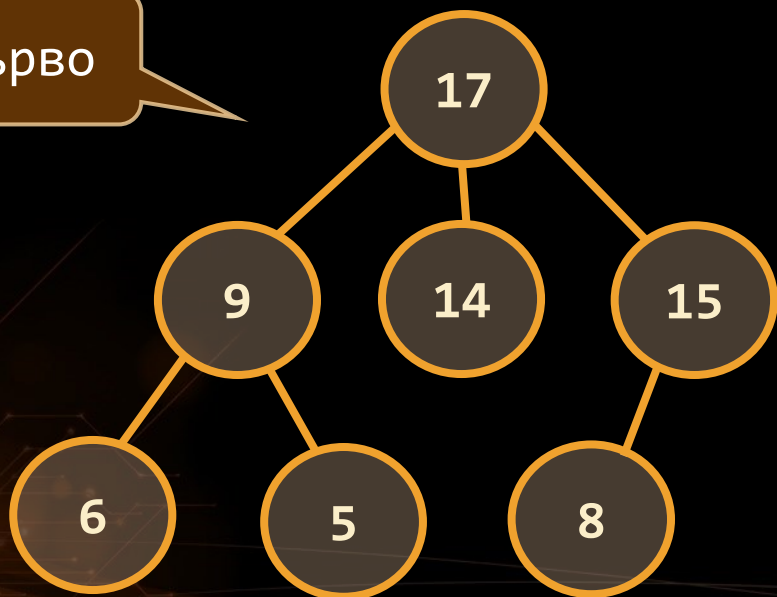


# Дървовидни структури от данни

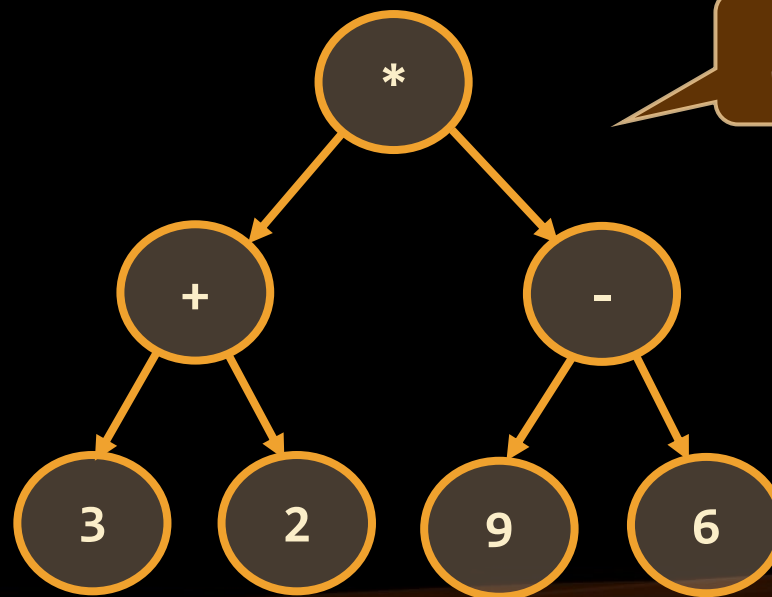
# Дървовидни структури от данни

- Дървовидните структури от данни са:
  - Разклонени йерархични структури от данни
  - Изградени от възли
  - Всеки възел е свързан с други възли (разклонения на дървото)

Дърво



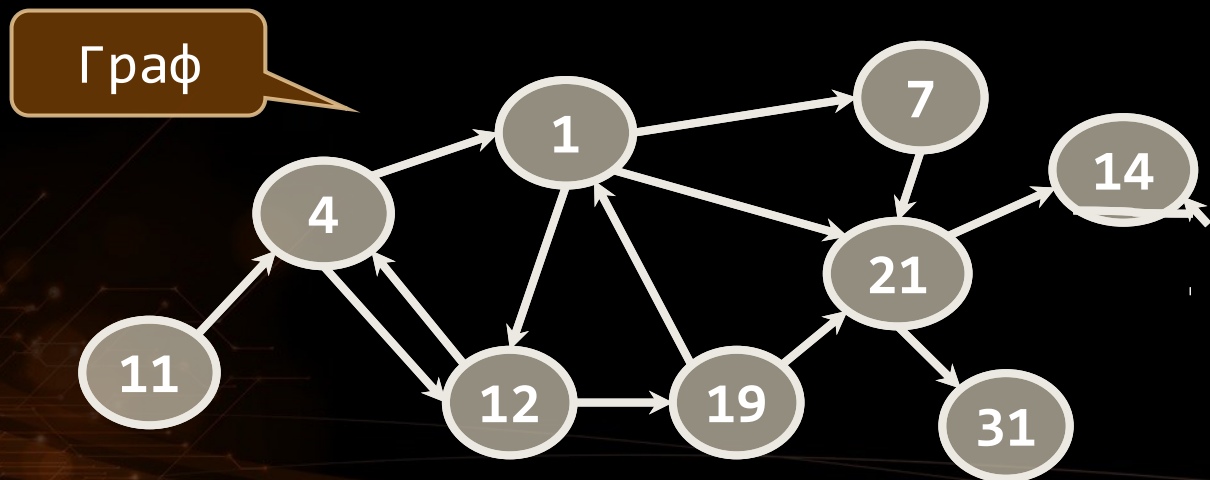
Двоично дърво

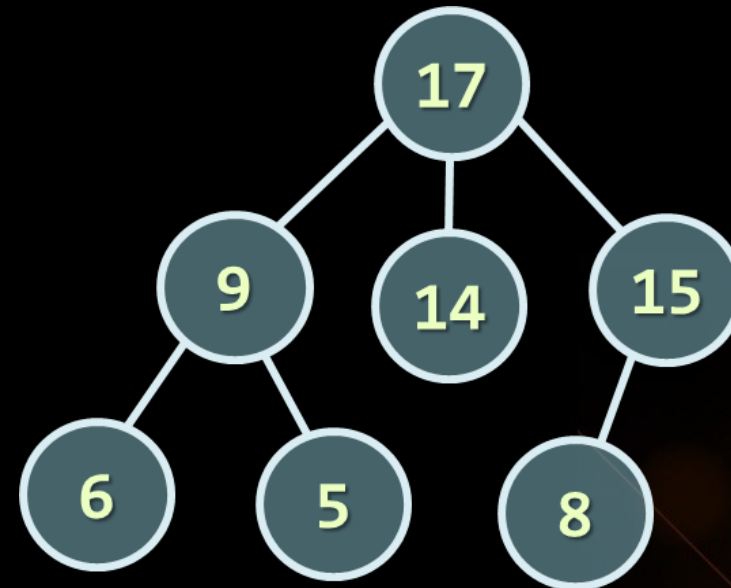
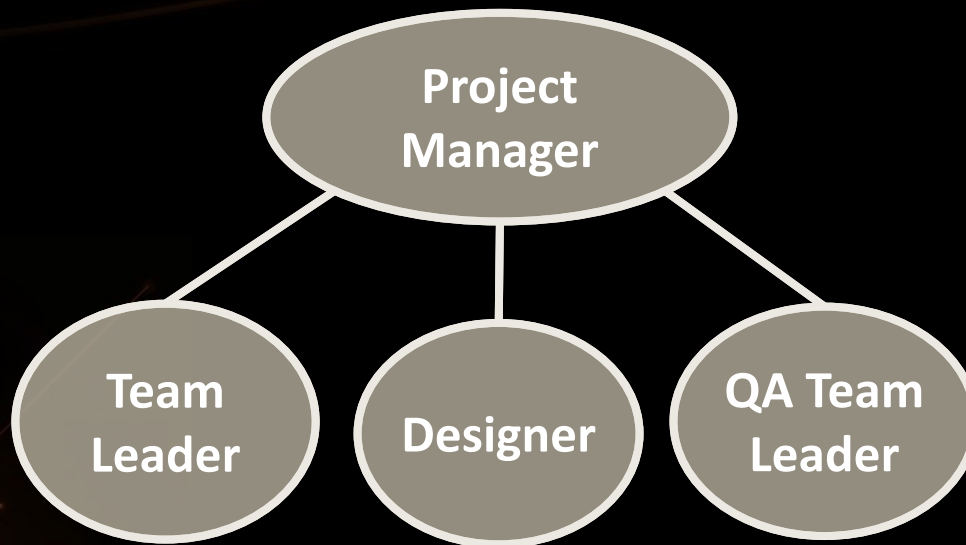




# Дървовидни структури от данни

- Дървовидните структури от данни:
  - **Дървета** - двоични, балансирани, подредени и др.
  - **Графи** - ориентирани, неориентирани, с тегла и др.
  - **Мрежи** - графи с особени свойства



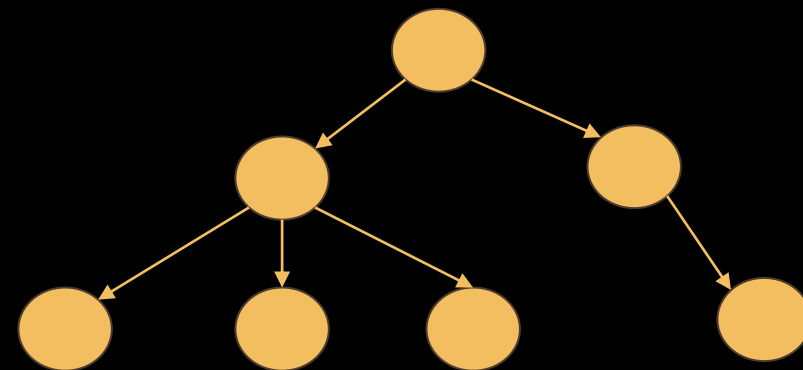


# Дървовидни структури от данни

## Терминология

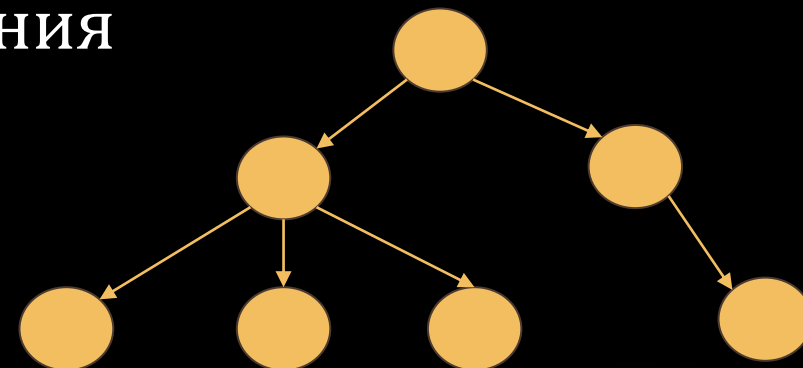
# Дървета – дефиниция

- Нека  $D = \{V, E\}$  е кореново дърво
- Всяко дърво се образува от възли и дъги, които ги свързват
- Формално върховете могат да бъдат от два вида:
  - Родител
  - Наследник
- Върхът без родител се нарича “корен”
- Всяко дърво има само корен
- Върх без наследници се нарича “листо”



# Дърво – обща дефиниция

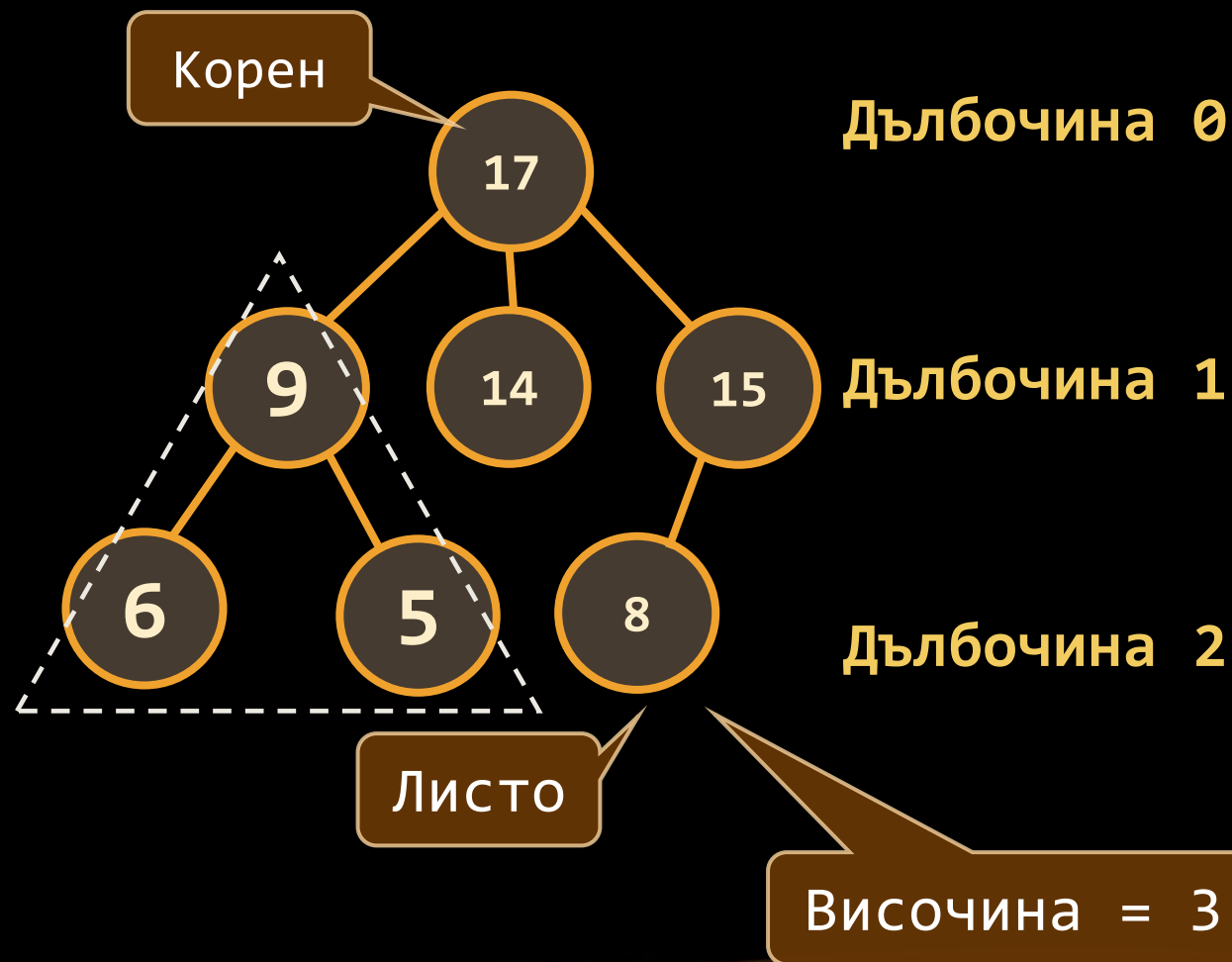
- Дърво от тип  $T$  е структура, образувана от:
  - Елемент от тип  $T$ , наречен “корен”
  - Крайно множество елементи от тип  $T$ , наречени “поддървета”
- Дървото се бележи с  $T = \{V, E\}$ , където:
  - $V$  е множеството от възли в структурата
  - $E$  е множеството от ребра в структурата
- Дървета, в които  $T$  има  $k$  на брой разклонения наричаме  $k$ -ични дървета





# Дървовидни структури от данни – терминология

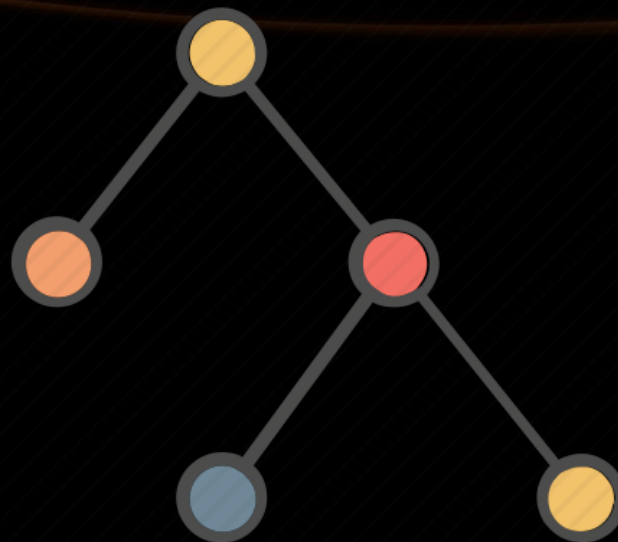
- Възел, ребро
- Корен, родител, дете, брат
- Дълбочина, височина
- Под-дърво
- Вътрешен възел, листо
- Предшественик, наследник



# Двоични дървета

Възлите на двоичните дървета имат по не повече от две разклонения

- Двоични дървета
- Няма правила за подредба на елементите
- Наредени (сортирани) двоични дървета (правила за подредба на възлите)
- Двоични дървета за търсене (частен случай на сортирани дървета):
  - Лявото разклонение на всеки възел има по-малка стойност от стойността на възела
  - Дясното разклонение на всеки възел има по-голяма стойност от стойността на възела.



# Двоични дървета

## Реализация

# Рекурсивна дефиниция на дървета

- Рекурсивна дефиниция на дървета:
- Всеки възел е дърво
- Възлите имат 0 или много деца, които също са дървета

```
public class Tree<T>
```

```
{
```

```
    private T value;
```

Стойността  
на възела

```
    private IList<Tree<T>> children;
```

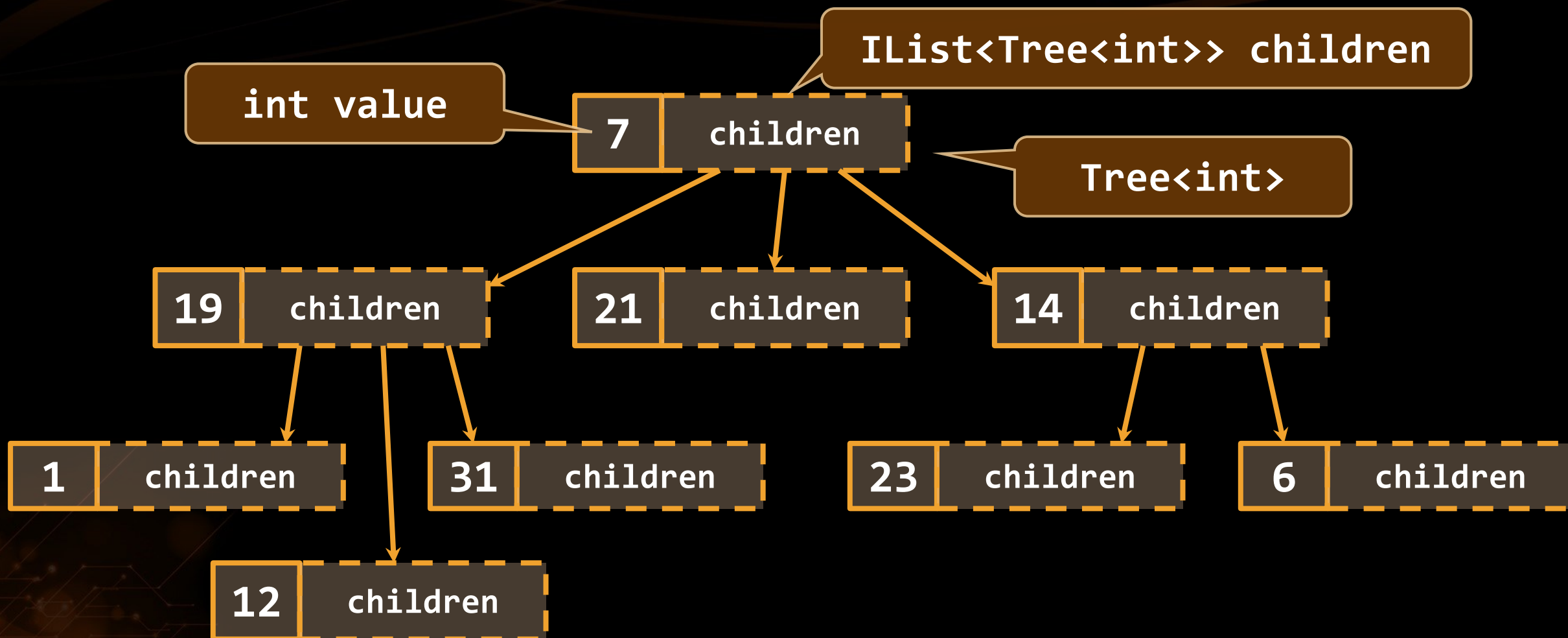
```
    ...
```

```
}
```

Списък с възли -  
деца (поддървета)



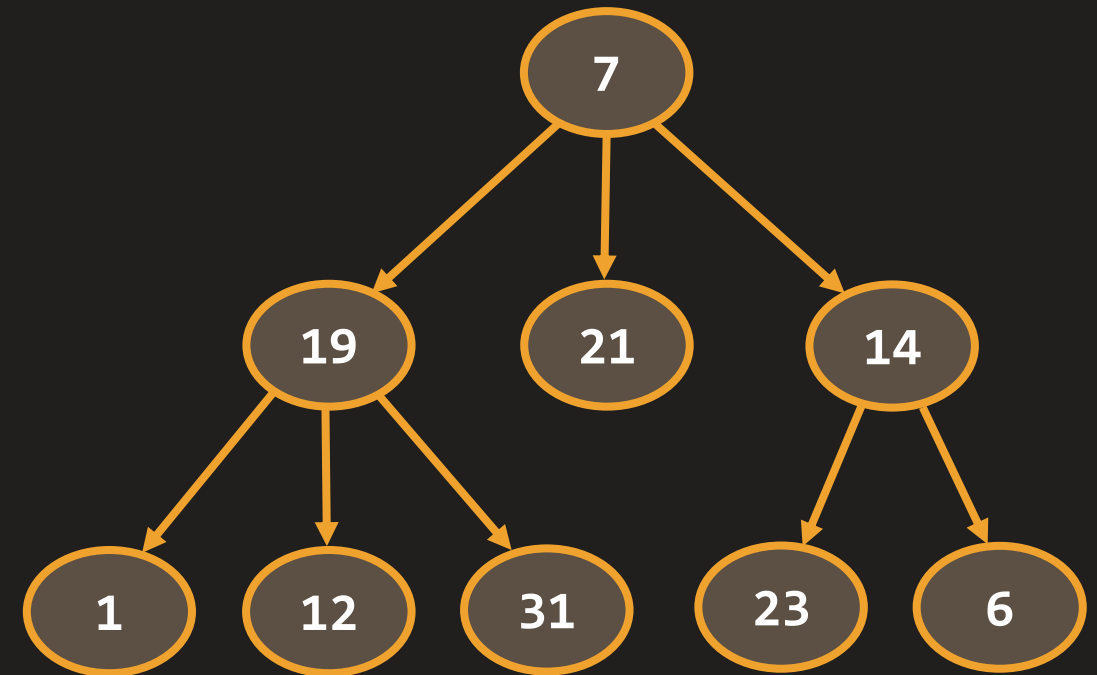
# Структурата Tree<T> - Пример



# Задача: Реализирайте възел на дърво

- Създайте рекурсивно дефинирана структура описваща дърво

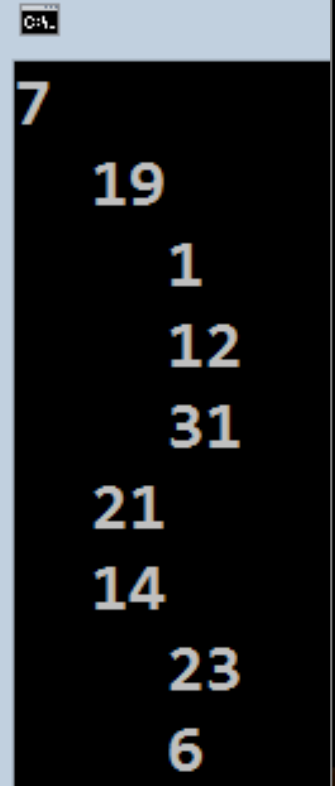
```
Tree<int> tree =  
    new Tree<int>(7,  
        new Tree<int>(19,  
            new Tree<int>(1),  
            new Tree<int>(12),  
            new Tree<int>(31)),  
        new Tree<int>(21),  
        new Tree<int>(14,  
            new Tree<int>(23),  
            new Tree<int>(6))  
    );
```



# Задача: Отпечатайте елементите на дърво

- Отпечатайте на конзолата елементите на дърво с 2 интервала отместване за всяко следващо ниво

```
Tree<int> tree =  
    new Tree<int>(7,  
        new Tree<int>(19,  
            new Tree<int>(1),  
            new Tree<int>(12),  
            new Tree<int>(31)),  
        new Tree<int>(21),  
        new Tree<int>(14,  
            new Tree<int>(23),  
            new Tree<int>(6))  
    );
```

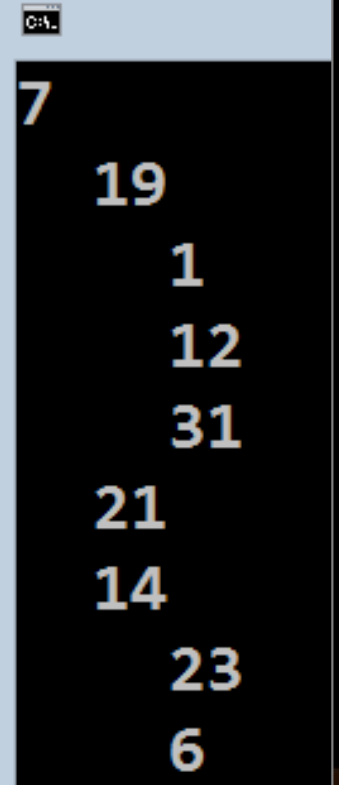


```
7  
  19  
    1  
    12  
    31  
  21  
  14  
    23  
    6
```

# Решение: Отпечатайте елементите на дърво

- Рекурсивен алгоритъм за обхождане на елементите на дърво

```
public class Tree<T>
{
    ...
    public void Print(int indent = 0)
    {
        Console.Write(new string(' ', 2 * indent));
        Console.WriteLine(this.Value);
        foreach (var child in this.Children)
            child.Print(indent + 1);
    }
}
```



```
7
  19
    1
    12
      31
  21
    14
    23
      6
```





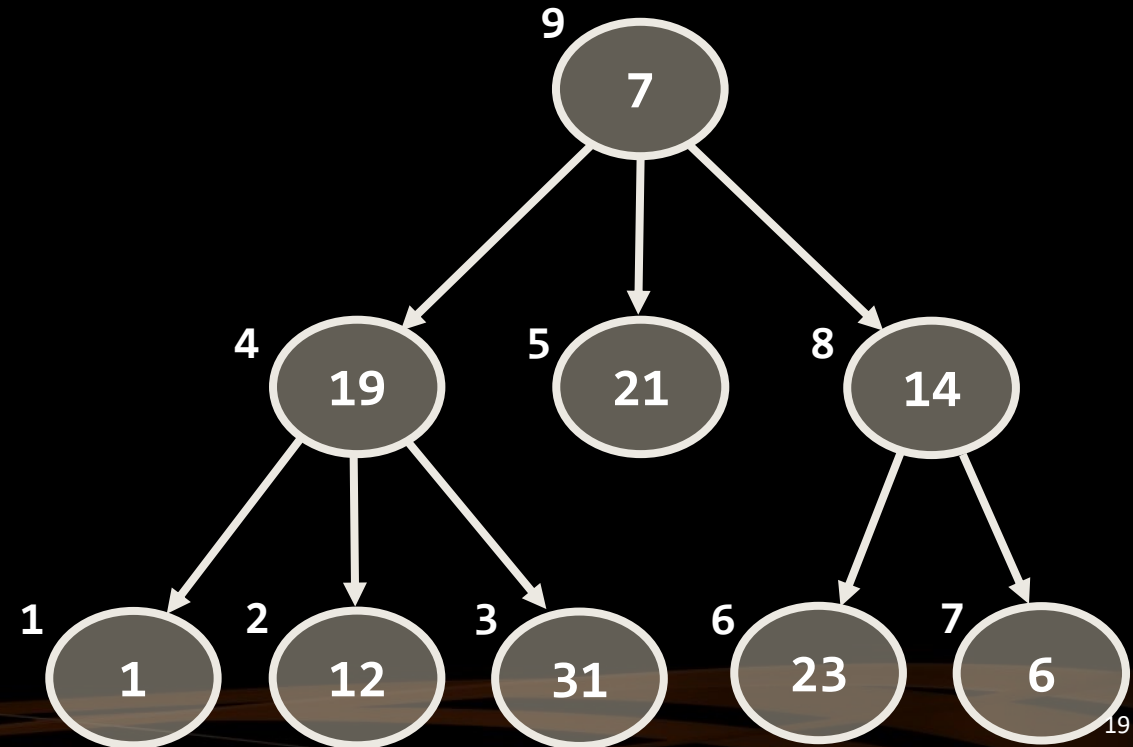
# Обхождане на дървовидни структури

- Обхождане на дърво представлява посещаването на всеки негов възел точно по веднъж
- Последователността на обхождането може да варира, в зависимост от алгоритъма за обхождане:
  - Обхождане в дълбочина (DFS):
    - Първо се посещават наследниците на възела
    - Стандартна реализация - чрез рекурсия
  - Обхождане в ширина (BFS):
    - Първо се посещава най-близкия възел
    - Стандартна реализация - чрез опашка

# Обхождане в дълбочина (DFS)

- Обхождане в дълбочина (DFS) - за всеки възел:
  - Посещават се всички негови деца
  - Ако възела няма деца или всички негови деца са вече обходени се обработва стойността му

```
DFS (node)
{
  for each child c of node
    DFS(c);
  print node;
}
```

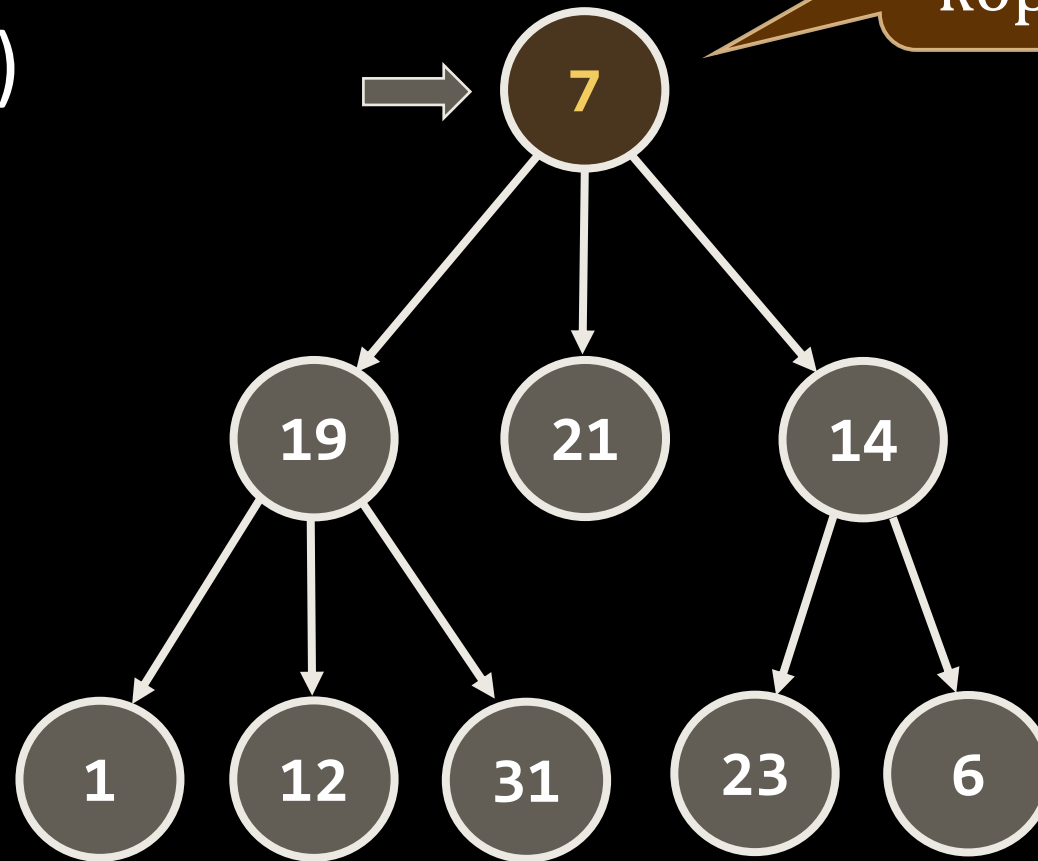


# DFS в действие (стъпка 1)

✦ Стек: 7

✦ Изход: (празен)

Стартираме DFS от  
корена на дървото

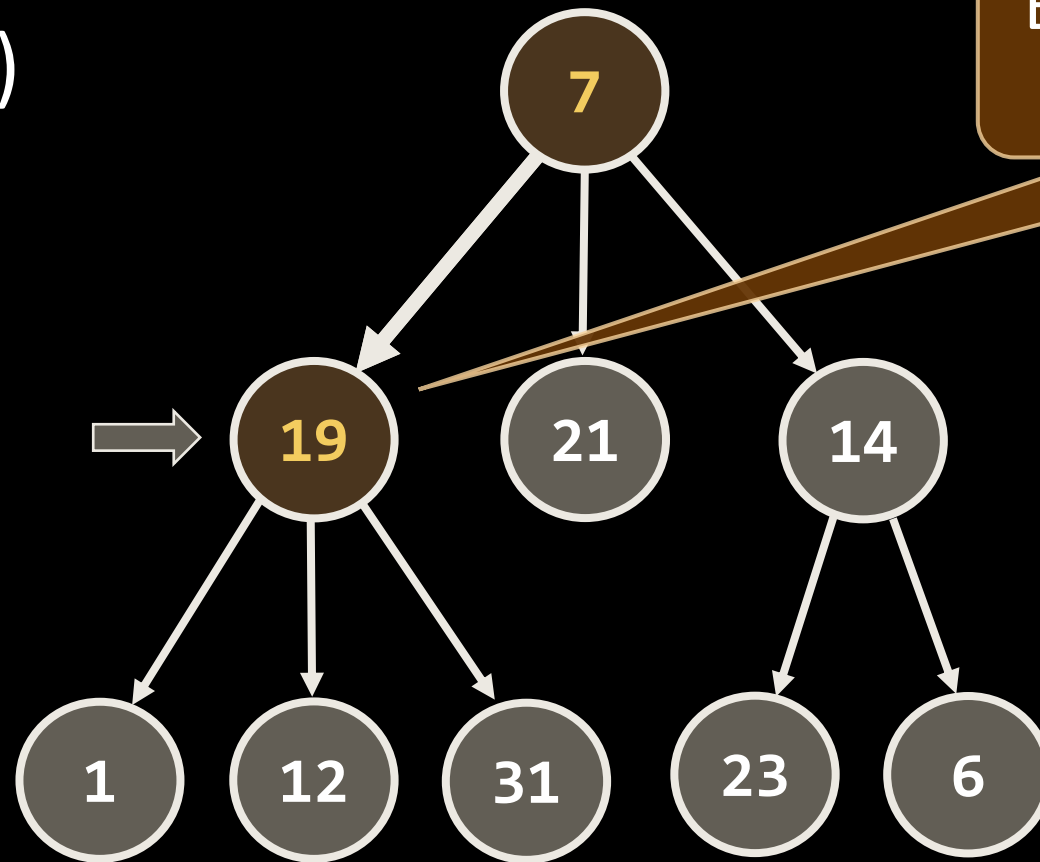




## DFS в действие (стъпка 2)

✦ Стек: 7, 19

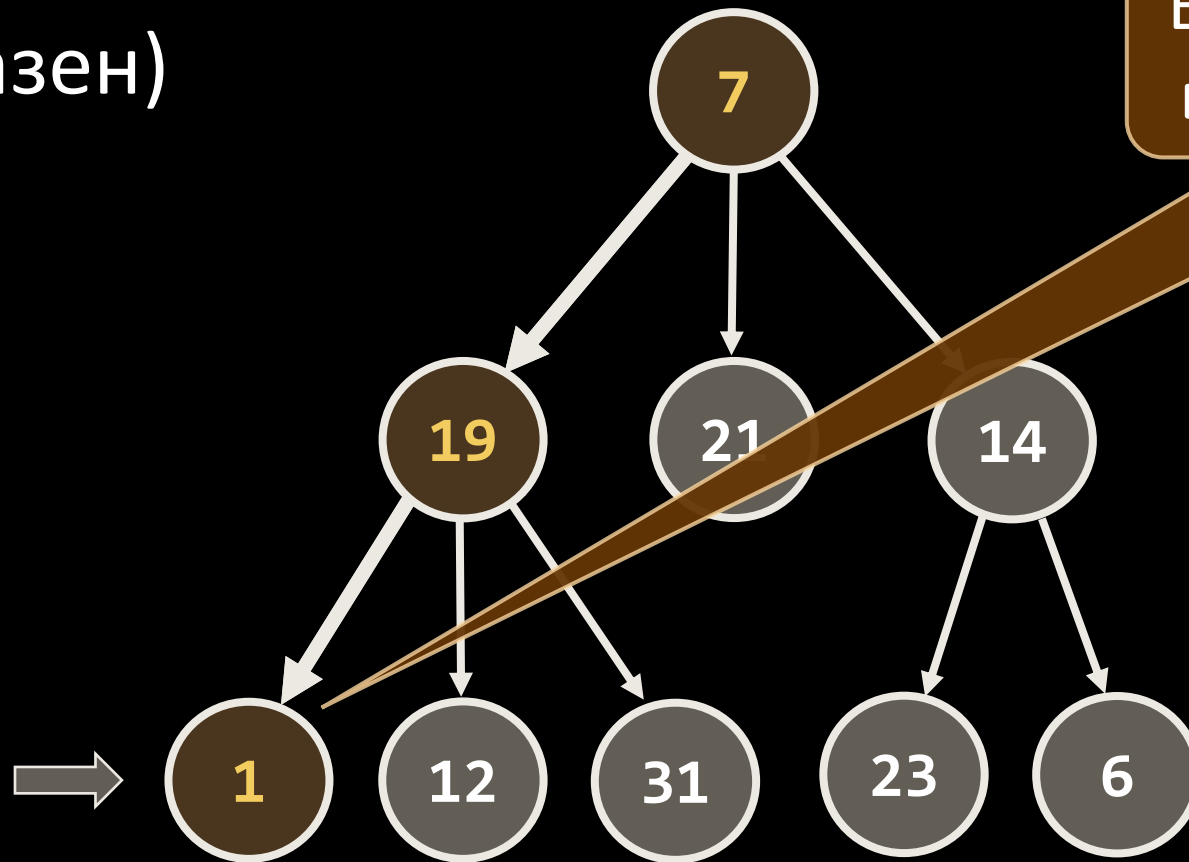
✦ Изход: (празен)



Влизаме рекурсивно  
в първия наследник

## DFS в действие (стъпка 3)

- ✦ Стек: 7, 19, 1
- ✦ Изход: (празен)

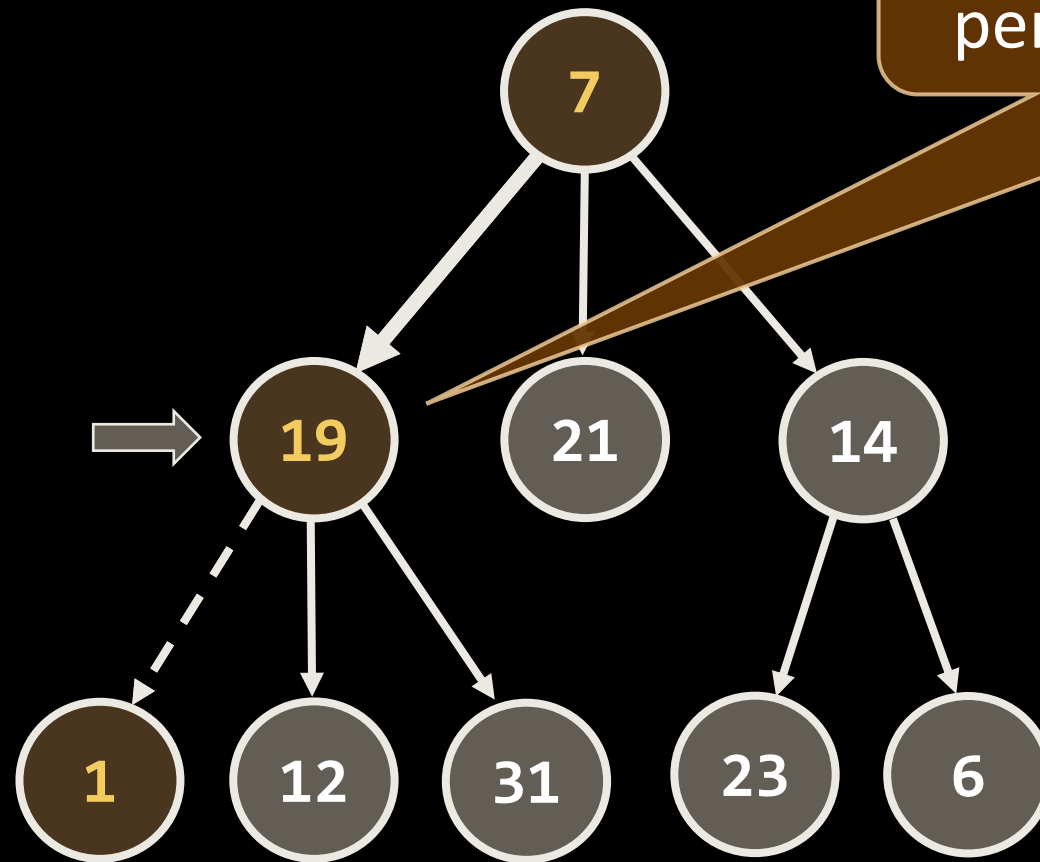


Влизаме рекурсивно  
в първия наследник

## DFS в действие (стъпка 4)

✦ Стек: 7, 19

✦ Изход: 1

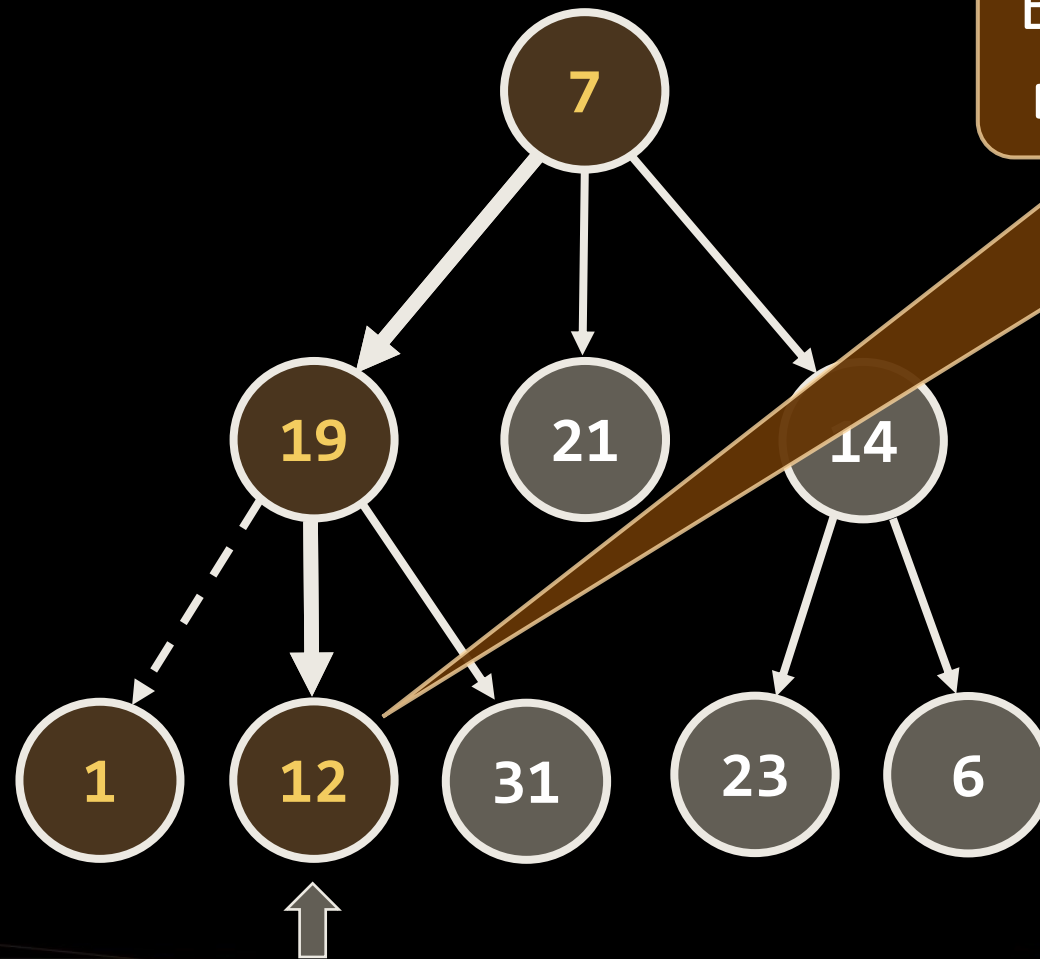


Връщаме се обратно от рекурсивното извикване

# DFS в действие (стъпка 5)

✦ Стек: 7, 19, 12

✦ Изход: 1

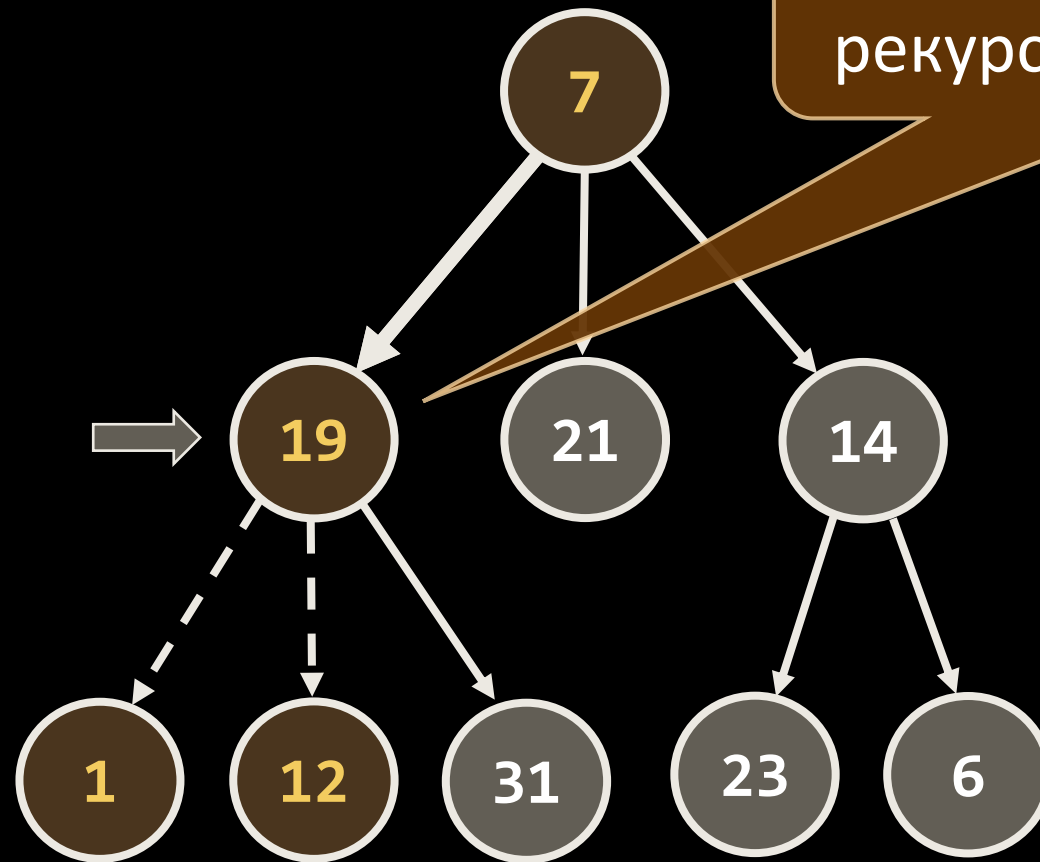




# DFS в действие (стъпка 6)

✦ Стек: 7, 19

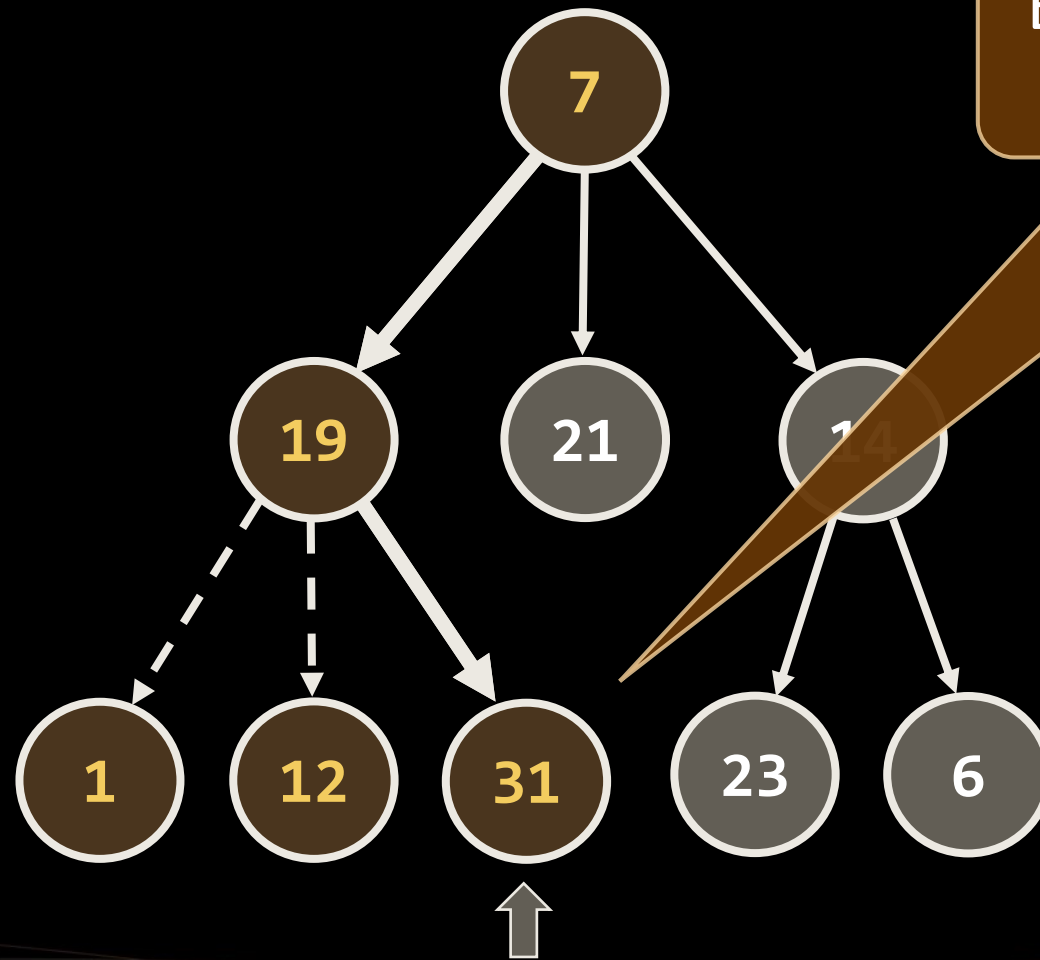
✦ Изход: 1, 12



# DFS в действие (стъпка 7)

✦ Стек: 7, 19, 31

✦ Изход: 1, 12

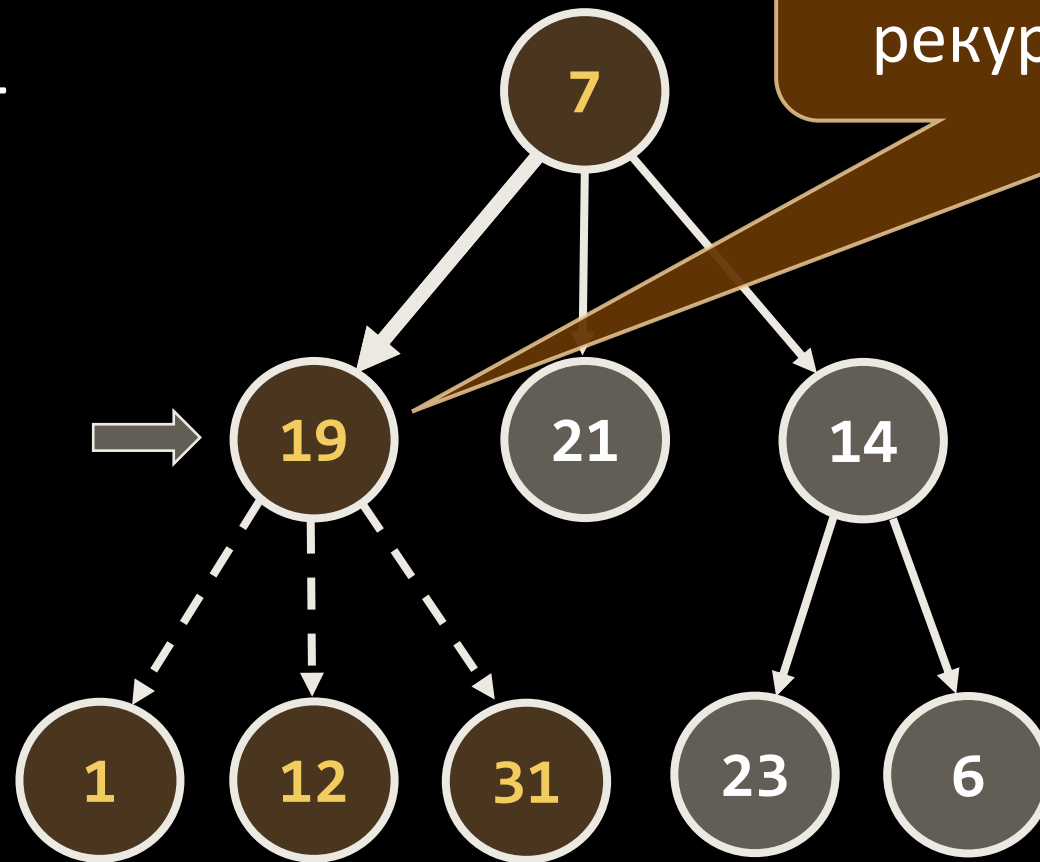


Влизаме рекурсивно  
в първия наследник

## DFS в действие (стъпка 8)

✦ Стек: 7, 19

✦ Изход: 1, 12, 31

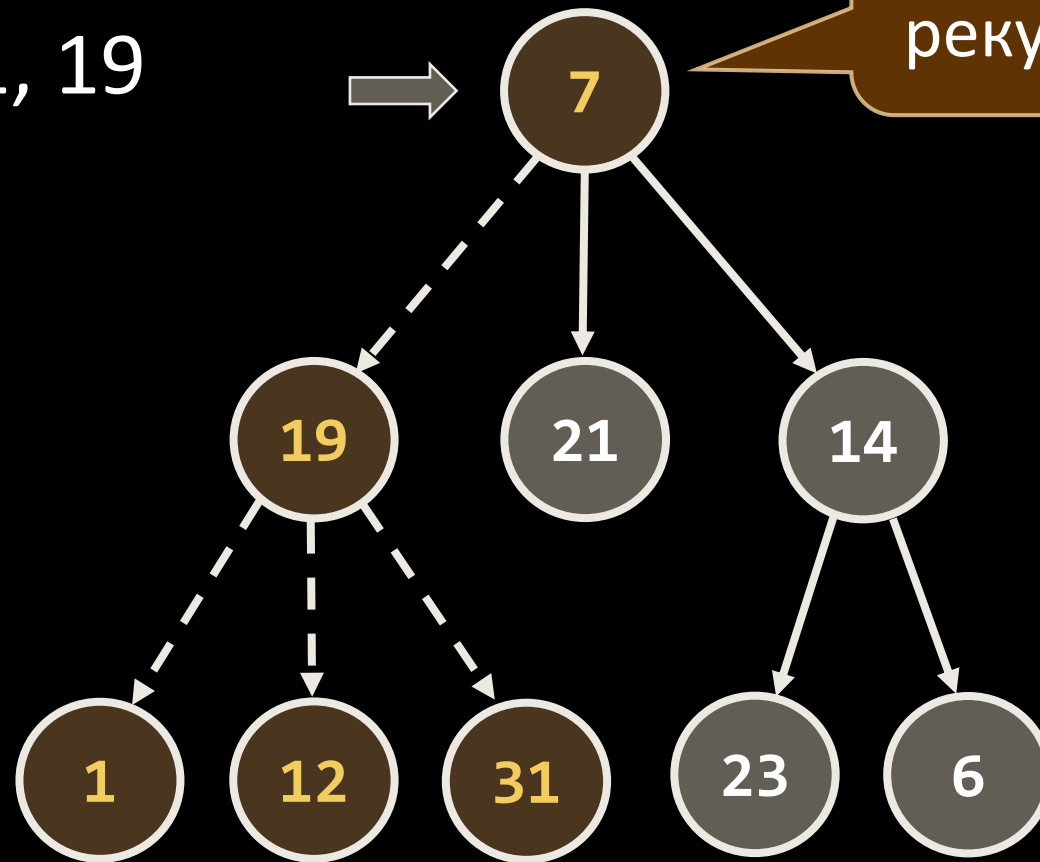


Връщаме се обратно от рекурсивното извикване

# DFS в действие (стъпка 9)

✦ Стек: 7

✦ Изход: 1, 12, 31, 19



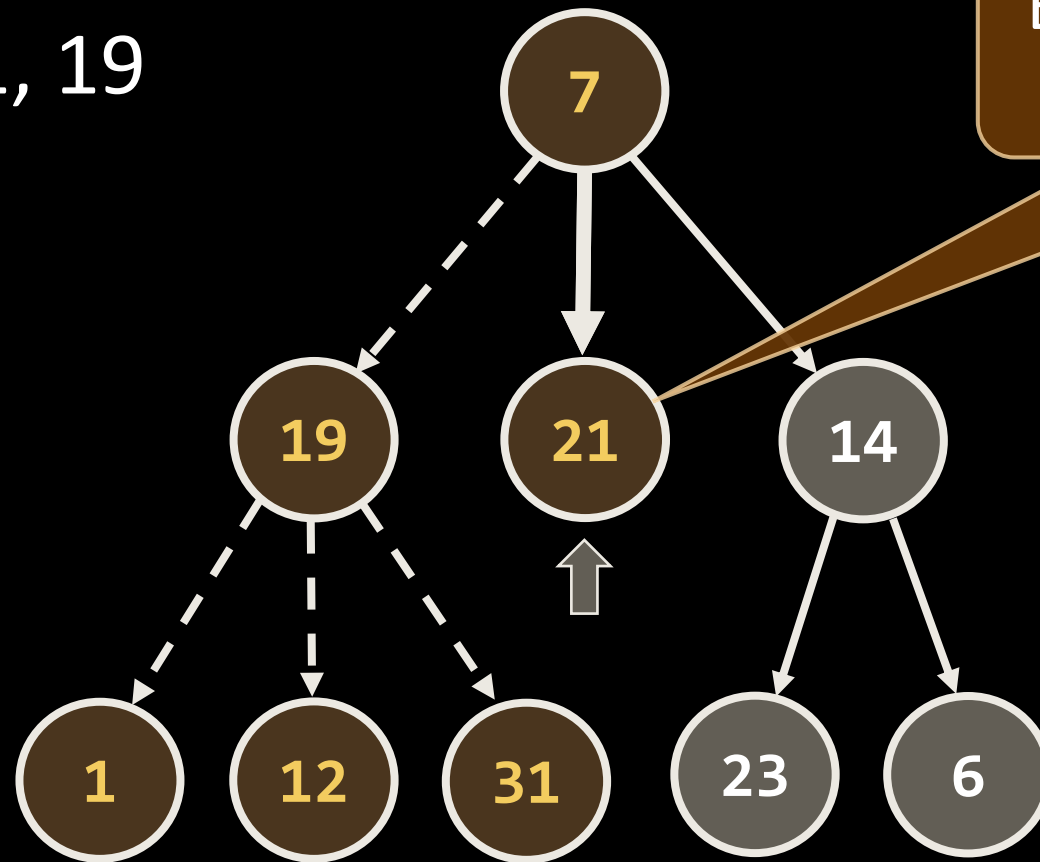
Връщаме се обратно от рекурсивното извикване



# DFS в действие (стъпка 10)

✦ Стек: 7, 21

✦ Изход: 1, 12, 31, 19



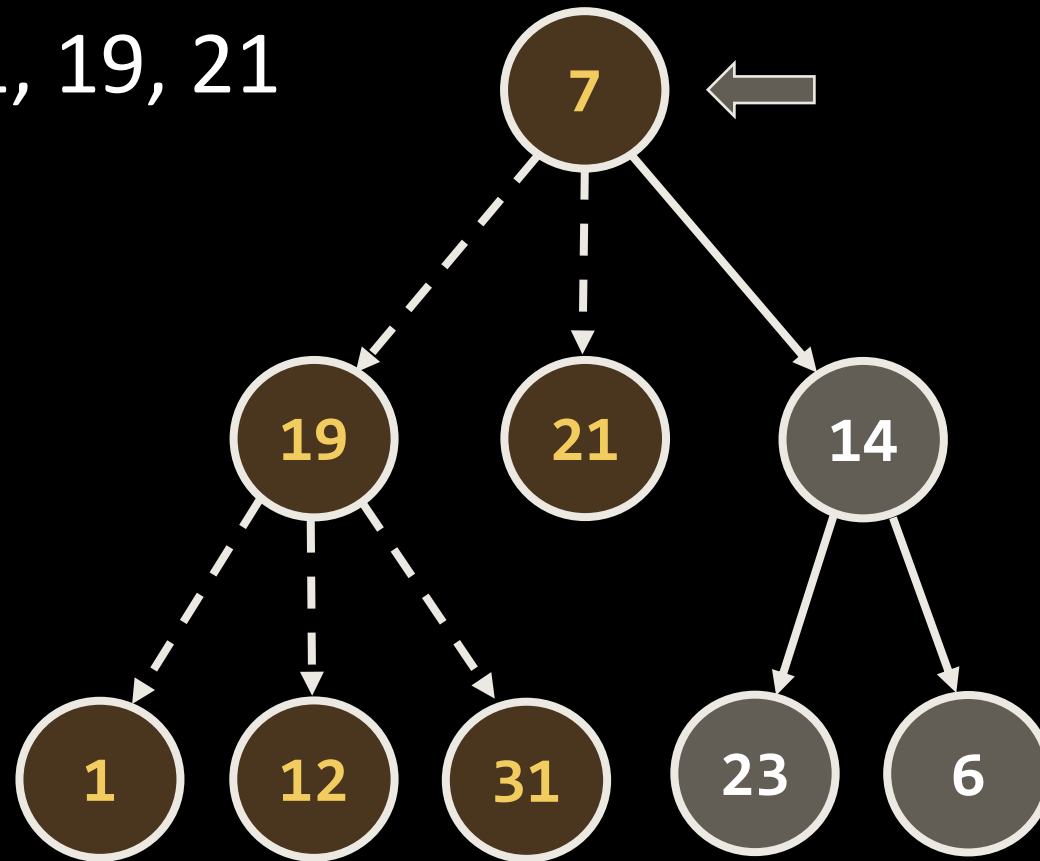
Влизаме рекурсивно  
в първия наследник

# DFS в действие (стъпка 11)

✦ Стек: 7

✦ Изход: 1, 12, 31, 19, 21

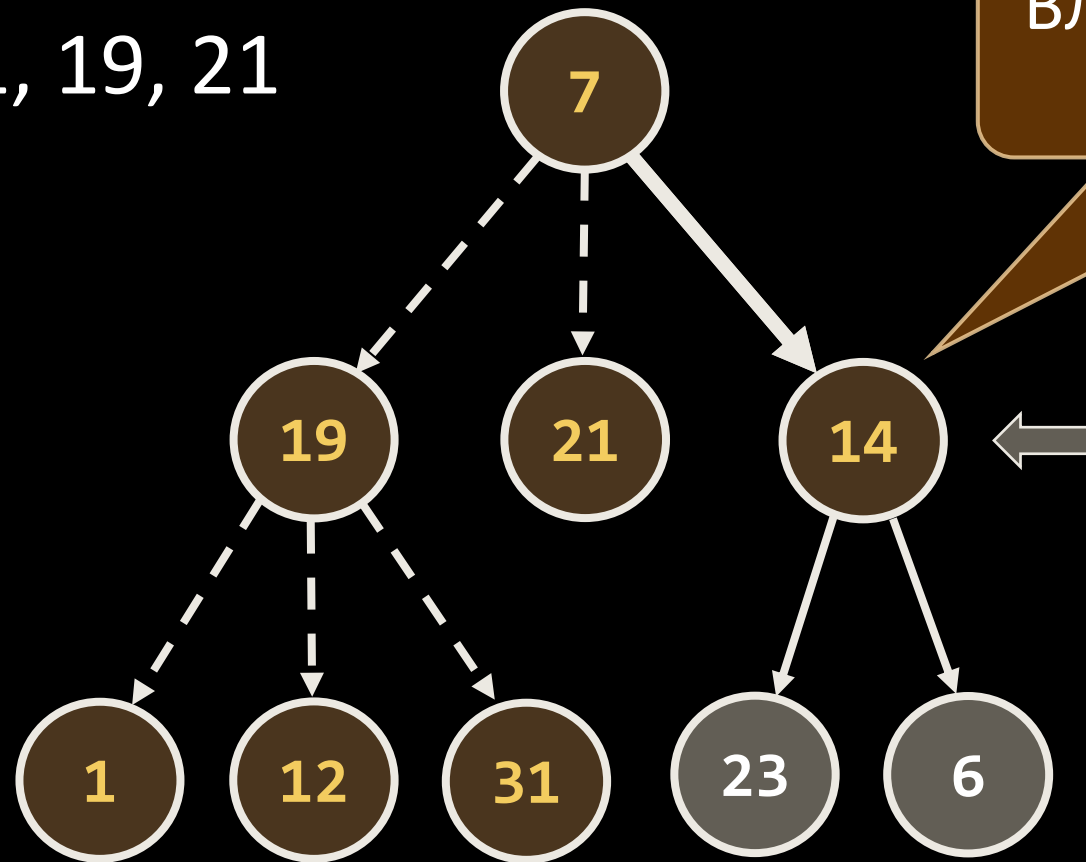
Връщаме се обратно от  
рекурсивното извикване



# DFS в действие (стъпка 12)

✦ Стек: 7, 14

✦ Изход: 1, 12, 31, 19, 21

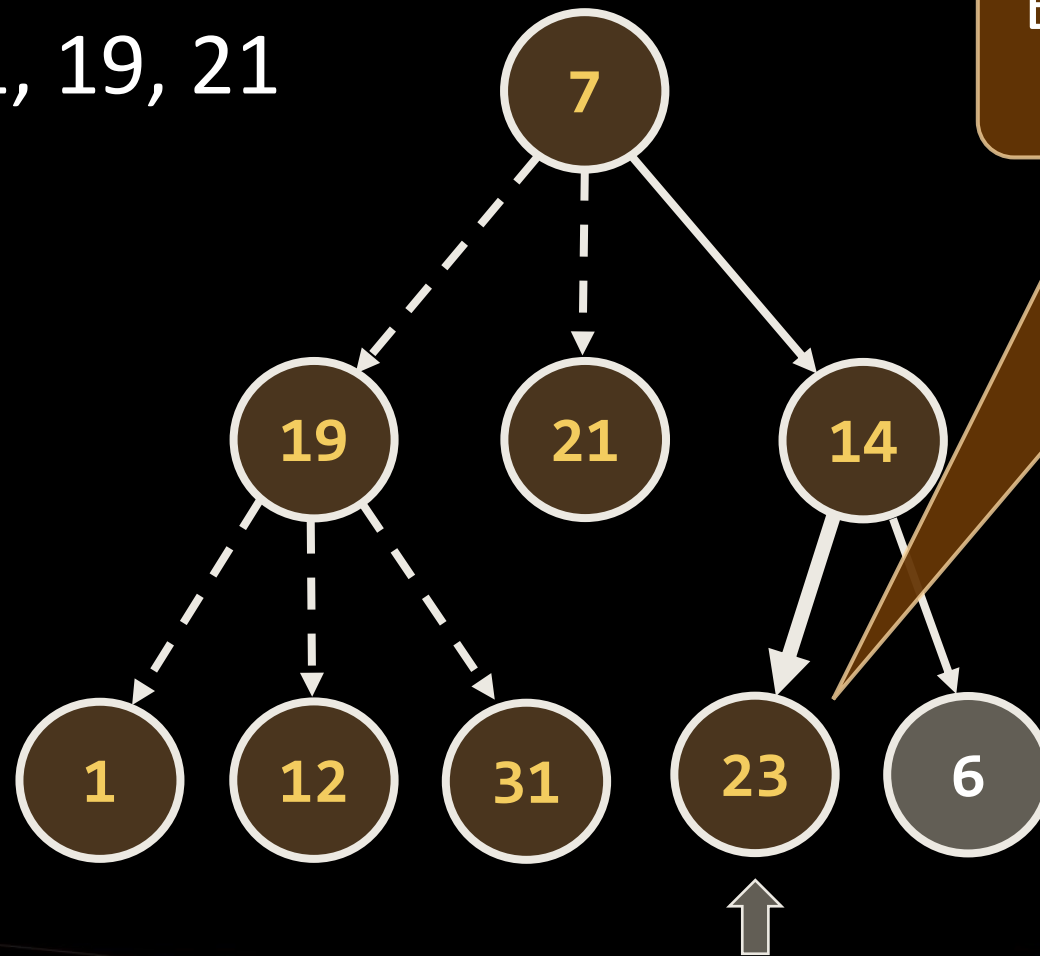


Влизаме рекурсивно в първия наследник

# DFS в действие (стъпка 13)

✦ Стек: 7, 14, 23

✦ Изход: 1, 12, 31, 19, 21

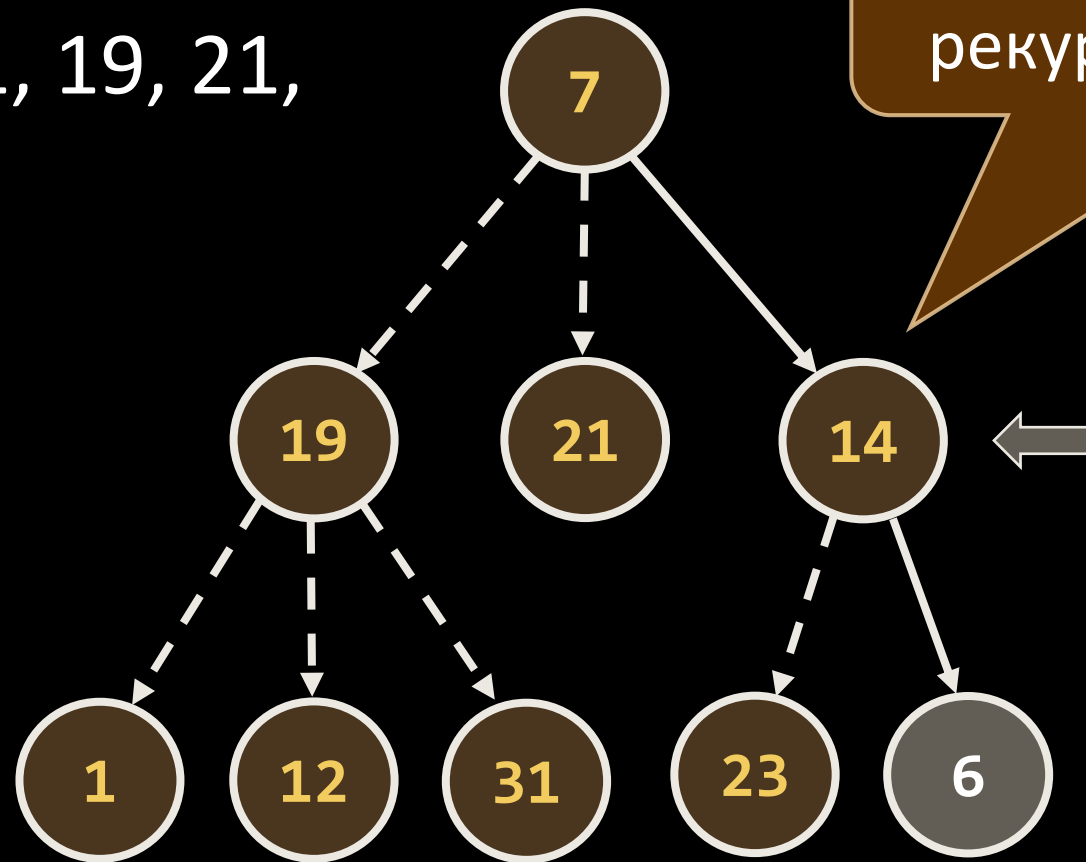


Влизаме рекурсивно  
в първия наследник

# DFS в действие (стъпка 14)

✦ Стек: 7, 14

✦ Изход: 1, 12, 31, 19, 21, 23



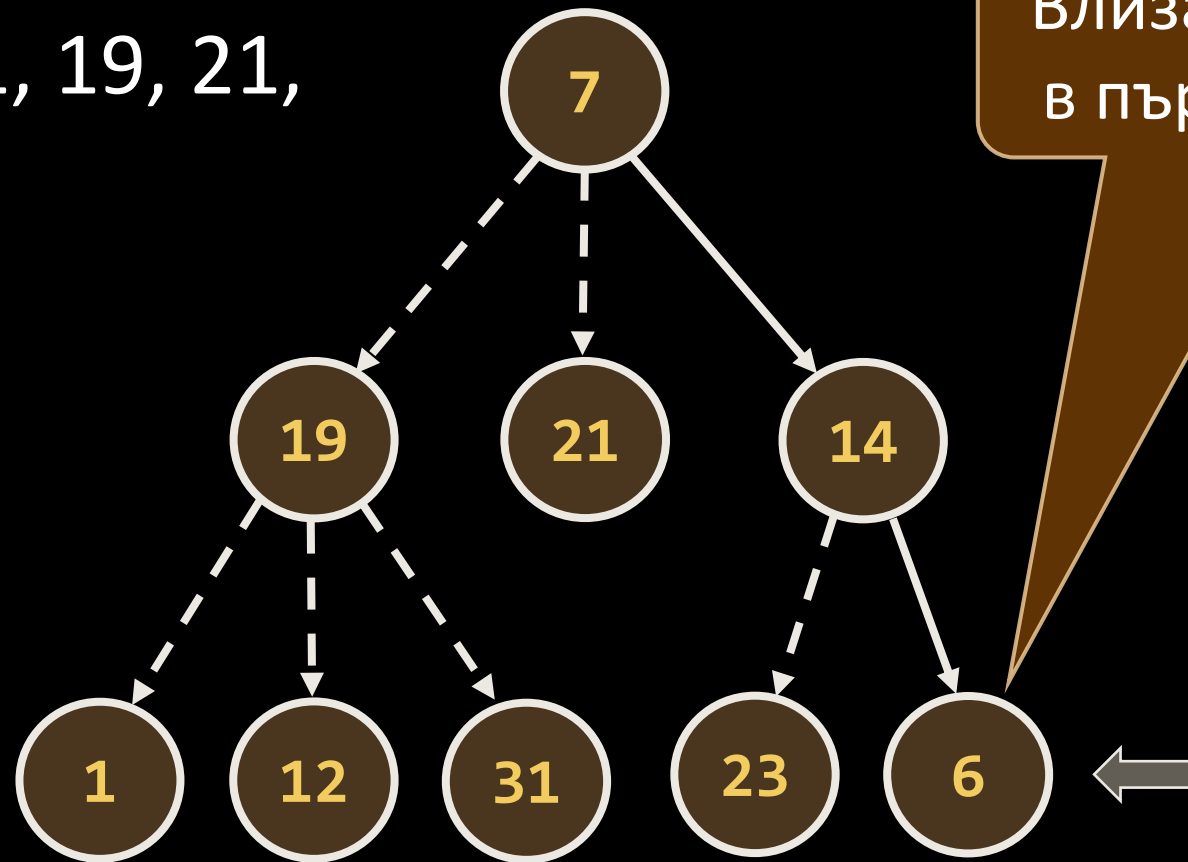
Връщаме се обратно от рекурсивното извикване



# DFS в действие (стъпка 15)

✦ Стек: 7, 14, 6

✦ Изход: 1, 12, 31, 19, 21, 23

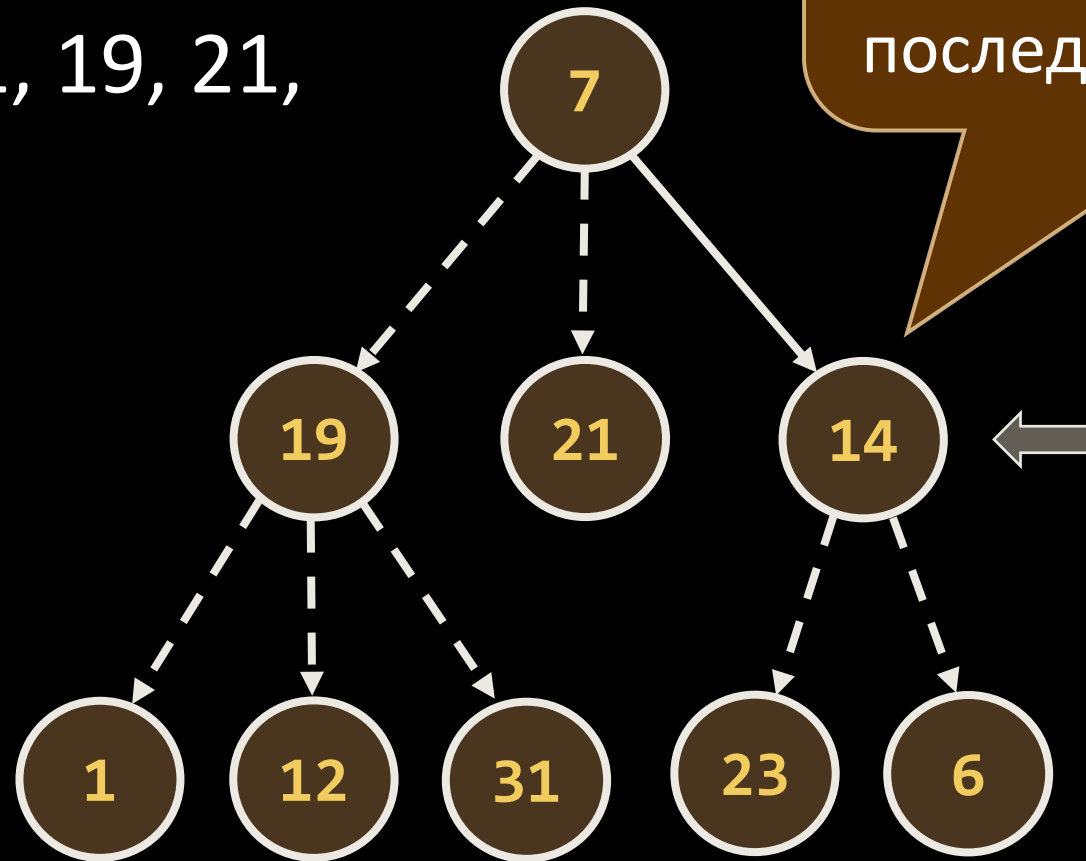


Влизаме рекурсивно  
в първия наследник

# DFS в действие (стъпка 16)

✦ Стек: 7, 14

✦ Изход: 1, 12, 31, 19, 21, 23, 6



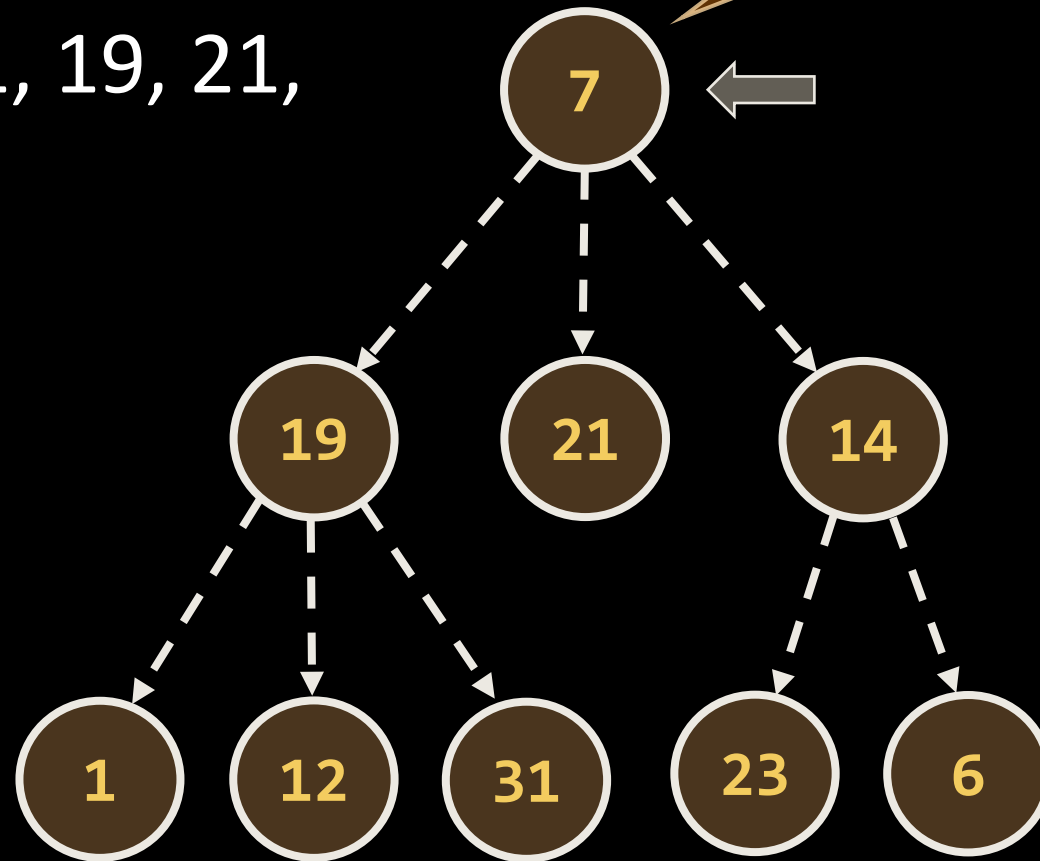
Връщаме се обратно от рекурсивното извикване и принтираме стойността на последно посещения възел

# DFS в действие (стъпка 17)

✦ Стек: 7

✦ Изход: 1, 12, 31, 19, 21,  
23, 6, 14

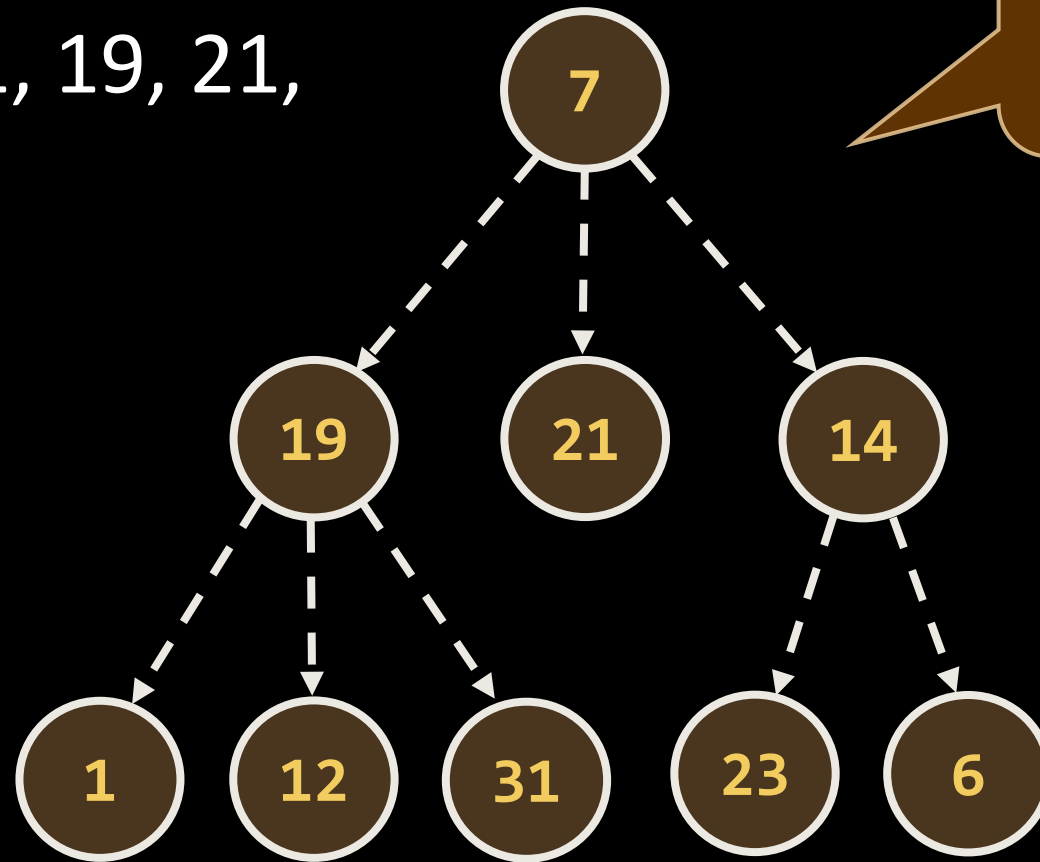
Връщаме се обратно от  
рекурсивното извикване



# DFS в действие (стъпка 18)

✦ Стек: (празен)

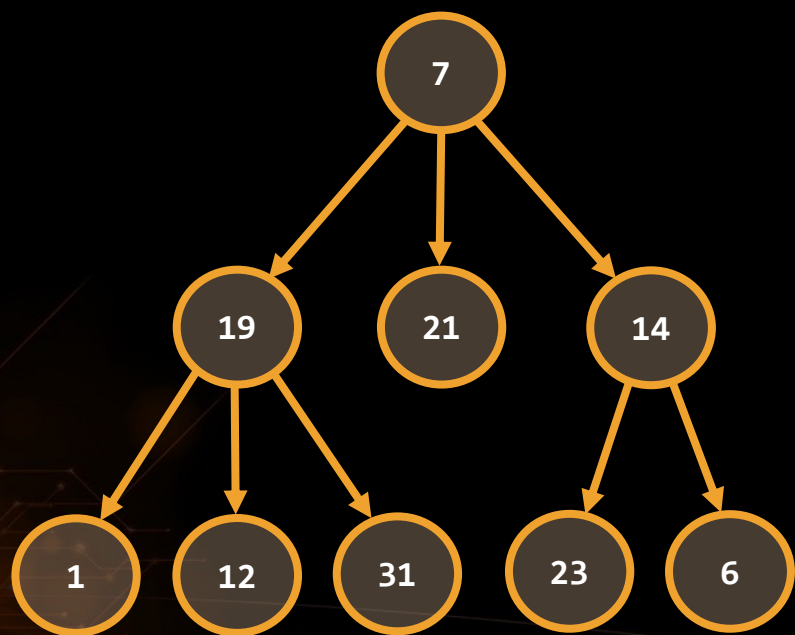
✦ Изход: 1, 12, 31, 19, 21,  
23, 6, 14, 7



Обхождането в  
дълбочина -  
приключено

# Задача: Извличане на елементи от дърво (DFS)

- ✦ Обходете дърво от тип **Tree<T>**, като дефинирате:
  - ✦ **IEnumerable<T> OrderDFS()**, който връща елементите на дървото по поредността на обхождане с DFS



1 12 31 19 21 23 6 14 7



# Задача: Извличане на елементи от дърво (DFS)

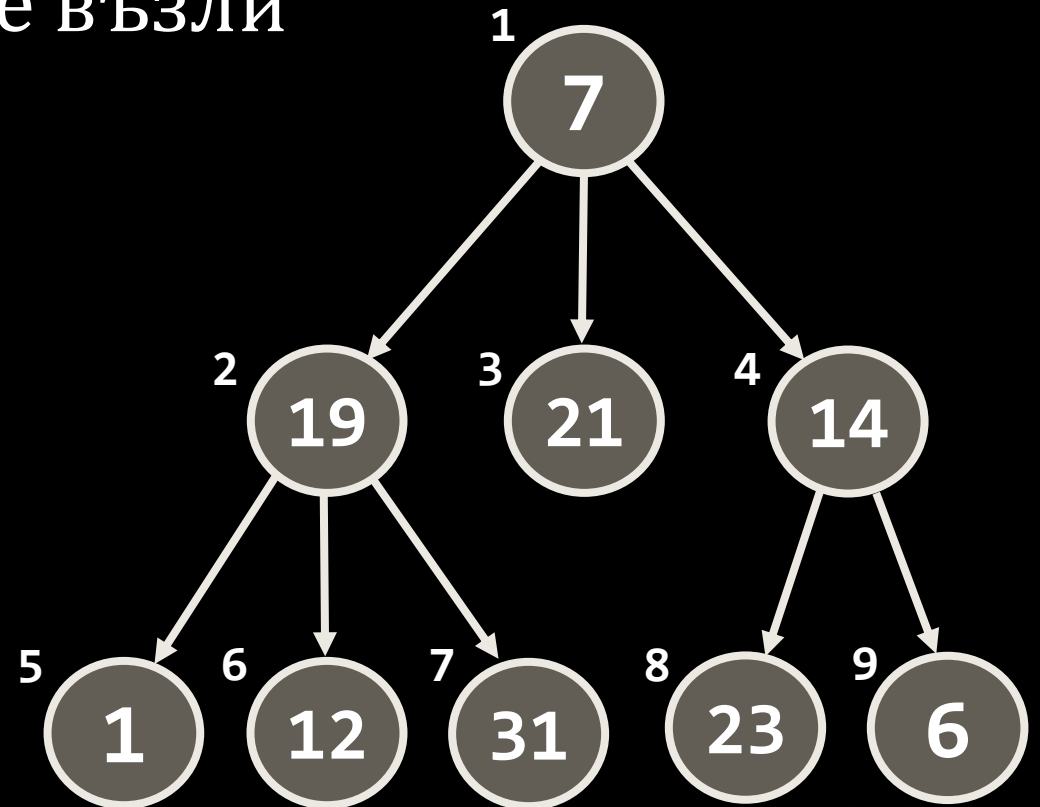
```
public IEnumerable<T> OrderDFS()  
{  
    List<T> order = new List<T>();  
    this.DFS(this, order);  
    return order;  
}  
  
private void DFS(Tree<T> tree, List<T> order)  
{  
    foreach (var child in tree.Children)  
        this.DFS(child, order);  
  
    order.Add(tree.Value);  
}
```

# Обхождане в ширина (BFS)

✦ **Обхождане в ширина (BFS)** - за всеки възел:

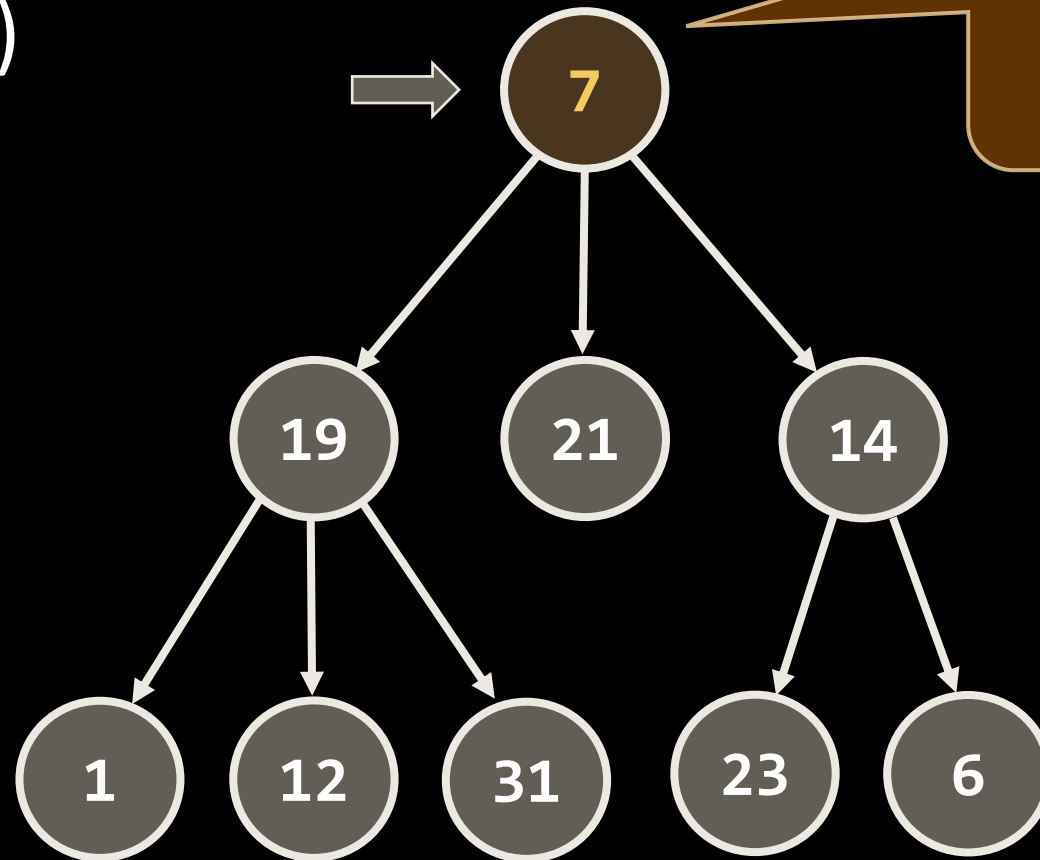
- Обработва се стойността на възела
- Посещават се всички съседните възли

```
BFS (node)
{
    queue ← node
    while queue not empty
        v ← queue
        print v
        for each child c of v
            queue ← c
}
```



# BFS в действие (стъпка 1)

- ✦ Опашка: 7
- ✦ Изход: (празен)

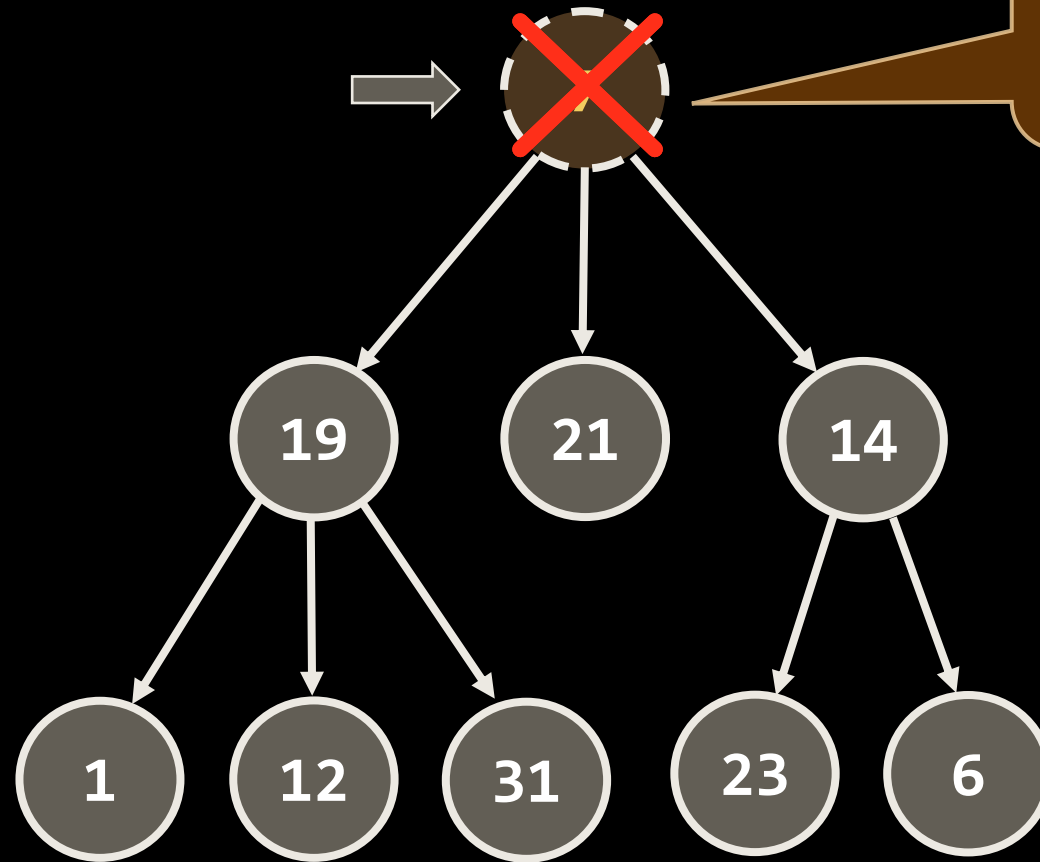


Първоначално  
добавяме корена в  
опашката

## BFS в действие (стъпка 2)

✦ Опашка: ~~7~~

✦ Изход: 7

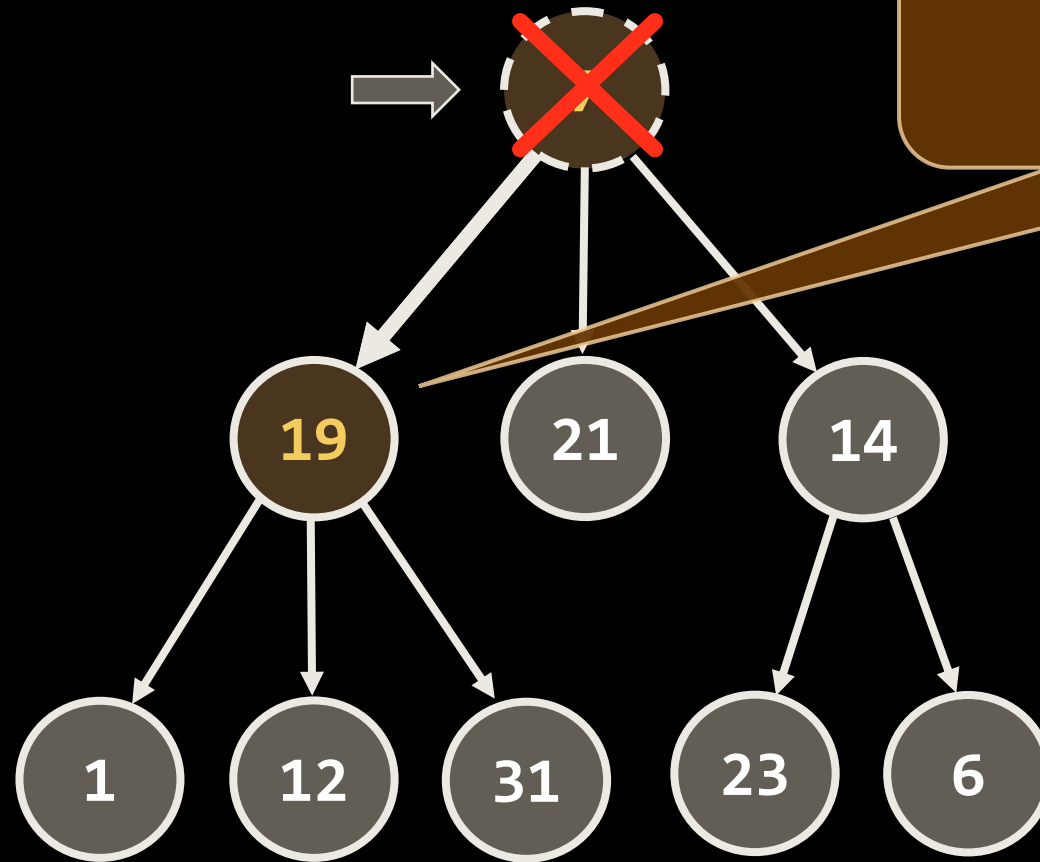


Премахваме елемент  
от опашката и го  
отпечатваме

## BFS в действие (стъпка 3)

✦ Опашка: ~~7~~, 19

✦ Изход: 7



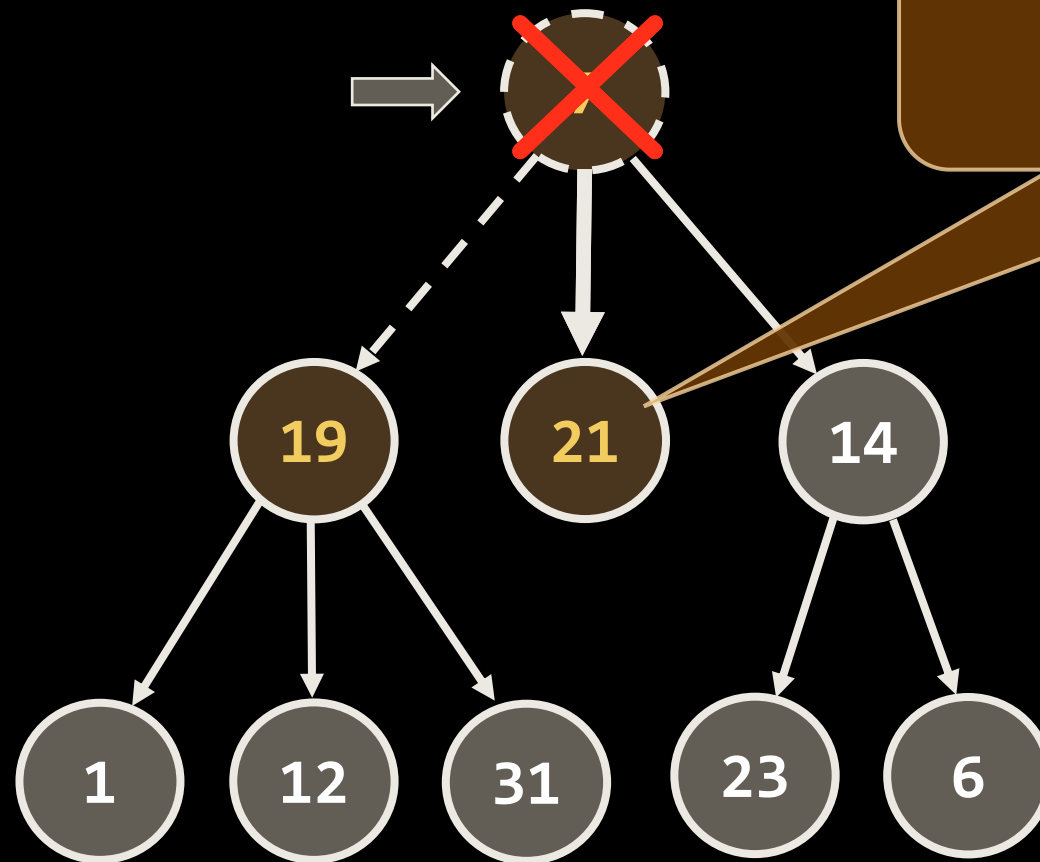
Добавяме в опашката  
всички деца на  
обхождания възел



## BFS в действие (стъпка 4)

✦ Опашка: ~~7~~, 19, 21

✦ Изход: 7

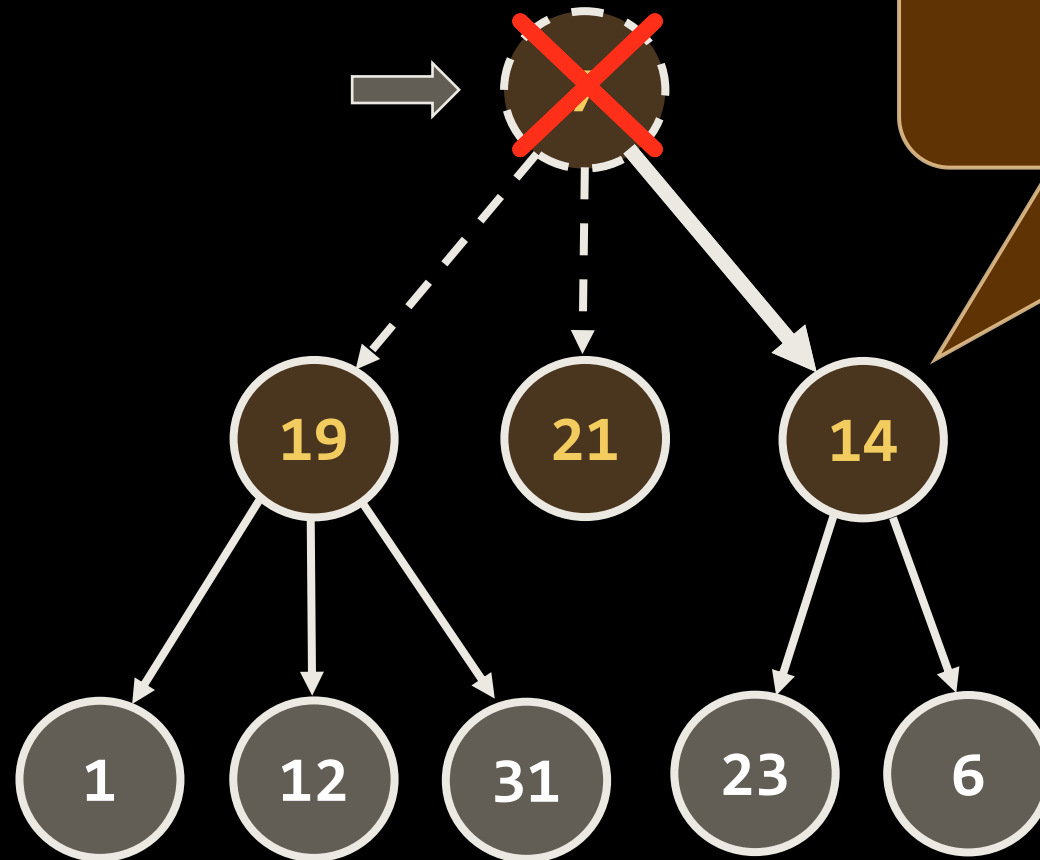


Добавяме в опашката  
всички деца на  
обхождания възел

## BFS в действие (стъпка 5)

✦ Опашка: ~~7~~, 19, 21, 14

✦ Изход: 7

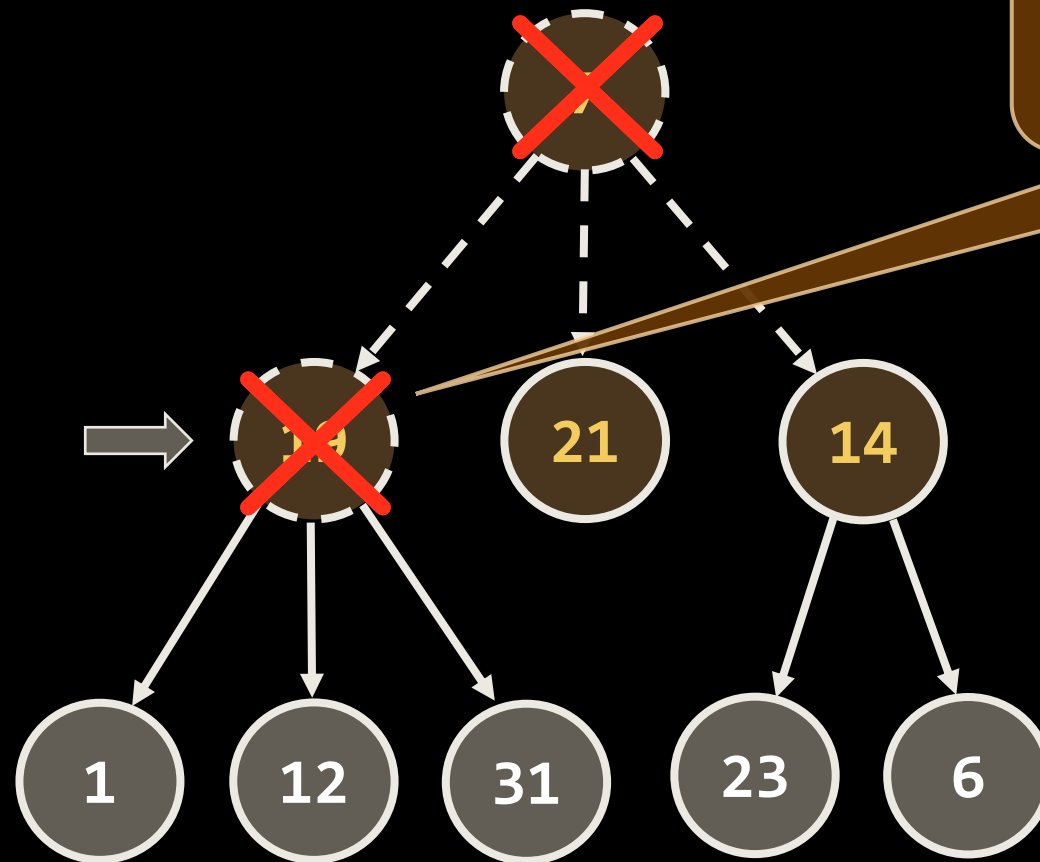


Добавяме в опашката  
всички деца на  
обхождания възел

## BFS в действие (стъпка 6)

✦ Опашка: ~~7~~, ~~19~~, 21, 14

✦ Изход: 7, 19

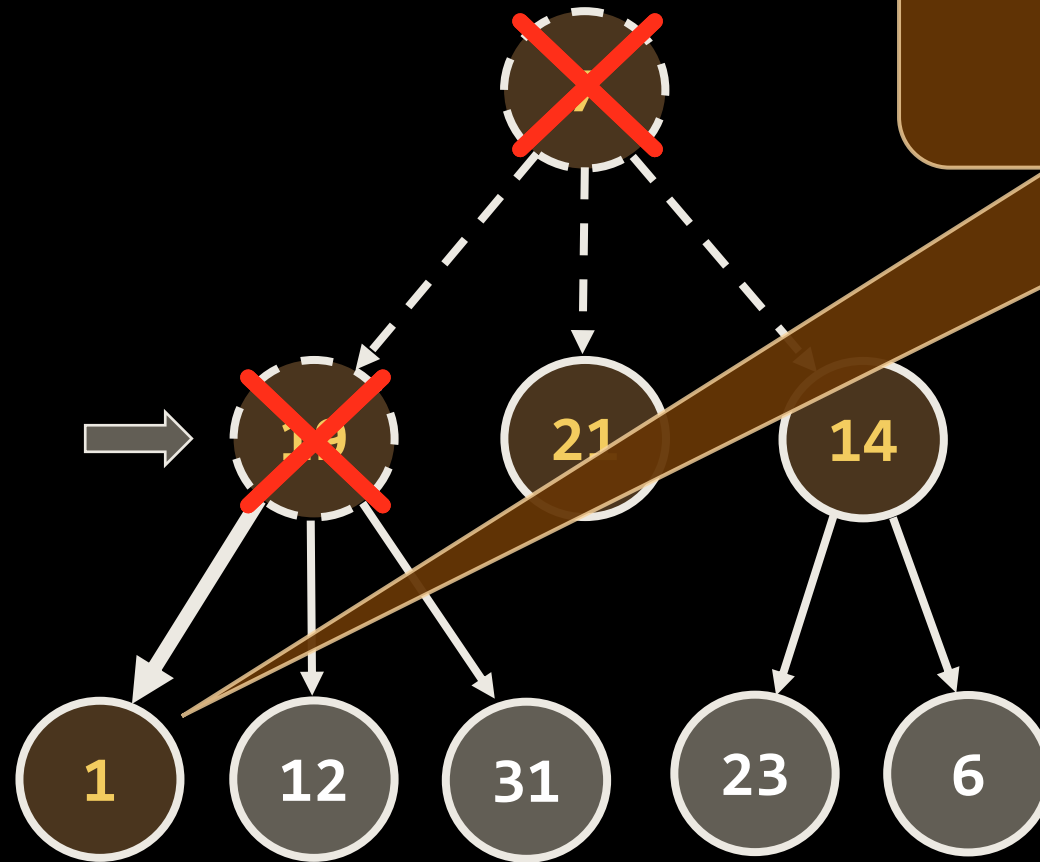


Премахваме елемент  
от опашката и го  
отпечатваме

## BFS в действие (стъпка 7)

✦ Опашка: ~~7~~, ~~19~~, 21, 14, 1

✦ Изход: 7, 19

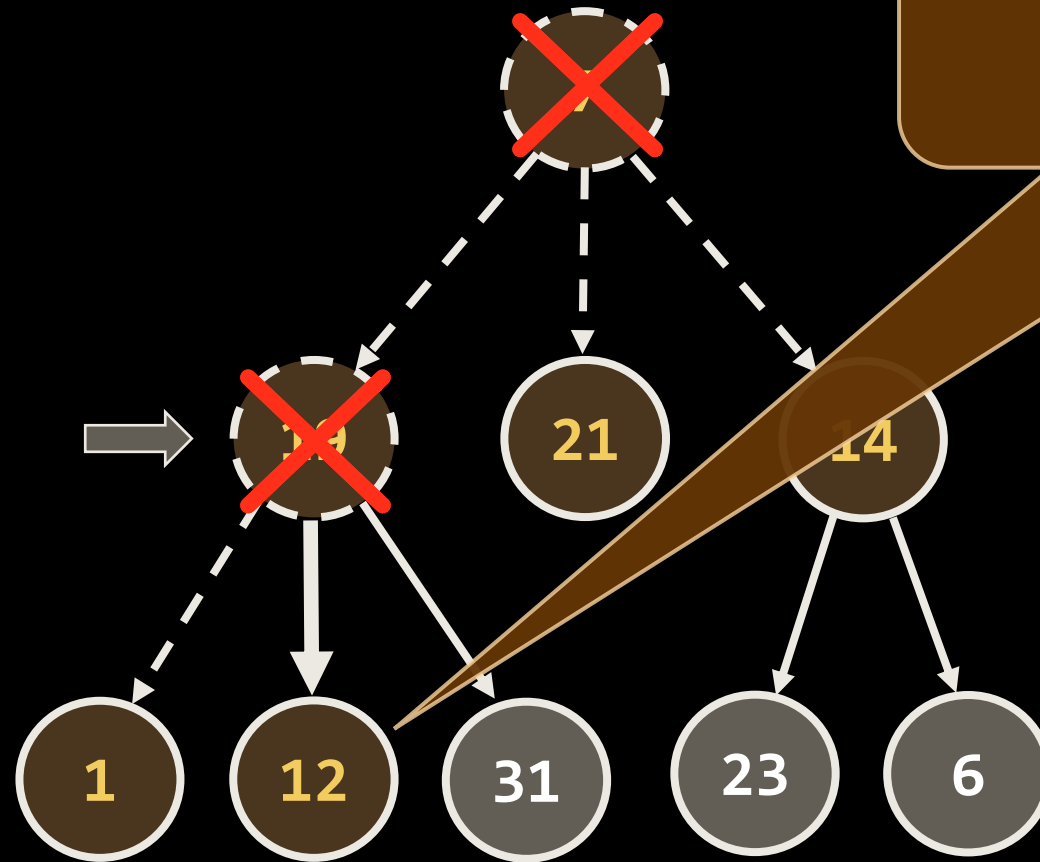


Добавяме в опашката  
всички деца на  
обхождания възел

## BFS в действие (стъпка 8)

✦ Опашка: ~~7~~, ~~19~~, 21, 14, 1, 12

✦ Изход: 7, 19



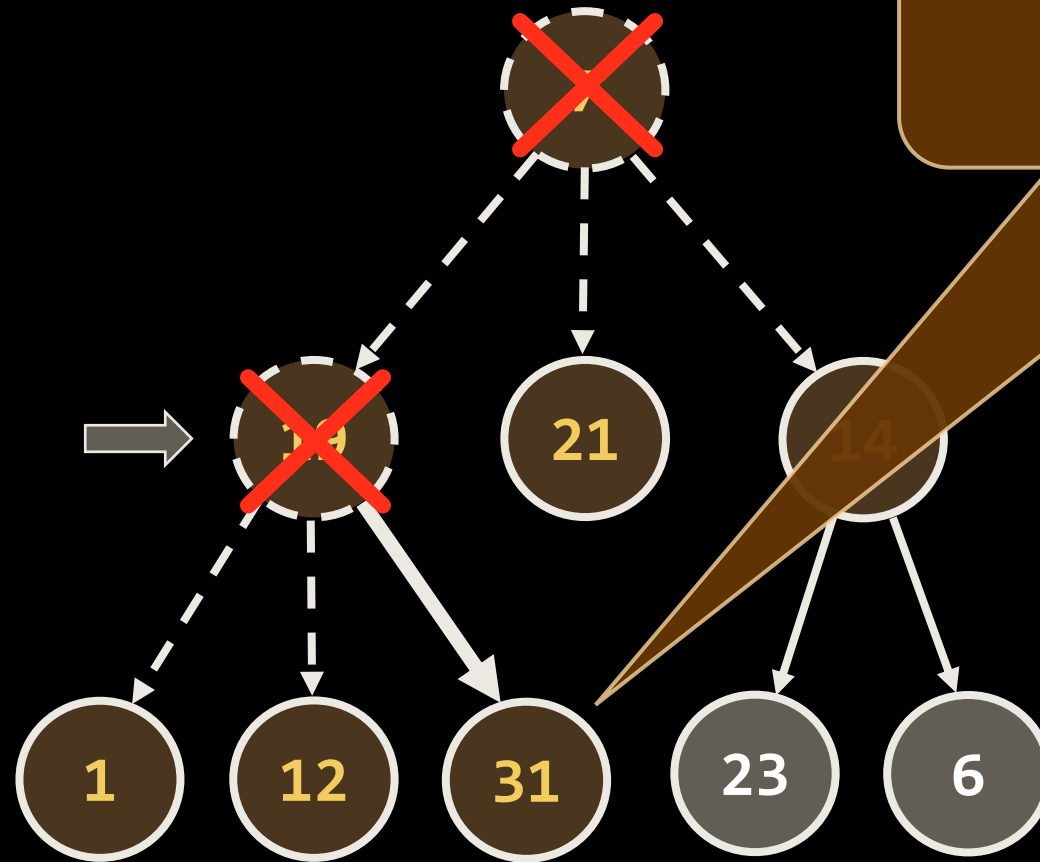
Добавяме в опашката  
всички деца на  
обхождания възел



## BFS в действие (стъпка 9)

✦ Опашка: ~~7~~, ~~19~~, 21, 14, 1, 12, 31

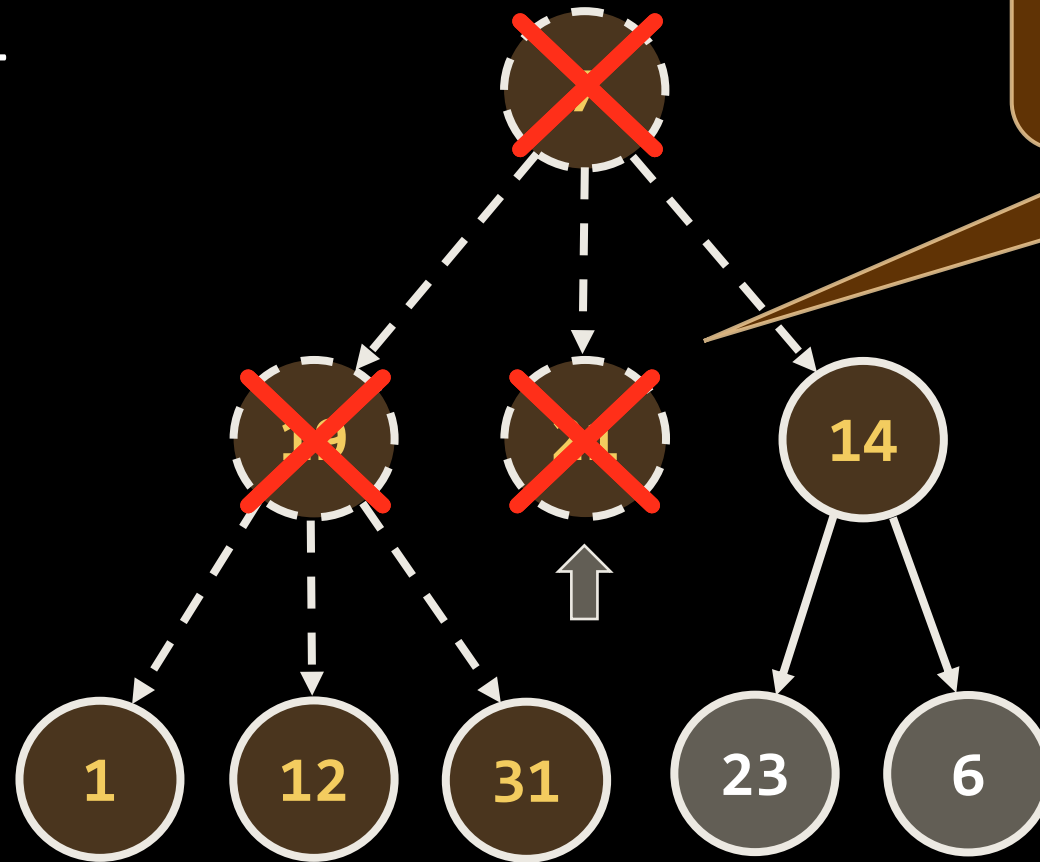
✦ Изход: 7, 19



# BFS в действие (стъпка 10)

✦ Опашка: ~~7~~, ~~19~~, ~~21~~, 14, 1, 12, 31

✦ Изход: 7, 19, 21

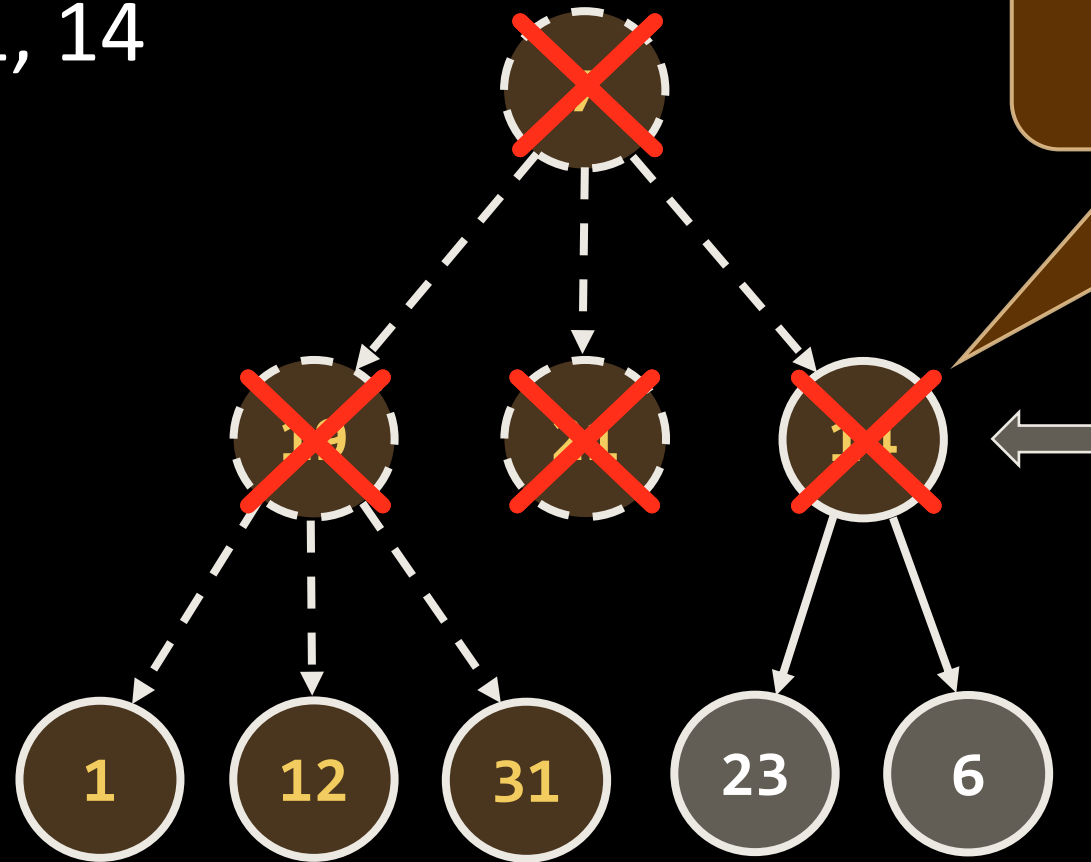


Премахваме елемент  
от опашката и го  
отпечатваме

# BFS в действие (стъпка 11)

✦ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31

✦ Изход: 7, 19, 21, 14

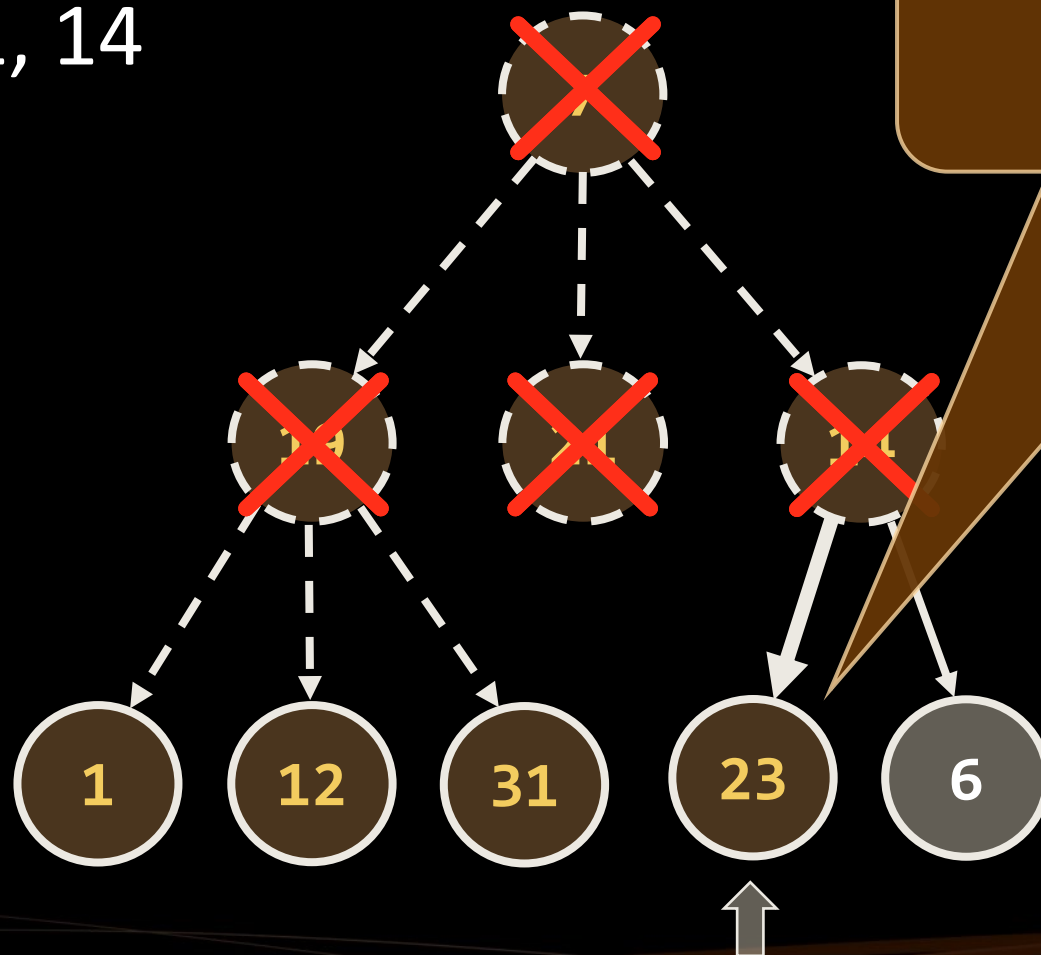


Премахваме елемент  
от опашката и го  
отпечатваме

## BFS в действие (стъпка 12)

✦ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31, 23

✦ Изход: 7, 19, 21, 14

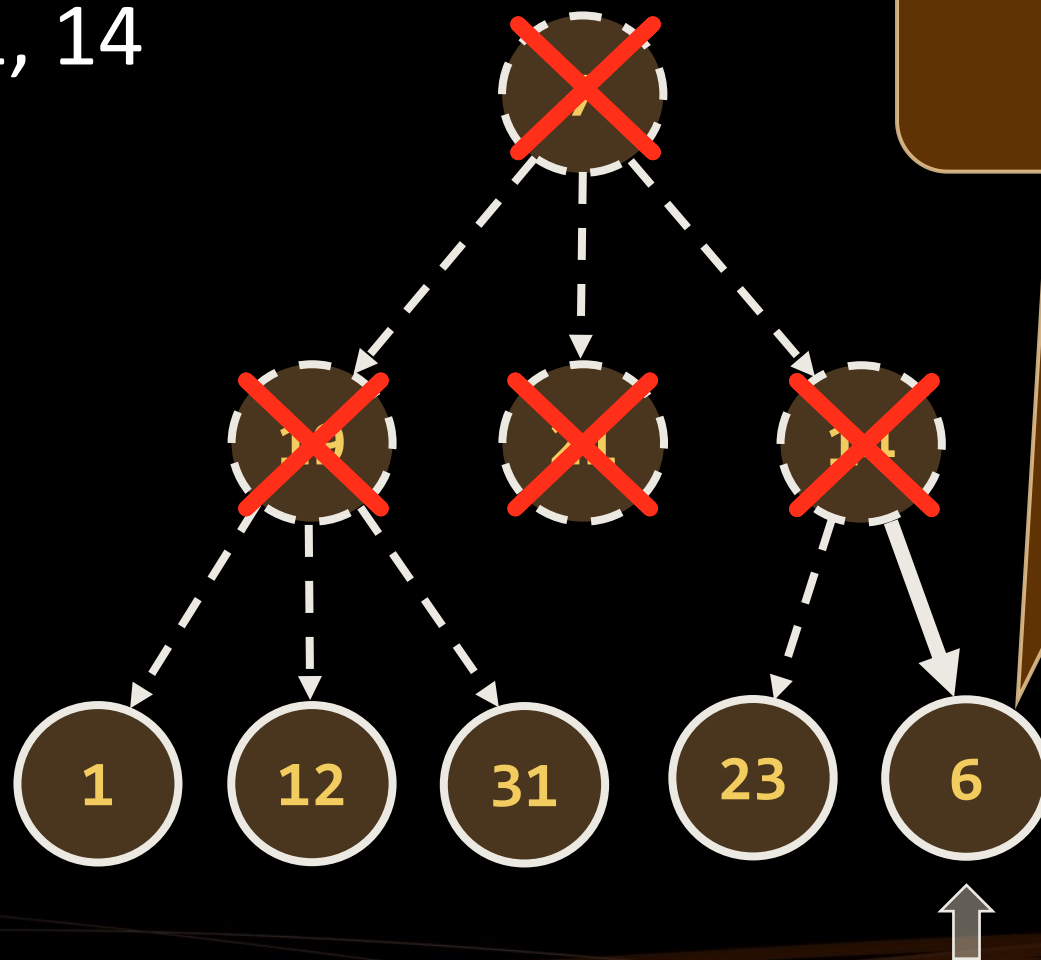


Добавяме в опашката  
всички деца на  
обхождания възел

## BFS в действие (стъпка 13)

✦ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31, 23, 6

✦ Изход: 7, 19, 21, 14



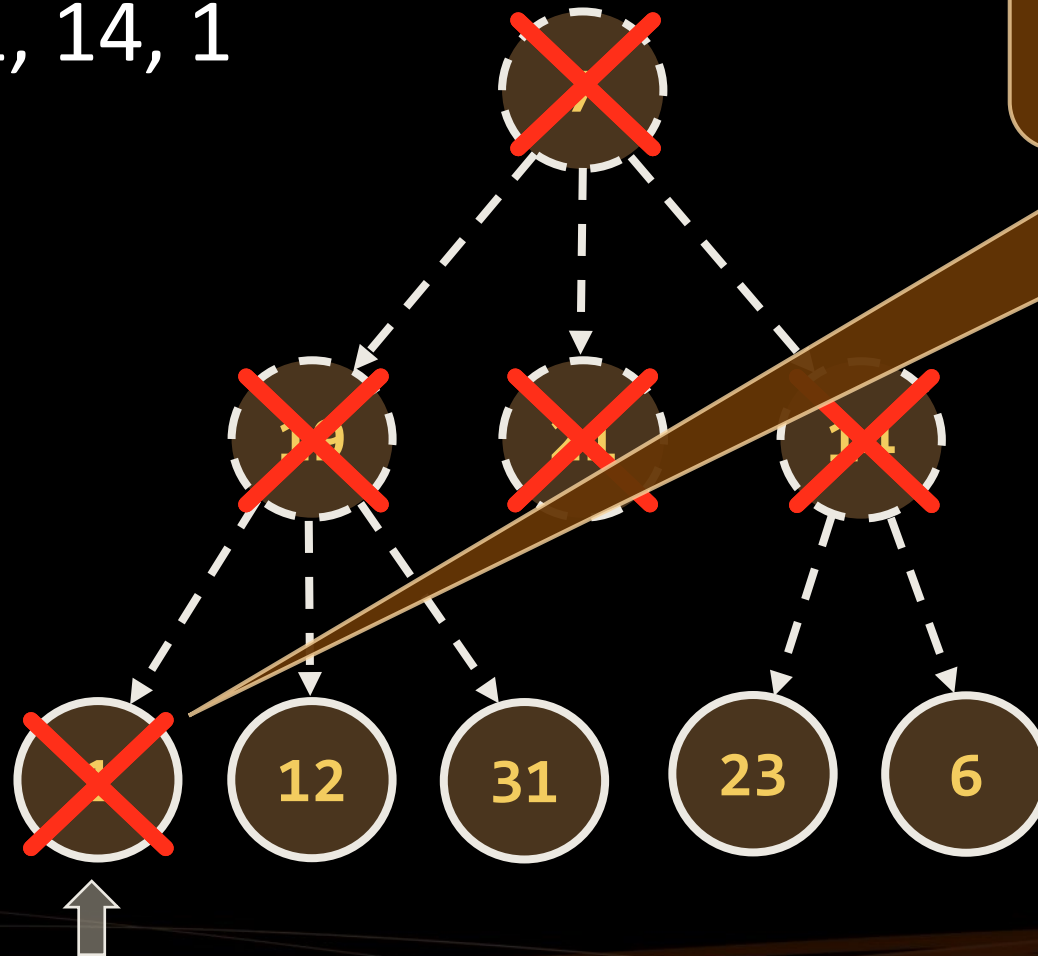
Добавяме в опашката  
всички деца на  
обхождания възел



## BFS в действие (стъпка 14)

✦ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, 12, 31, 23, 6

✦ Изход: 7, 19, 21, 14, 1

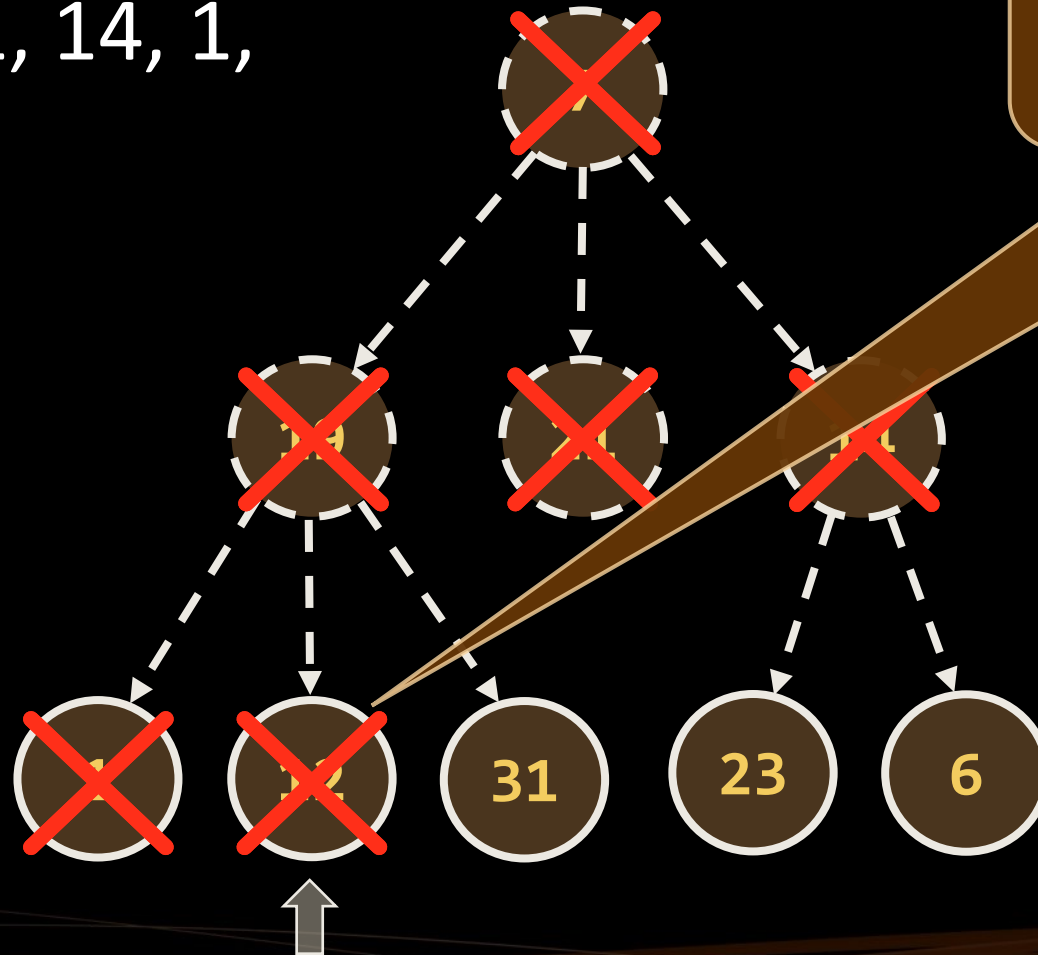


Премахваме елемент  
от опашката и го  
отпечатваме

## BFS в действие (стъпка 15)

✦ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, 31, 23, 6

✦ Изход: 7, 19, 21, 14, 1,  
12

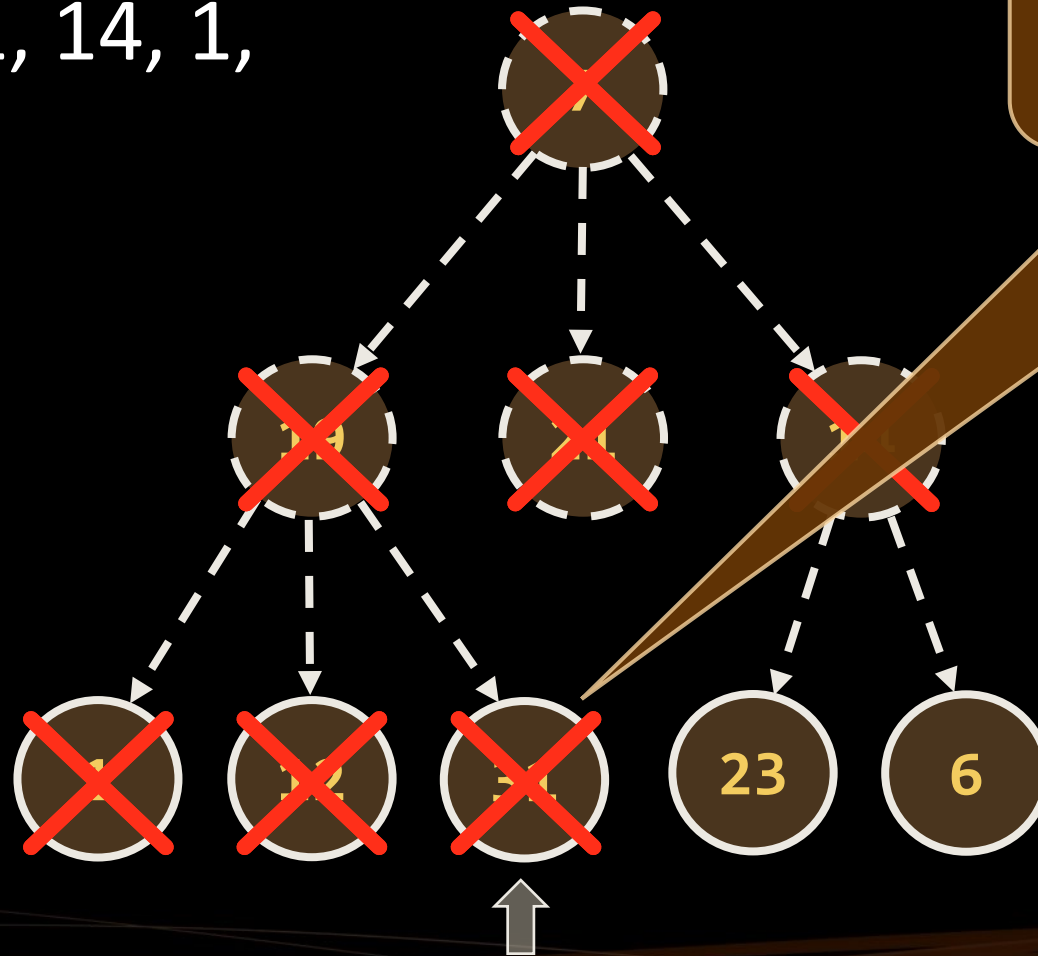


Премахваме елемент  
от опашката и го  
отпечатваме

# BFS в действие (стъпка 16)

✦ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, 23, 6

✦ Изход: 7, 19, 21, 14, 1,  
12, 31

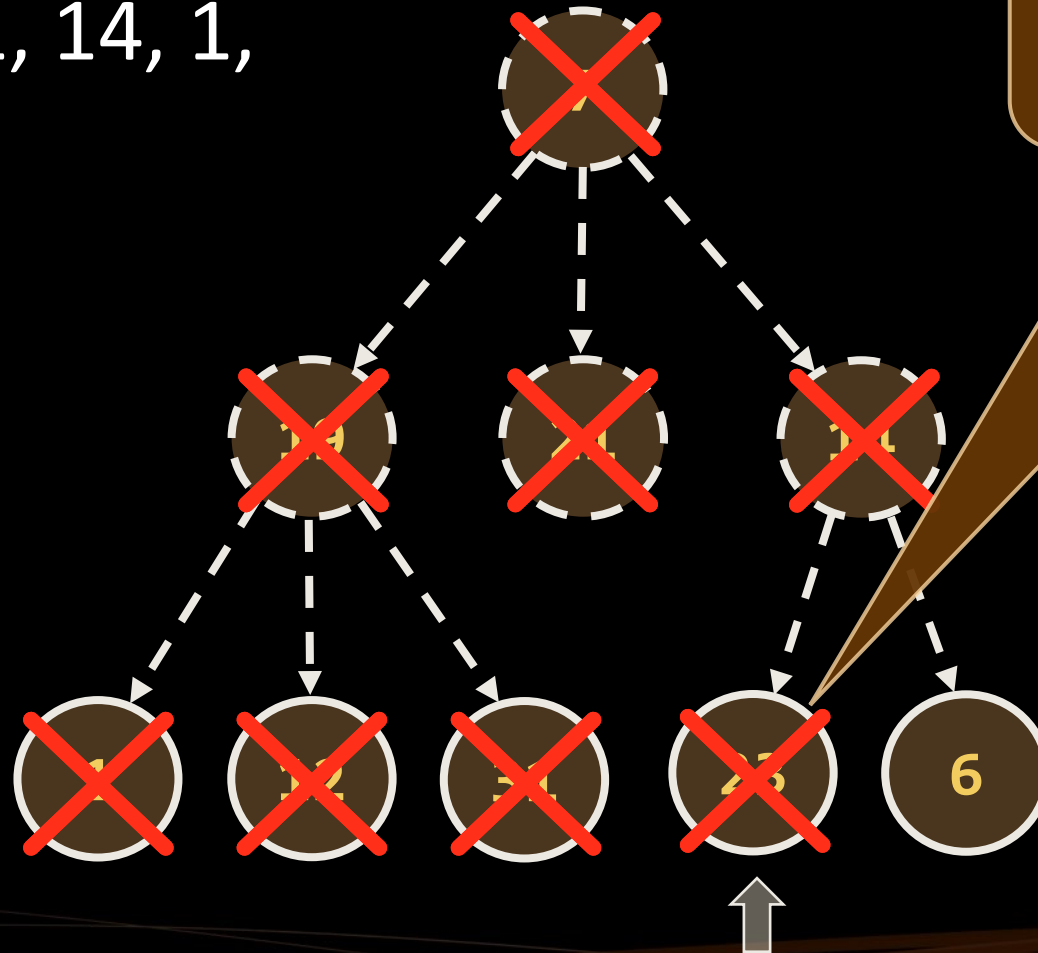


Премахваме елемент  
от опашката и го  
отпечатваме

# BFS в действие (стъпка 17)

★ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, 6

★ Изход: 7, 19, 21, 14, 1,  
12, 31, 23

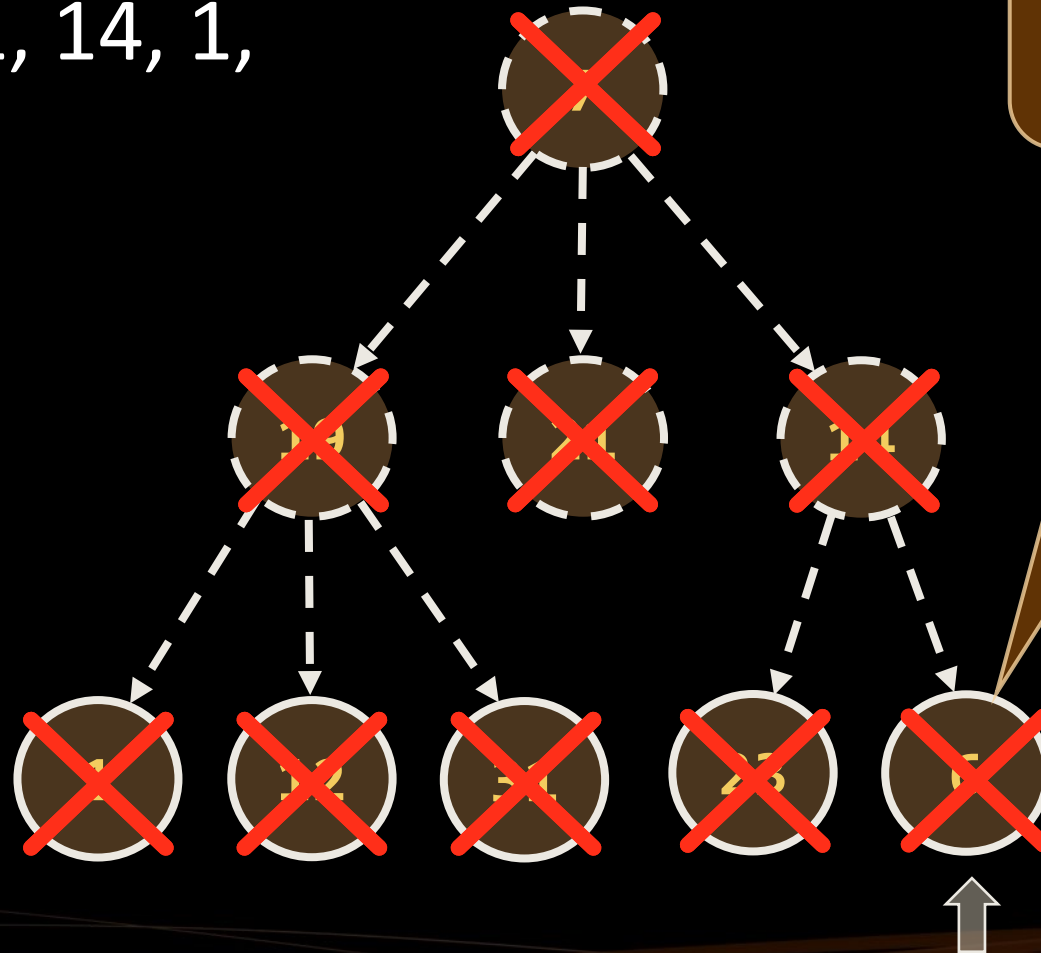


Премахваме елемент  
от опашката и го  
отпечатваме

## BFS в действие (стъпка 18)

★ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, ~~6~~

★ Изход: 7, 19, 21, 14, 1,  
12, 31, 23, 6



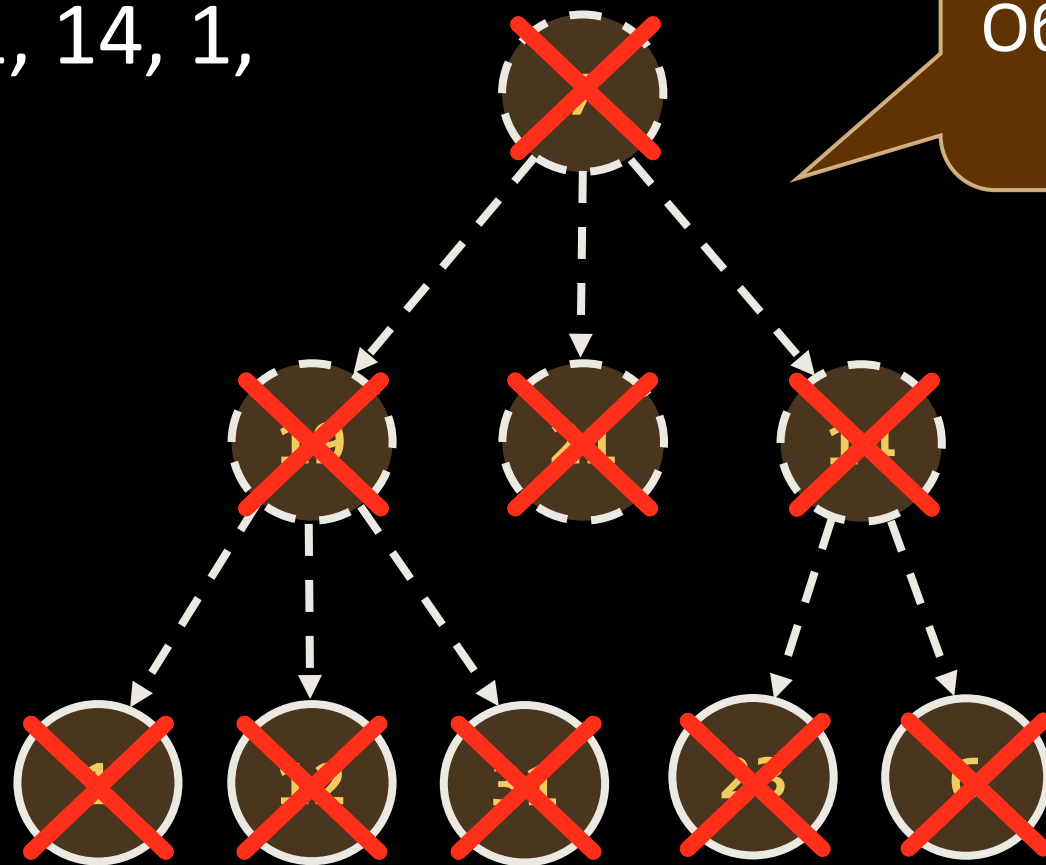
Премахваме елемент  
от опашката и го  
отпечатваме



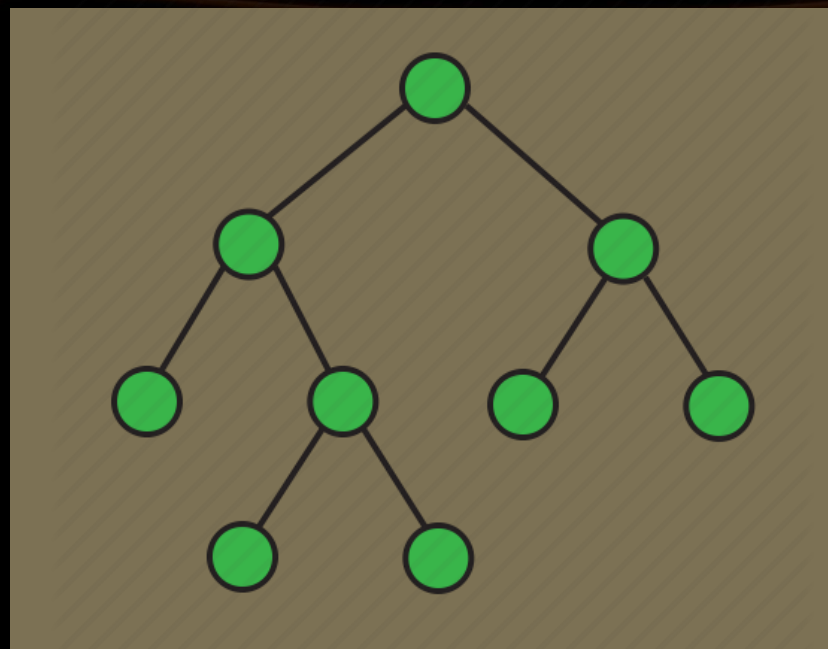
# BFS в действие (стъпка 19)

★ Опашка: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, ~~6~~

★ Изход: 7, 19, 21, 14, 1,  
12, 31, 23, 6



Опашката е празна!!!  
Обхождането в ширина  
- приключено



# Двоични дървета за търсене

## Добавяне, търсене, редакция изтриване

# Двоични дървета за търсене

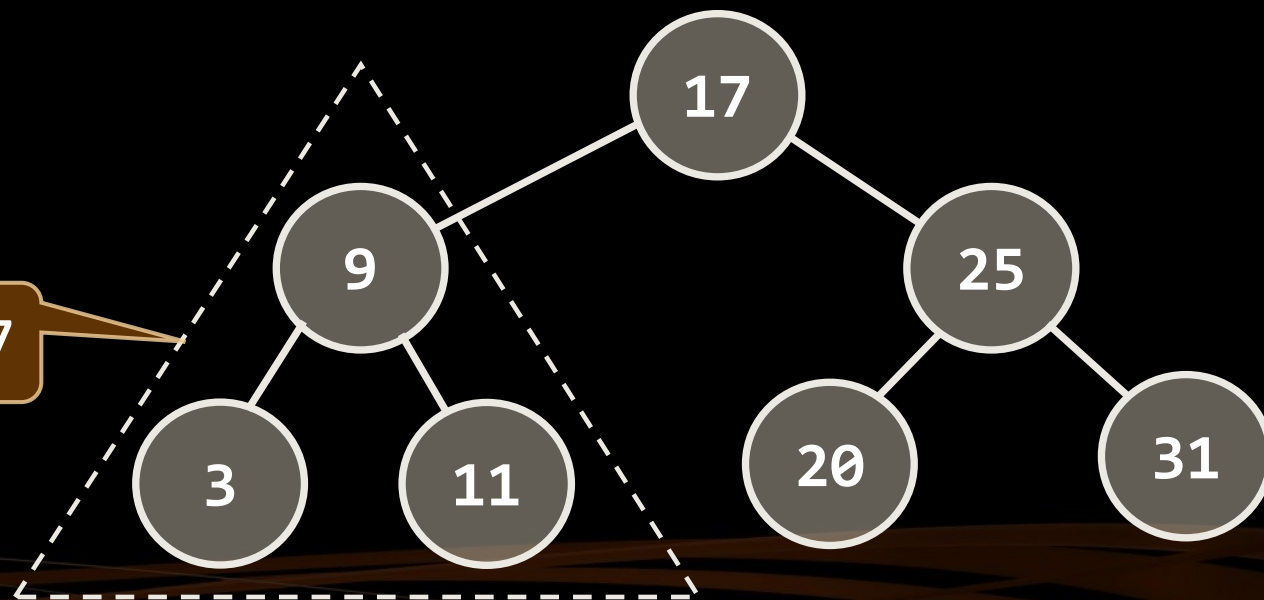
★ Двоичните дървета за търсене са **подредени**:

- За всеки възел  $x$ :

Какво става с  
елементите  
равни на  $x$ ?

- Елементите в лявото поддърво на  $x$  са по-малки от  $x$
- Елементите в дясното поддърво на  $x$  са по-големи от  $x$

Възлите са  $< 17$



# Двоично дърво за търсене - възел

```
public class BinaryTree<T>
{
    private class Node
    {
        public Node Left { get; set; }
        public Node Right { get; set; }
        public T Item { get; set; }
    }

    private Node Root { get; set; }
    public int Count { get; private set; }
    public void Add(T item)...
    public void Remove(T item)...
    public bool Contains(T item)...
}
```

# Двоично дърво за търсене - търсене

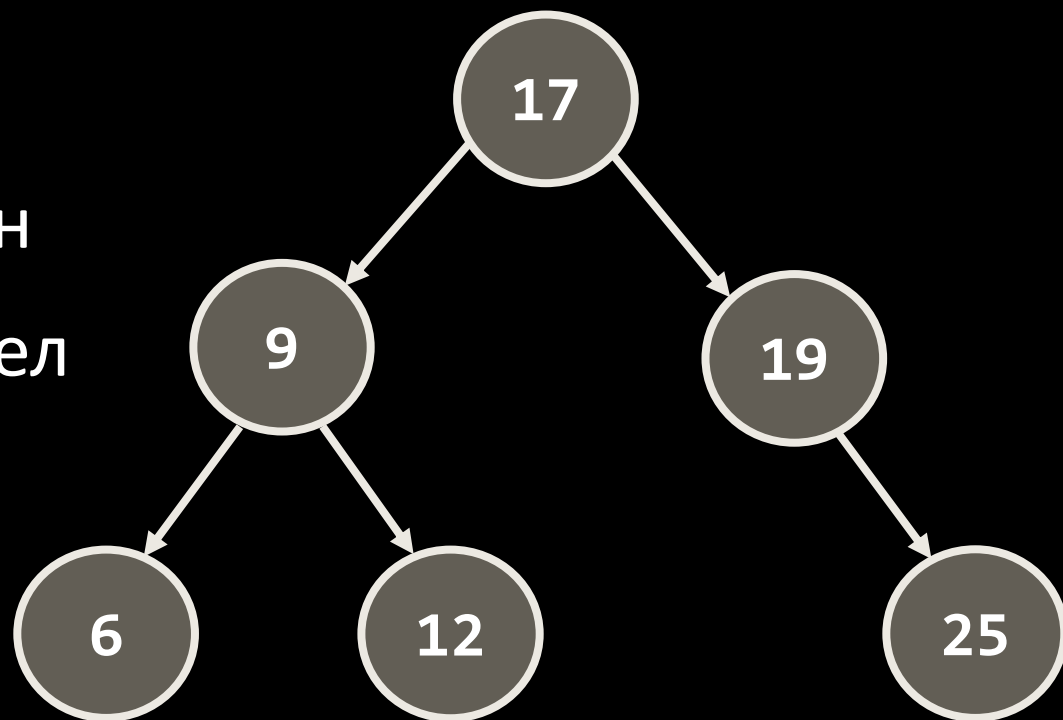
✦ Търсене на елемент  $x$  в двоично дърво за търсене

✦ if  $node \neq null$

✦ if  $x < node.value \rightarrow$  левия клон

✦ else if  $x > node.value \rightarrow$  десния клон

✦ else if  $x == node.value \rightarrow$  върни възел



Търсим 12  $\rightarrow$  17 9 12

Търсим 27  $\rightarrow$  17 19 25 null



# Своично дърво за търсене - търсене

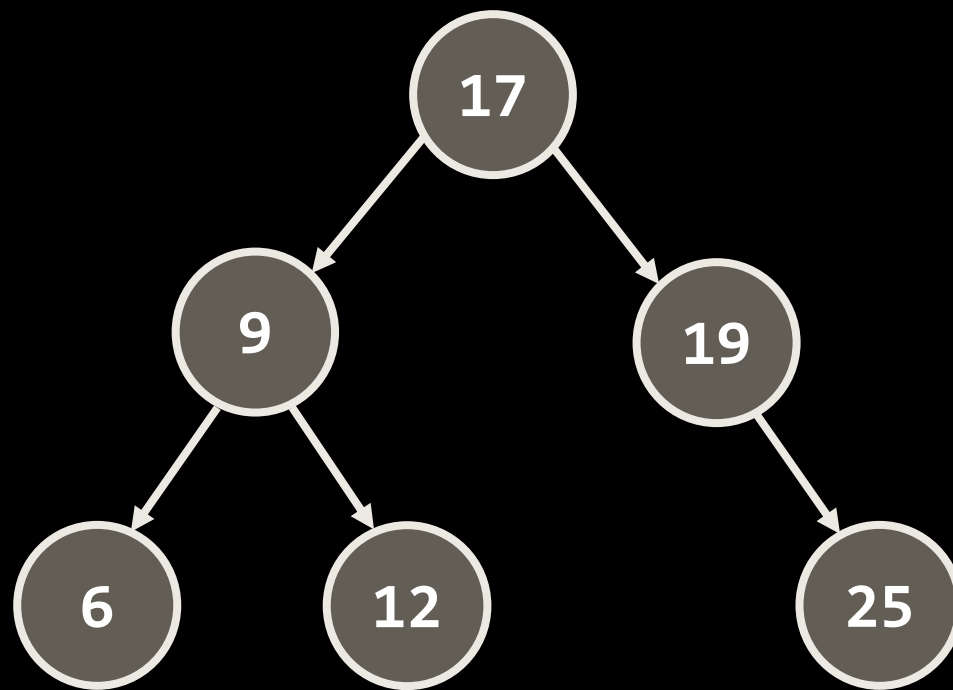
```
public bool Contains(T item)
{
    if (Root == null)
        return false;

    Node iterator = Root;
    while (true)
    {
        if (iterator == null)
            return false;
        else if (iterator.Item.CompareTo(item) == 0)
            return true;
        else if (iterator.Item.CompareTo(item) > 0)
            iterator = iterator.Left;
        else if (iterator.Item.CompareTo(item) < 0)
            iterator = iterator.Right;
    }
}
```

# Двоично дърво за търсене - добавяне

## ✦ Добавяне на елемент $x$ в двоично дърво за търсене

- if  $node == null \rightarrow$  добави  $x$
- else if  $x < node.value \rightarrow$  ляв клон
- else if  $x > node.value \rightarrow$  десен клон
- ✦ else  $\rightarrow$  възела съществува



Добавяне 12    17 9 12 return

Добавяне 27    17 19 25 null (добавяне)

# Двоично дърво за търсене - добавяне

```
public void Add(T item)
{
    Node node = new Node();
    node.Item = item;

    if (Root == null)
    {
        Root = node;
        return;
    }

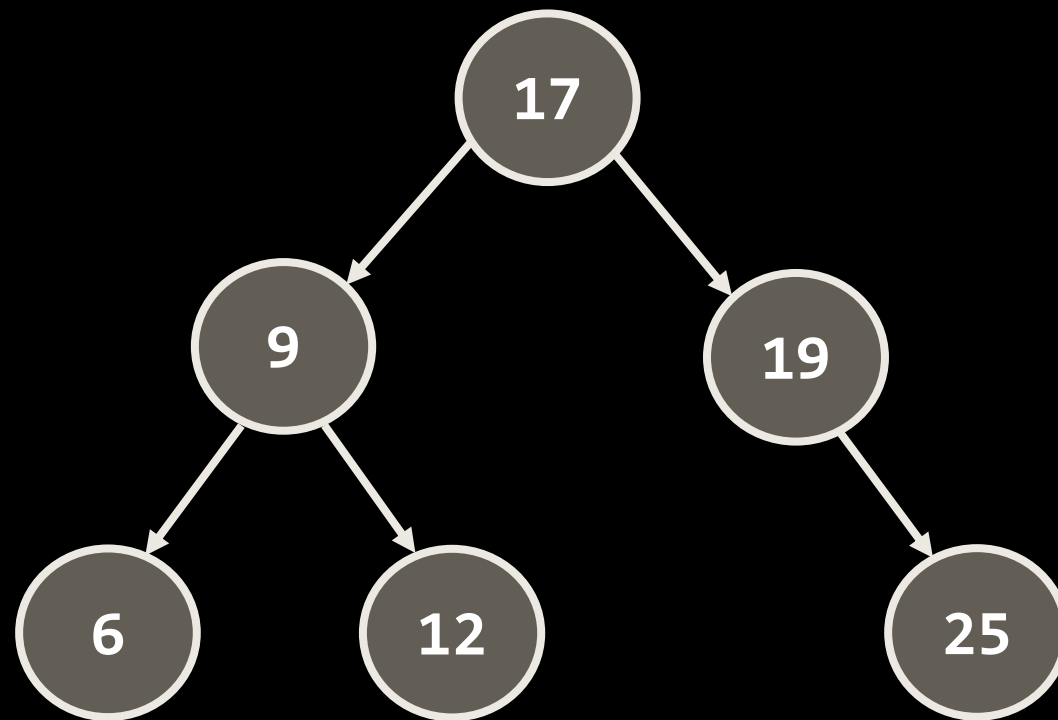
    Node iterator = Root;
    while (true)
    {
        if (iterator.Left != null && iterator.Item.CompareTo(item) >= 0)
            iterator = iterator.Left;
        else if (iterator.Right != null && iterator.Item.CompareTo(item) < 0)
            iterator = iterator.Right;
        else
            break;
    }

    if (iterator.Item.CompareTo(item) >= 0)
        iterator.Left = node;
    else if (iterator.Item.CompareTo(item) < 0)
        iterator.Right = node;
}
```

# Двоично дърво за търсене - премахване

## ✦ Премахване на елемент $x$ в двоично дърво за търсене

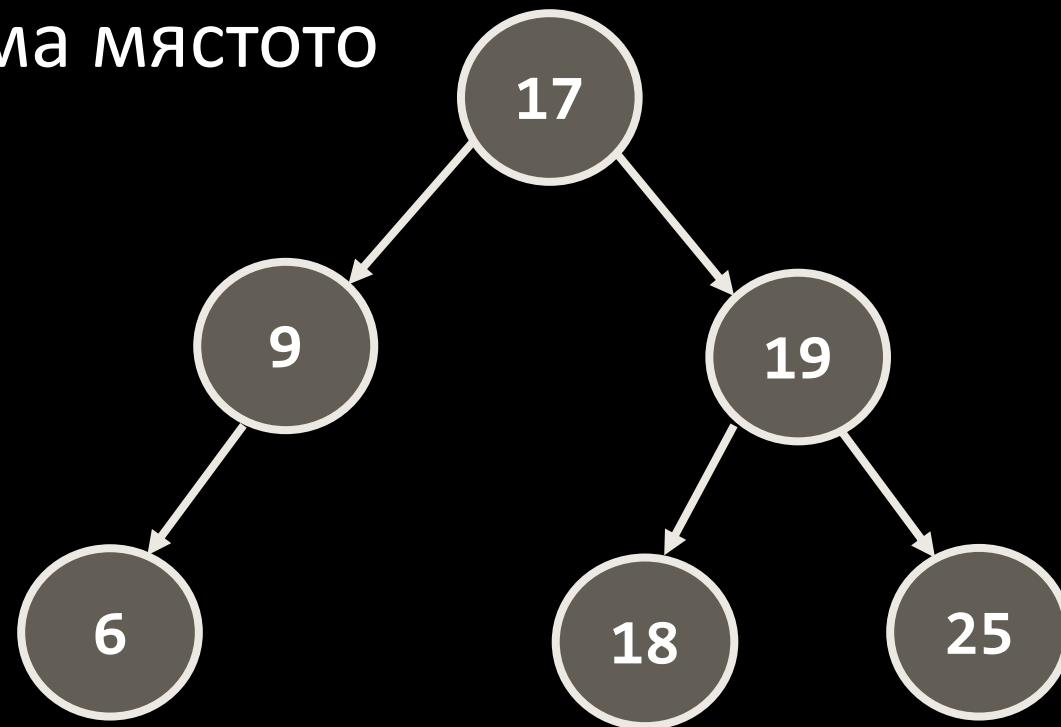
- ✦ if  $node == null \rightarrow$  изход
- ✦ else if  $x$  is leaf  $\rightarrow$  премахни
- ✦ else if  $x$  is not leaf  $\rightarrow$  подмени
  - ✦ (3 случая при подмяна на възел)



# Двоично дърво за търсене - премахване

- ✦ Премахване на елемент, който няма дясно поддърво
  - ✦ Намираме елемента за премахване
  - ✦ Корена на лявото поддърво заема мястото на премахнатия елемент

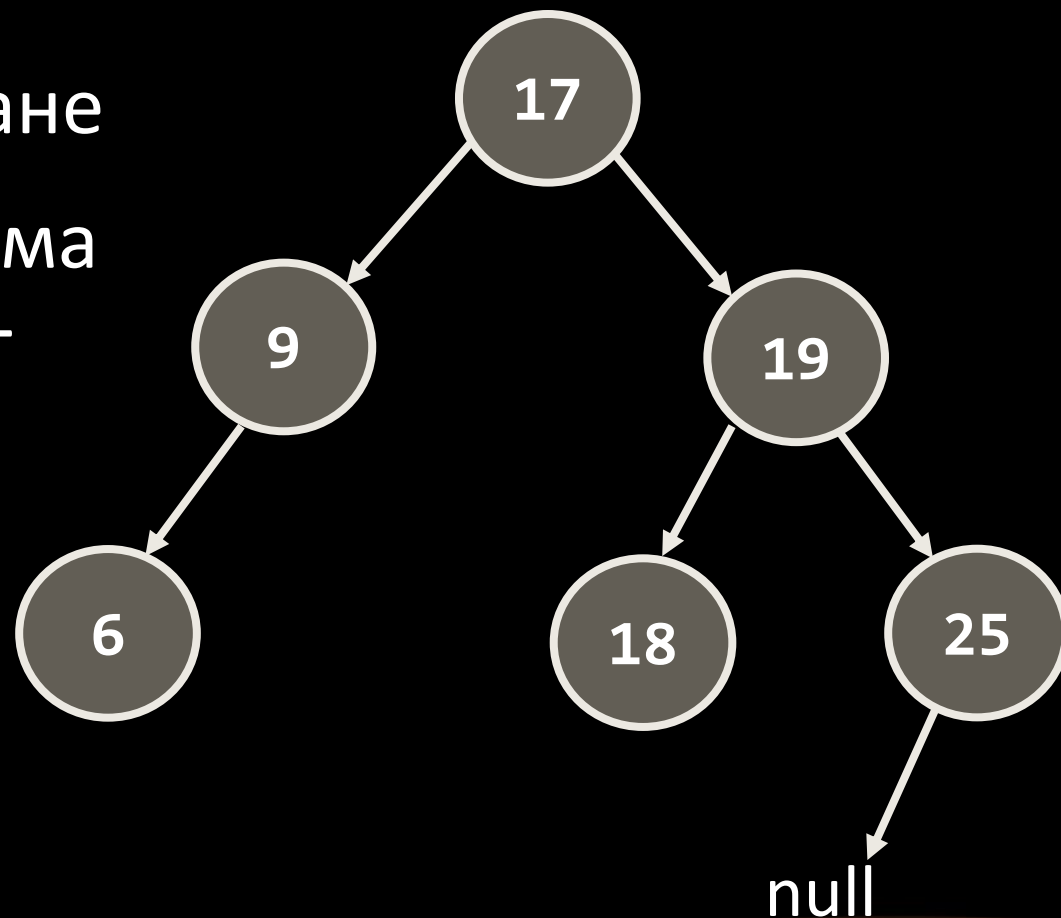
✦ Example: Delete 9





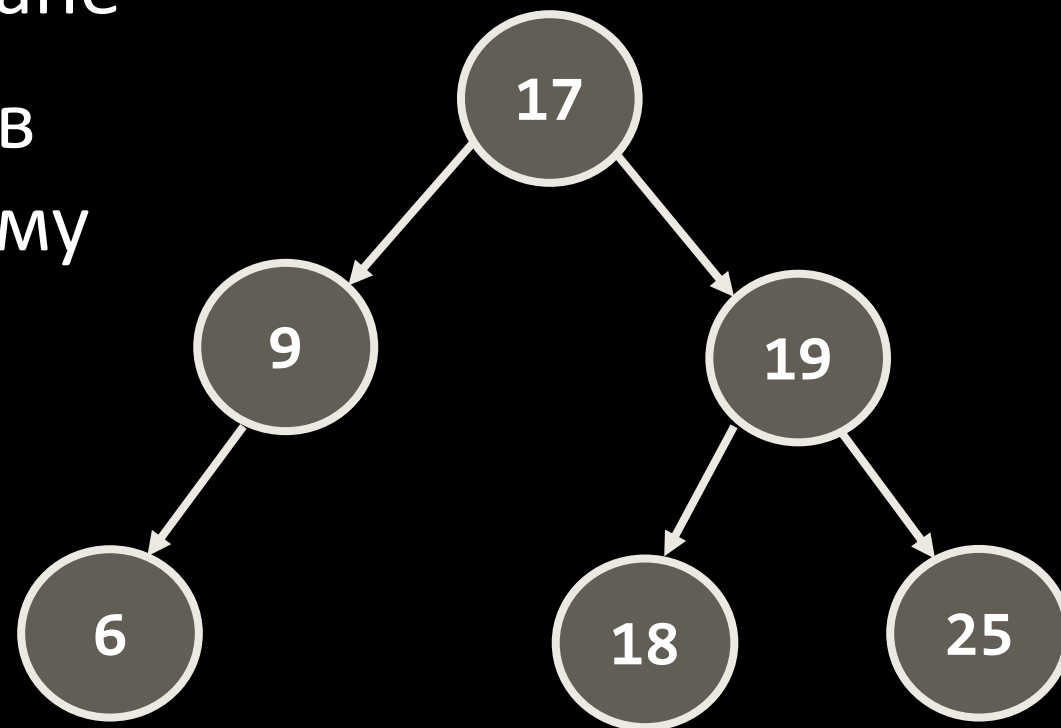
# Двоично дърво за търсене - премахване

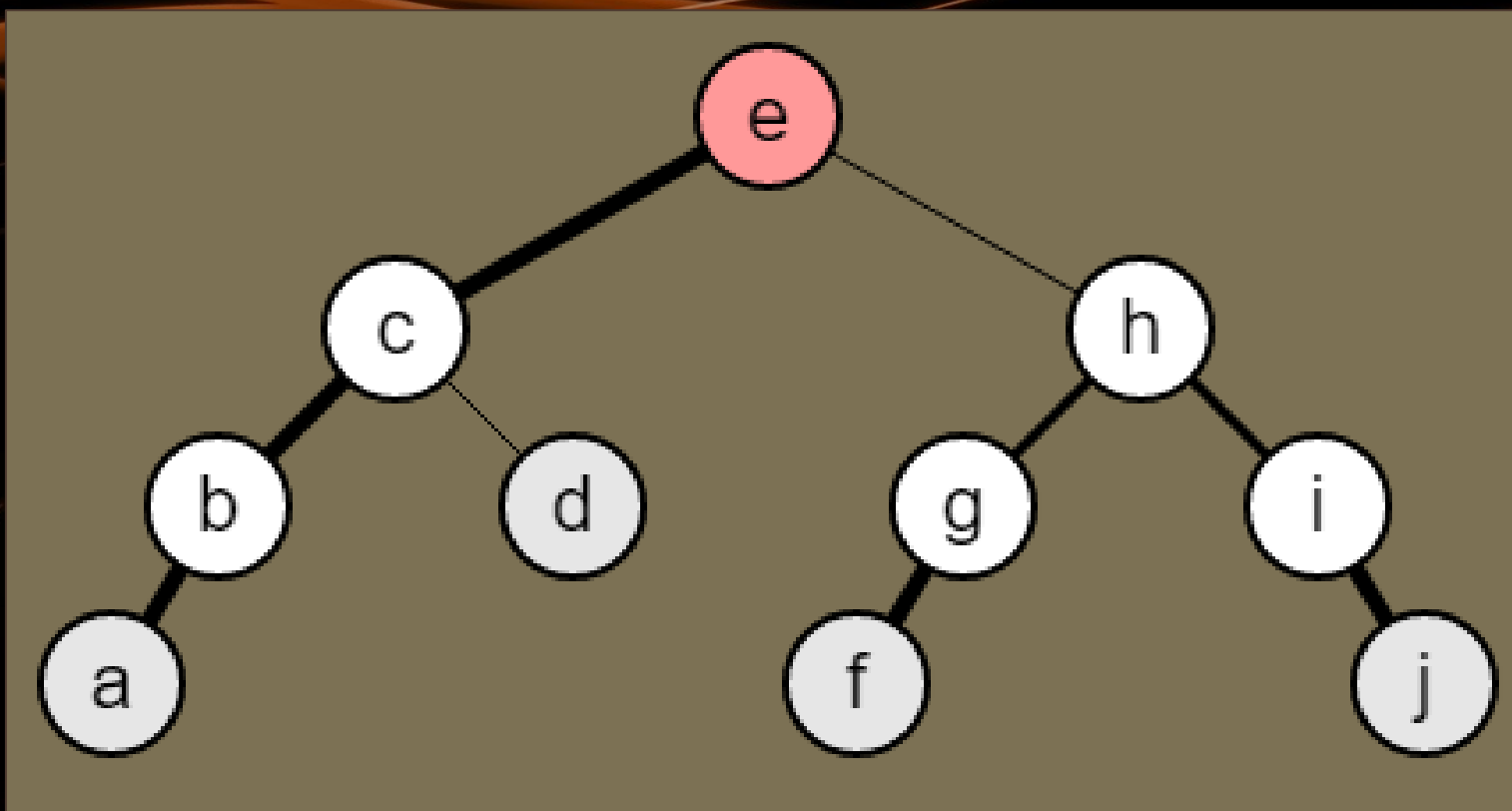
- ✦ Премахване на елемент, чието дясно поддърво няма ляво поддърво
  - ✦ Намираме елемента за премахване
  - ✦ Корена на дясното поддърво заема мястото на премахнатия елемент
- ✦ Example: Delete 19



# Двоично дърво за търсене - премахване

- ✦ Премахване на елемент, който има и ляво и дясно поддърво
  - ✦ Намираме елемента за премахване
  - ✦ Намираме най-малкия елемент в лявото разклонение на дясното му поддърво
  - ✦ Разменяме двата елемента и извършваме премахването
- ✦ Example: Delete 17





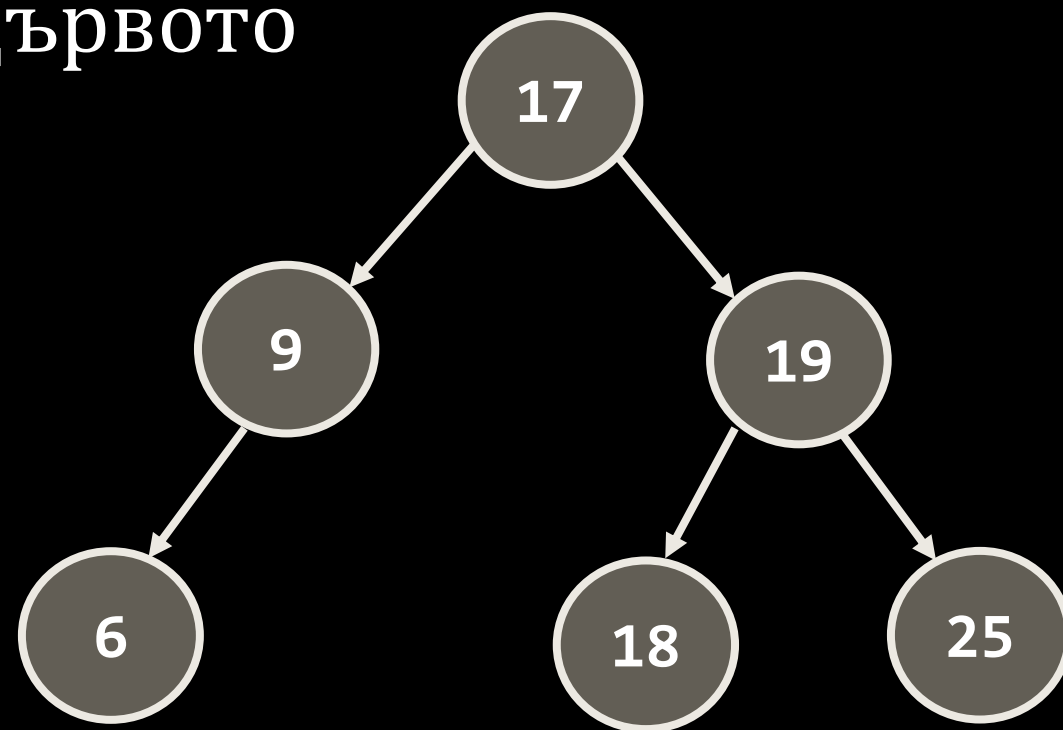
**Балансирани дървета**

Предназначение

# Двоични дървета за търсене – сложност

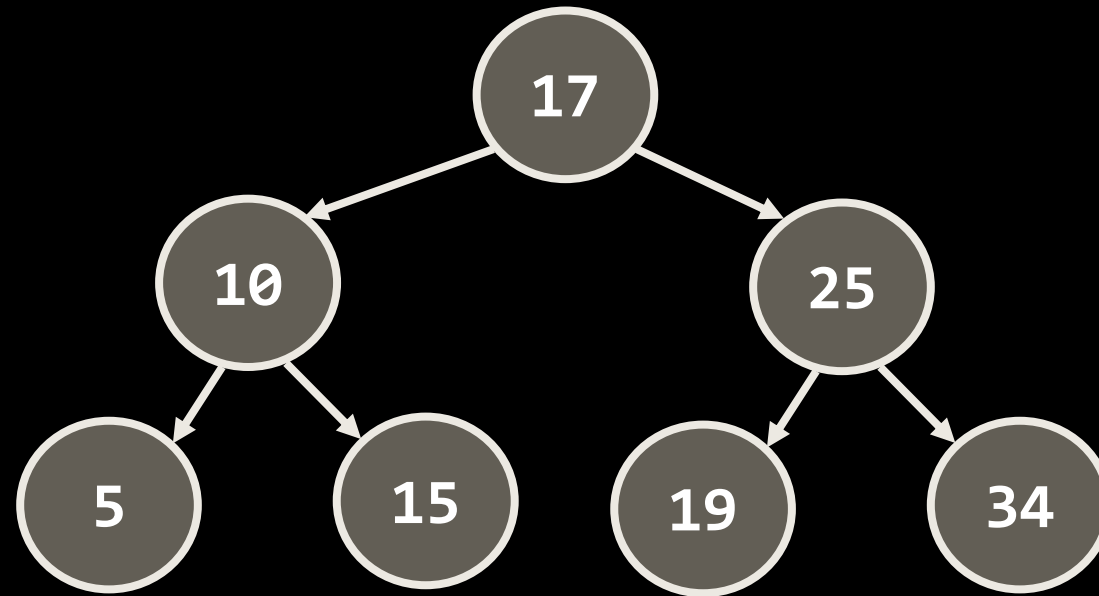
- Добавяне – височината на дървото
- Търсене – височината на дървото
- Премахване – височината на дървото

$O(n)$



# Двоично дърво за търсене - най-добър случай

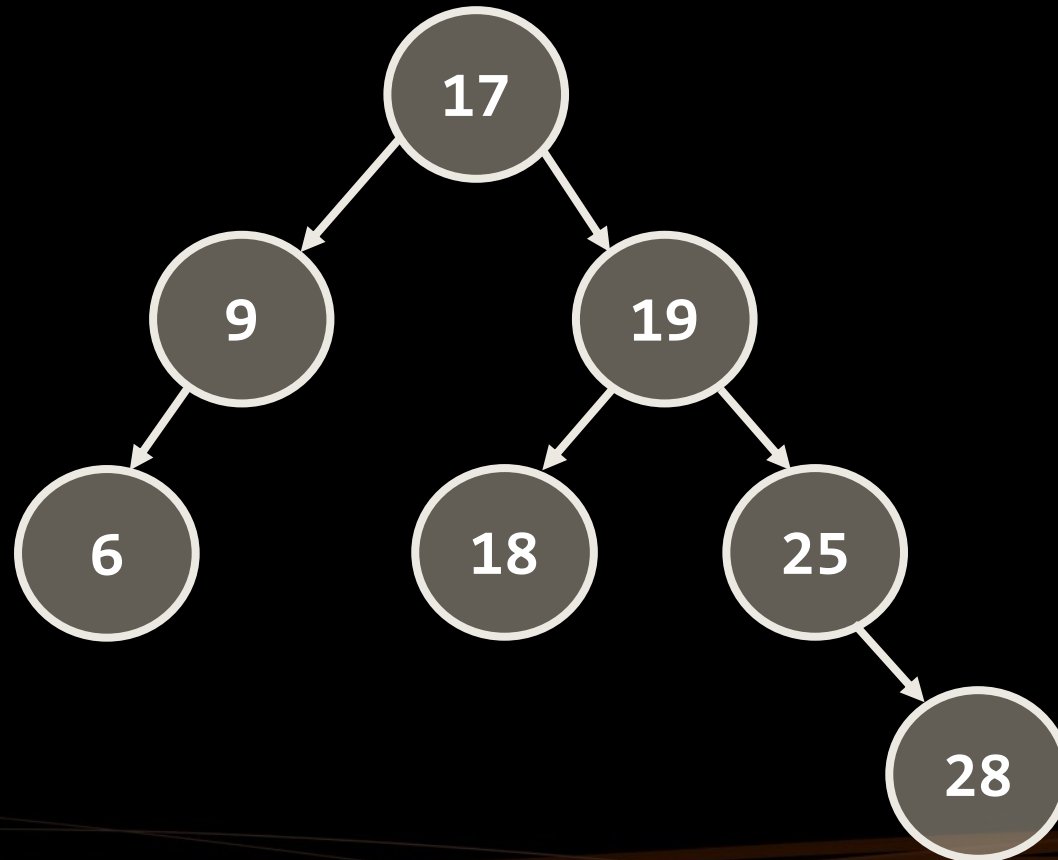
- Пример: Добавяме 17, 10, 25, 5, 15, 19, 34





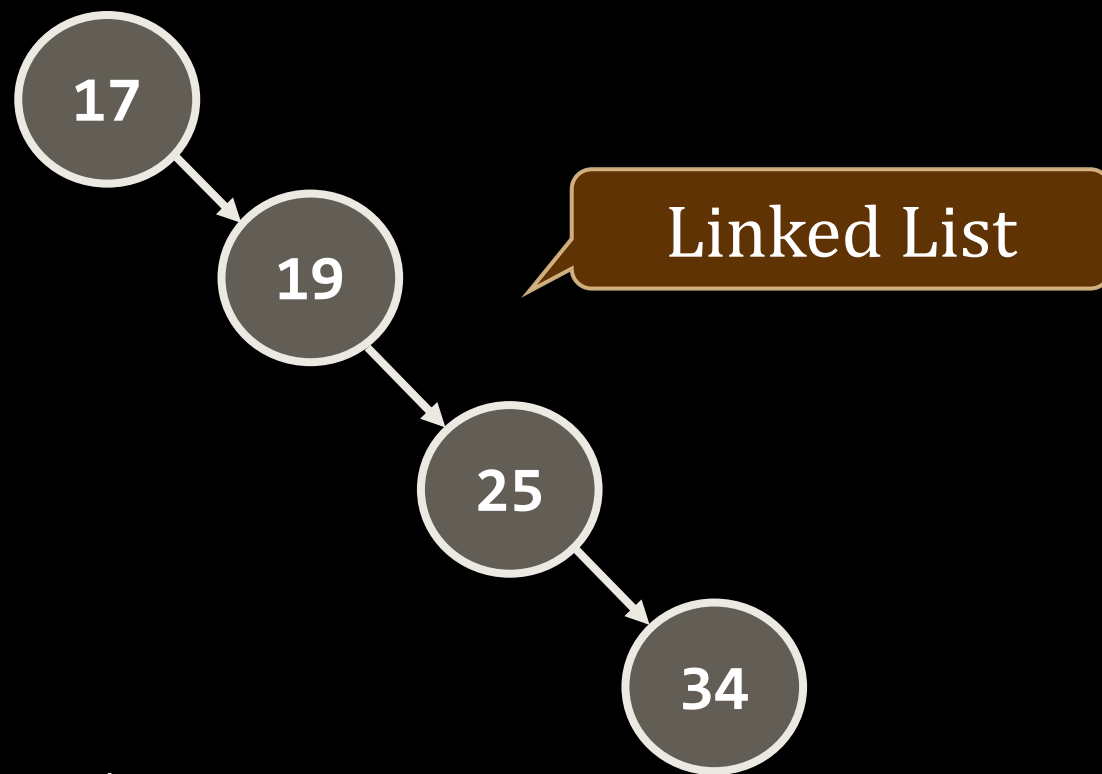
# Двоично дърво за търсене - стандартен случай

- Добавяне на стойности в произволна последователност
- Пример: Добавяме 17, 19, 9, 6, 25, 28, 18



# Двоично дърво за търсене - най-лош случай

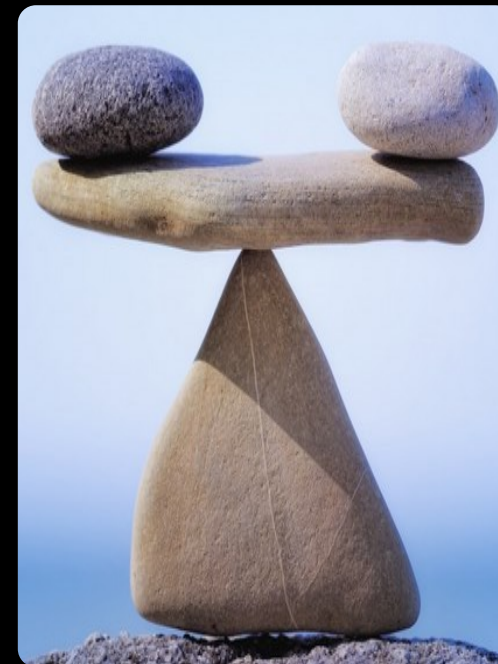
- Добавяне на стойности в нарастваща/намаляваща последователност



- Пример: Добавяме 17, 19, 25, 34

# Балансирани двоични дървета за търсене

- ✦ Двоичните дървета за търсене могат да бъдат **балансирани**
  - В **балансираните дървета** всеки възел има почти еднакъв брой възли във своите поддървета
- ✦ Балансираните дървета имат височина приблизително равна на  $\log(n)$



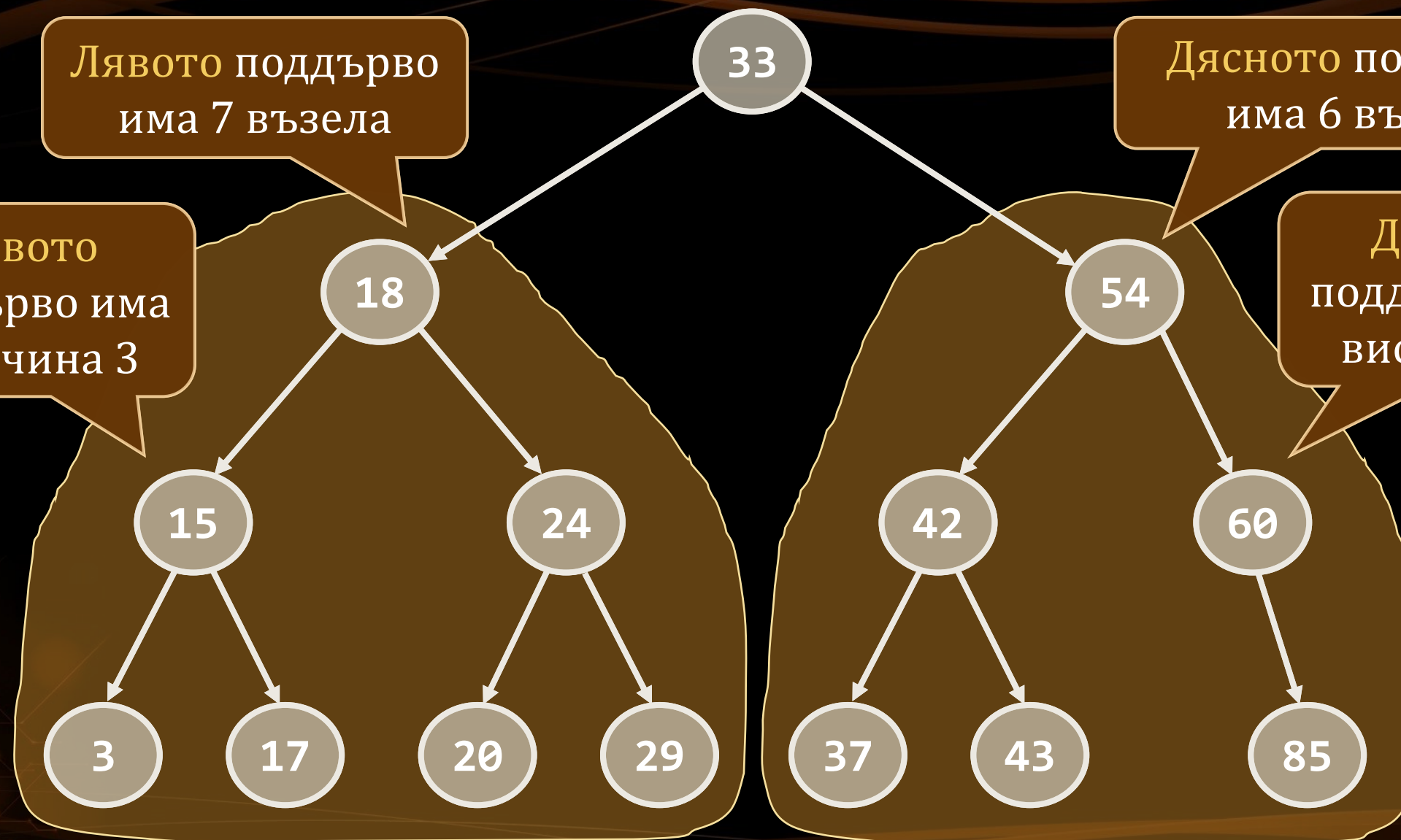
# Балансирани двоични дървета за търсене

Лявото поддърво  
има 7 възела

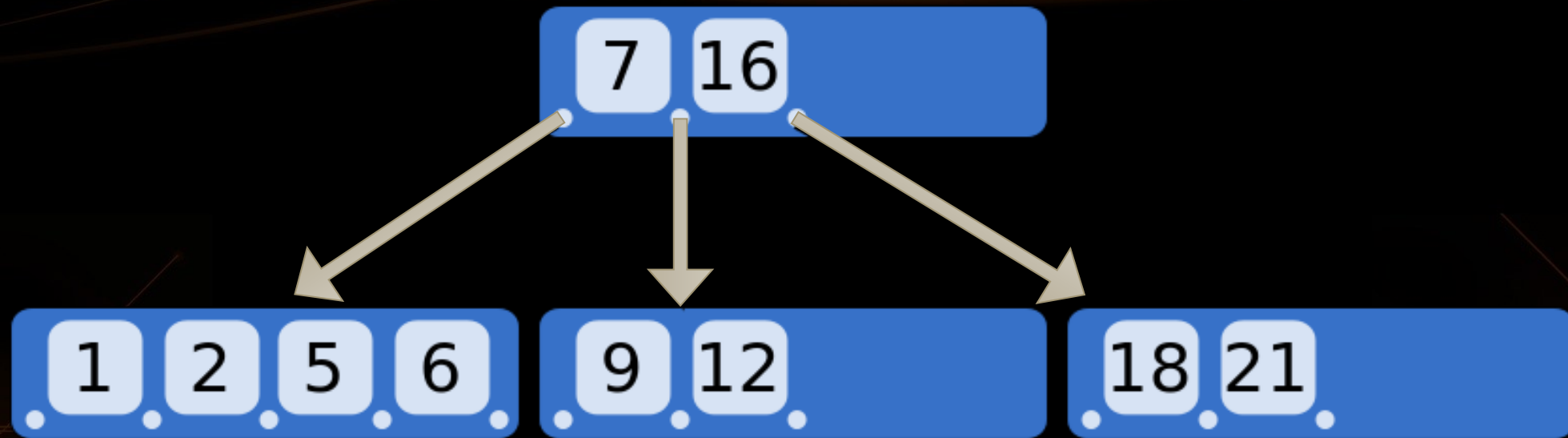
Дясното поддърво  
има 6 възела

Лявото  
поддърво има  
височина 3

Дясното  
поддърво има  
височина 3







## Б-Дървета (B-Trees)

Предназначение

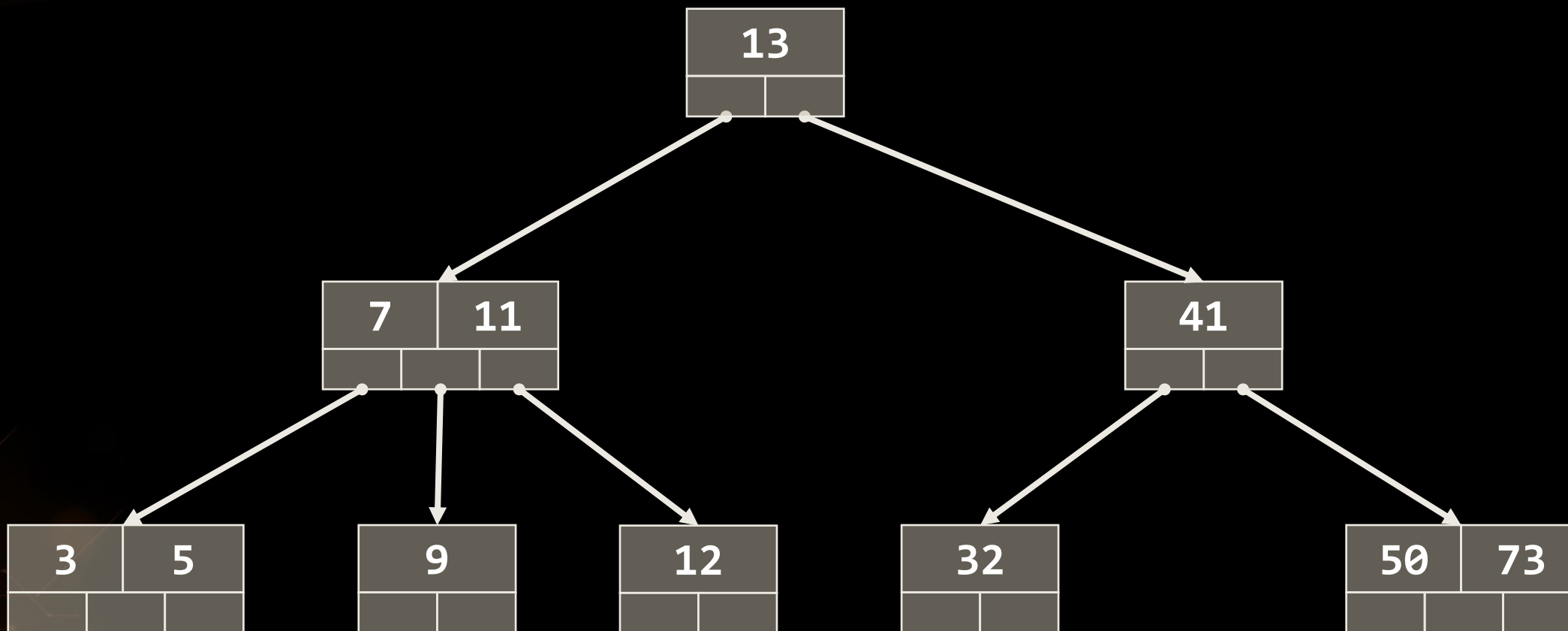


# Какво са B-Trees?

- B-trees са генерализация на концепцията за подредени двоични дървета за търсене (визуализация)
  - Всеки възел в B-tree от ред  $b$  съхранява между  $b$  и  $2*b$  ключове и има между  $b+1$  и  $2*b+1$  наследника
  - Ключовете във всеки възел са подредени нарастващо
  - Всички ключове в наследниците има стойности, ограничени в диапазона на техните леви и десни родителски ключове
- B-trees могат ефективно да се съхраняват на твърди дискове

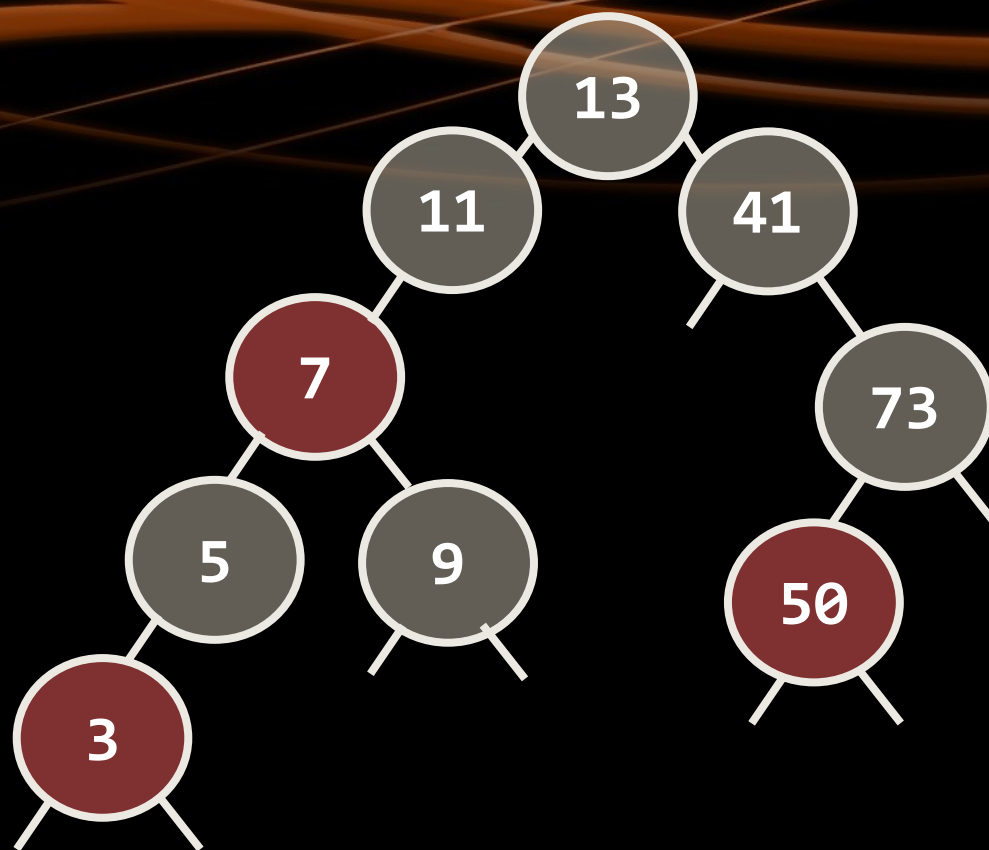
# B-Tree – пример

✦ B-Tree от ред 3, познати още като 2-3 дървета



# В-Trees и други балансирани дървета за търсене

- Възлите в В-Trees могат да имат **много наследници**
  - В-trees нямат нужда от често пребалансиране
- В-Trees са добри за **индексиране в бази от данни**:
  - Защото всеки възел може да се съхрани в отделен клъстер на твърдия диск
  - Минимизация на дисковите операции (които са много бавни)
- В-Trees са почти перфектно балансирани
  - Броя на възлите от корена до кой да е **null** възел са едни и същи



## Червено-черни дървета

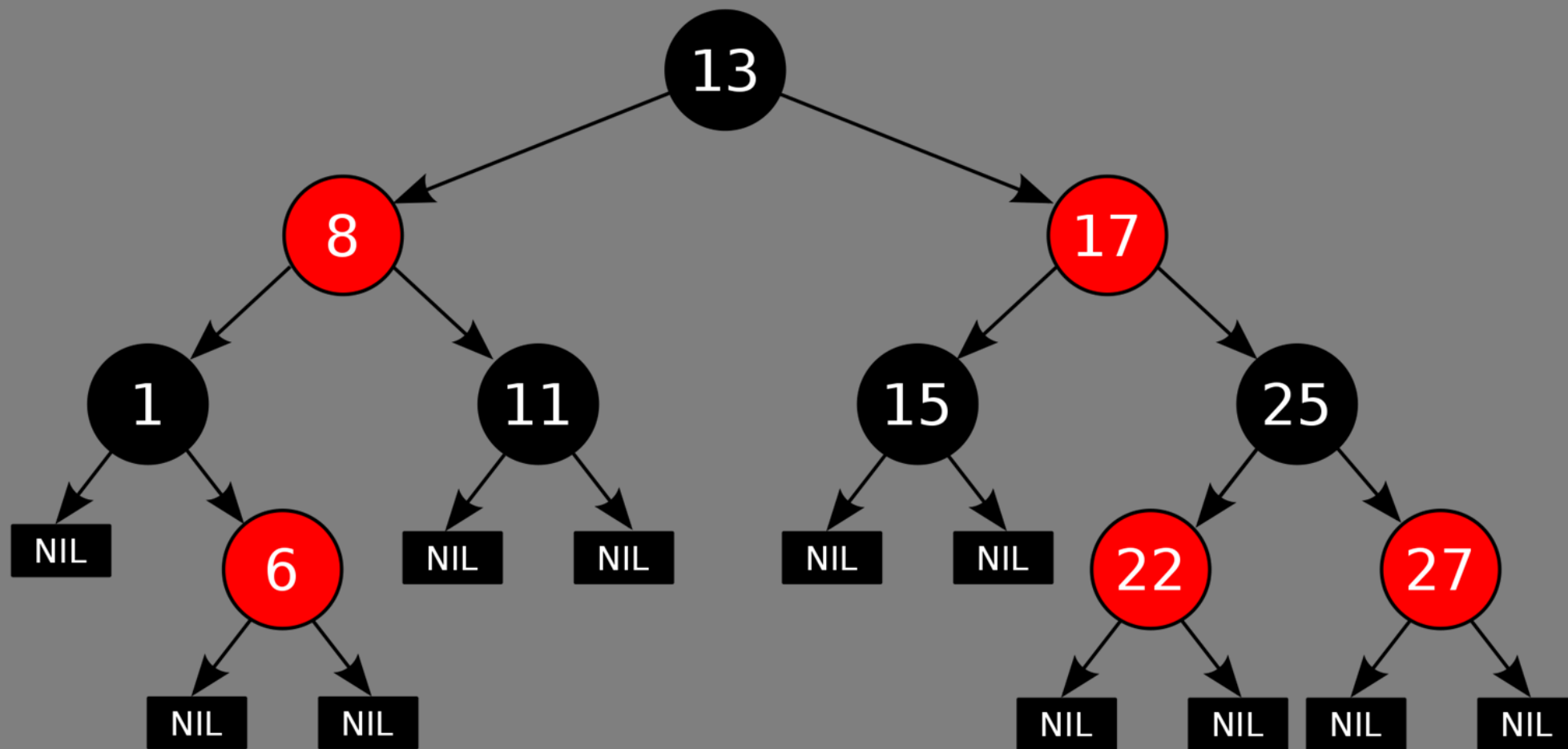
Семпла репрезентация на 2-3 дървета

# Свойства на червено-черни дървета

1. Всички листа са черни
2. Корена е черен
3. Няма възел, който да има две червени връзки към него
4. Всеки път от даден възел до листо в негово поддърво има еднакъв брой черни възли
5. Червените възли са винаги от ляво



# Червено-черно дърво



# Обобщение

- Дървета и дървовидни структури
- Подредени двоични дървета, балансирани дървета, В-дървета
- Упражнения: структура от данни “дърво”, използване на класове и библиотеки за дървовидни структури
- Обхождания в дълбочина и ширина (DFS и BFS)
- Упражнения: обхождане в дълбочина (DFS)
- Упражнения: обхождане в ширина (BFS)



# Министерство на образованието и науката (МОН)

- Настоящият курс (презентации, примери, задачи, упражнения и др.) е разработен за нуждите на Национална програма "**Обучение за ИТ кариера**" на МОН за подготовка по професия "Приложен програмист"



Министерство  
на образованието  
и науката



Национална  
програма  
„Обучение за  
ИТ кариера“

- Курсът е базиран на учебно съдържание и методика, предоставени от **фондация "Софтуерен университет"** и се разпространява под свободен лиценз **CC-BY-NC-SA**



SoftUni  
Foundation

