

Základy algoritmizace

4. Abstraktní datové typy

doc. Ing. Jiří Vokřínek, Ph.D.

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Základy algoritmizace

- Dnes:
 - Abstraktní datové typy
 - Zásobník
 - Fronta
 - Řazení pomocí haldy

Abstraktní datový typ

Abstraktní datový typ

- **Datová struktura** (typ) je množina dat a operací s těmito daty
- **Abstraktní datový typ** formálně definuje data a operace s nimi

Nezávisle na konkrétní implementaci

- **Příklady:**
 - Fronta (queue)
 - Zásobník (stack)
 - Pole (array)
 - Tabulka (table)
 - Seznam (list)
 - Strom (tree)
 - Množina (set)

Abstraktní datový typ

- **Abstraktní datový typ** je množina dat (hodnot) a příslušných operací, které jsou přesně specifikovány a to **nezávisle na konkrétní implementaci**

- **Můžeme definovat:**
 - Matematicky – signatura a axiomy
 - Rozhraním (interface) a popisem operací, rozhraní poskytuje:
 - Konstruktor vracející odkaz (na objekt nebo strukturu)
Objektově orientovaný i procedurální přístup
 - Operace, které akceptují odkaz na argument (data) jako argument a které mají přesně definovaný účinek na data

- **Příklad matematického popisu – datový typ Boolean**

Matematický popis ADT - Boolean

- **Syntax** popisuje, jak správně vytvořit logický výraz:

1. `true` a `false` jsou logické výrazy
2. Jestliže x a y jsou logické výrazy, pak

- I. $!(x)$ – negace
- II. $(x \& y)$ – logický součin `and`
- III. $(x | y)$ – logický součet `or`
- IV. $(x == y)$, $(x != y)$ – relační operátory

jsou logické výrazy

*Pokud se chceme vyhnout psát u každé operace závorky,
musíme definovat priority operátorů*

Konkrétní implementace se může syntakticky lišit – viz. `!` vs. `not`, atp.

Matematický popis ADT - Boolean

- **Sémantika** popisuje význam jednotlivých operací

- Můžeme definovat axiomy:

- `true == true : true`

- `false == false : true`

- `!(true) : false`

- `x & false : false`

- `x & y : y & x`

- `x | false : x`

- `true == false : false`

- `false == true : false`

- `!(false) : true`

- `x & true : x`

- `x | y : y | x`

- `x | true : true`

Abstraktní datový typ - vlastnosti

- Počet datových položek může být
 - Neměnný – **statický datový typ** – počet položek je konstantní
Např. pole, řetězec, třída, ...
 - Proměnný – **dynamický datový typ** – počet položek se mění v závislosti na provedené operaci
Např. vložení nebo odebrání určitého prvku
- Typ položek (dat) může být
 - **Homogenní** – všechny položky jsou stejného typu
 - **Nehomogenní** – položky mohou být různých typů
- Existence bezprostředního následníka je
 - **Lineární** – existuje bezprostřední následník prvku, např. pole, fronta, seznam, ...
 - **Nelineární** – neexistuje přímý jednoznačný následník, např. strom

Abstraktní datový typ jako objekt

- Objekt je instancí třídy

- Třída je vzor – má rozhraní, data (vnitřní proměnné), metody (funkce)

Objektově orientované programování

```
class MyAbstractDataType:  
    def __init__(self):  
        self.dataOne = 1  
        self.dataTwo = 2  
  
    def operationOne(self, argument):  
        self.dataOne = argument  
  
    def operationTwo(self):  
        return self.dataTwo
```

```
myData = MyAbstractDataType()  
myData.operationOne(42)  
myData.operationTwo()
```

Zásobník

Zásobník

- Dynamická datová struktura umožňující vkládání a odebrání hodnot tak, že naposledy vložená hodnota se odebere jako první

LIFO – Last In, First Out

- Základní operace
 - Vložení hodnoty na vrchol zásobníku
 - Odebrání hodnoty z vrcholu zásobníku
 - Test prázdnosti zásobníku



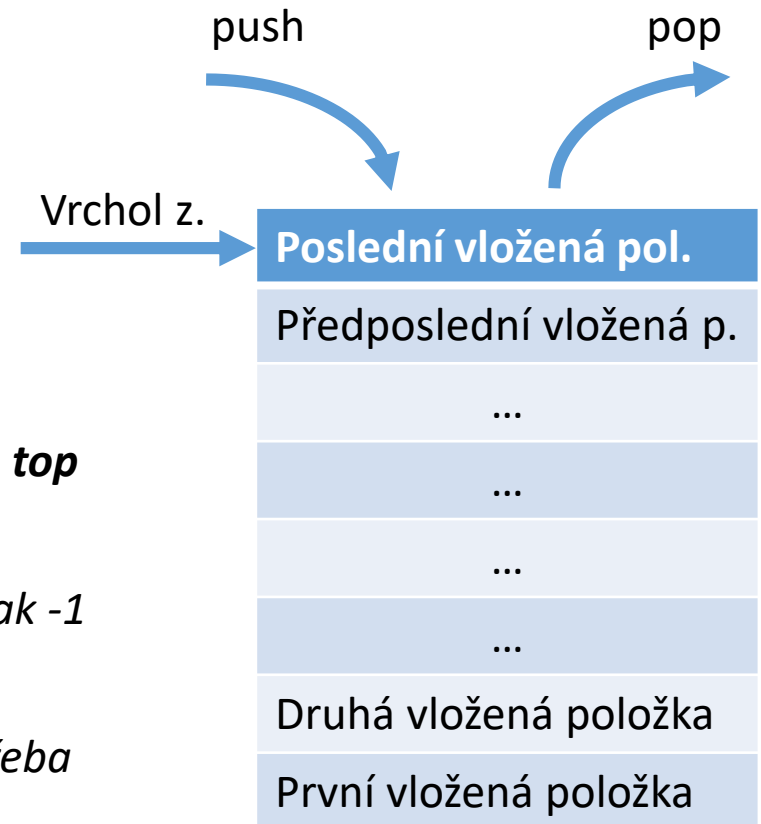
Zásobník

■ Základní operace

- **push** – vložení hodnoty na vrchol zásobníku
- **pop** – odebrání hodnoty z vrcholu zásobníku
- **isEmpty** – test prázdnosti zásobníku

■ Další operace mohou být

- **peek** – čtení hodnoty z vrcholu
Alternativně také třeba top
- **search** – vrátí pozici prvku
Pokud se nachází v zásobníku, jinak -1
- **size** – aktuální počet prvků
Nebývá potřeba



Zásobník

- Příklad implementace pomocí List
 - push \approx append
 - pop \approx pop
 - isEmpty \approx not len(stack)

```
stack = []
stack.append("dela")
stack.append("to")
stack.append("co")
stack.append("vime")

while len(stack):
    print(stack.pop())
```

Uměli byste implementovat zásobník pomocí pole?

Zásobník

- Příklad implementace s definicí rozhraní

```
class Stack:
```

```
    def __init__(self):  
        self.items = []
```

```
    def push(self, item):  
        self.items.append(item)
```

```
    def pop(self):  
        return self.items.pop()
```

```
    def isEmpty(self):  
        return (self.items == [])
```

```
s = Stack()  
s.push("dela")  
s.push("to")  
s.push("co")  
s.push("vime")
```

```
while not s.isEmpty():  
    print(s.pop())
```

Zásobník

- Příklad použití – kontrola vyváženosti závorek

```
def parChecker(symbolString):  
    s = Stack()  
    for symbol in symbolString:  
        if symbol == "(":  
            s.push(symbol)  
        elif symbol == ")":  
            if s.isEmpty():  
                return False  
            else:  
                s.pop()  
    return s.isEmpty()
```

```
parChecker(' (3+ (2* (-2) + (3*5) -1) / (3-2) *2) ')
```

```
parChecker(' ( (3+2) + (5*8) * (4) ')
```

Fronta

Fronta

- Dynamická datová struktura umožňující vkládání a odebírání hodnot v pořadí, v jakém byly vloženy

FIFO – First In, First Out

- Základní operace
 - Vložení hodnoty na konec fronty
 - Odebrání hodnoty z čela fronty
 - Test prázdnosti fronty



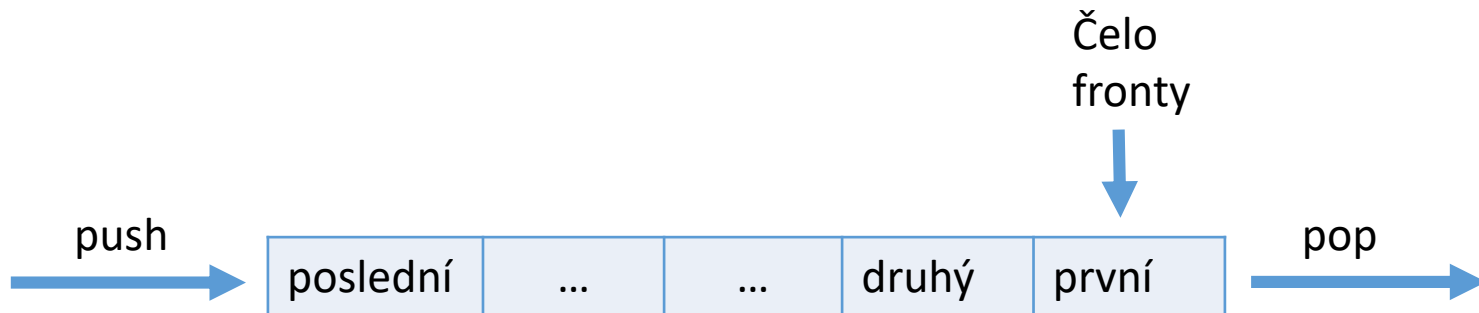
Fronta

- Základní operace

- `push` – vložení hodnoty na konec fronty (nebo též `enqueue`)
- `pop` – odebrání hodnoty z čela fronty (nebo též `dequeue`)
- `isEmpty` – test prázdnosti fronty

- Další operace mohou být

- `peek` – čtení hodnoty z čela
- `back` – čtení hodnoty z konce



Fronta

- Příklad implementace pomocí List
 - push \approx append
 - pop \approx pop(0)
 - isEmpty \approx not len(stack)

```
queue = []  
queue.append("dela")  
queue.append("to")  
queue.append("co")  
queue.append("vime")  
  
while len(queue):  
    print(queue.pop(0))
```

Pozor, operace pop(0) je pomalá – víte proč?

Fronta

■ Příklad implementace s definicí rozhraní

Jaký je rozdíl v použití seznamu oproti minulému příkladu?

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

```
q = Queue()
q.enqueue("dela")
q.enqueue("to")
q.enqueue("co")
q.enqueue("vime")

while not q.isEmpty():
    print(q.dequeue())
```

Fronta

- Rozšíření – prioritní fronta

- Prvky jsou odebírány na základě priority (např. velikosti)

Jak to lze naimplementovat?

- Hledání největšího prvku při odebírání
- Seřazení po vložení
- Seřazení při vložení

Určitě to jde lépe s pomocí složitějších struktur

Binární Halda

Binární Halda

■ Halda

- Binární strom, kde každý prvek je větší než jeho následovníci
- Je vyvážený
- Každý prvek je co nejvíce vlevo
- Implementujeme pomocí pole, kde potomci prvku n jsou na pozici $2n+1$ a $2n+2$

■ Základní operace

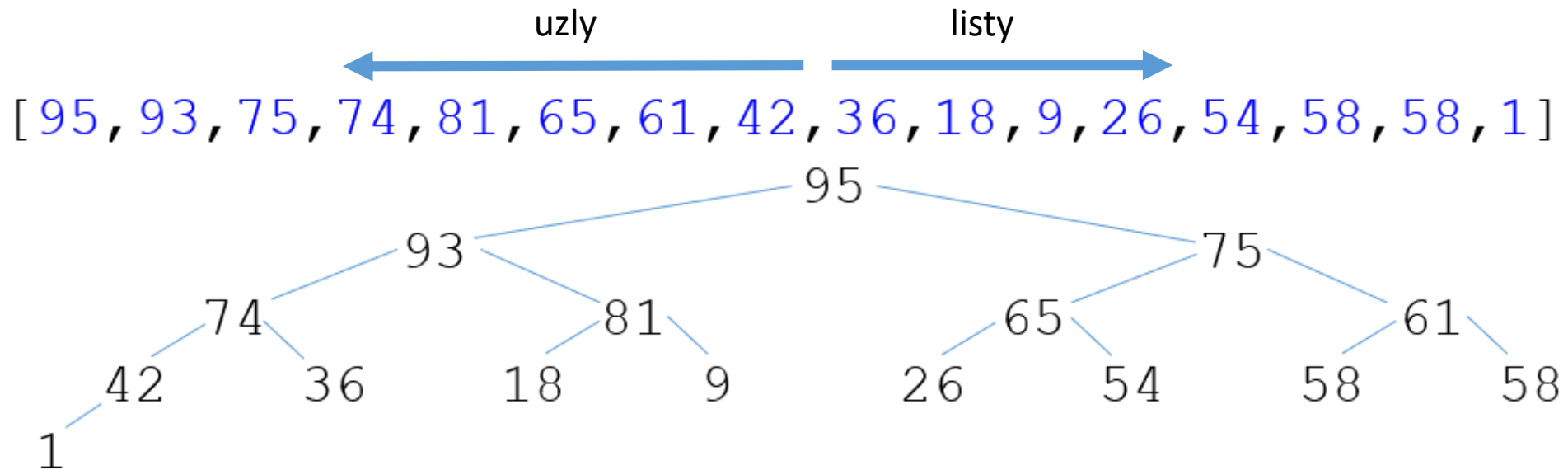
- `make` – vytvoření haldy
- `min`, `max` – vrátí minimální, resp. maximální prvek
- `push/pop/insert/delete` – přidání, resp. odebrání prvku

Binární Halda

- Tvorba haldy

- Halda – pole prvků h_1, \dots, h_n , kde $h_{n/2-1}, \dots, h_n$ jsou listy
- Je třeba zajistit vlastnosti haldy – pro $h_{n/2-2}$ až h_1 kontrolujeme přípustnost – výměna prvků

- Příklad: $n=16$, $n/2-1=7$, indexace od 0, potomci $2i+1, 2i+2$



Binární Halda

- Tvorba haldy

- Halda – pole prvků h_1, \dots, h_n , kde $h_{n/2-1}, \dots, h_n$ jsou listy
- Je třeba zajistit vlastnosti haldy – pro $h_{n/2-2}$ až h_1 kontrolujeme přípustnost – výměna prvků

```
for start in range((len(array)-2)//2, -1, -1):  
    movedown(array, start, len(array)-1)
```

Binární Halda

■ Tvorba haldy

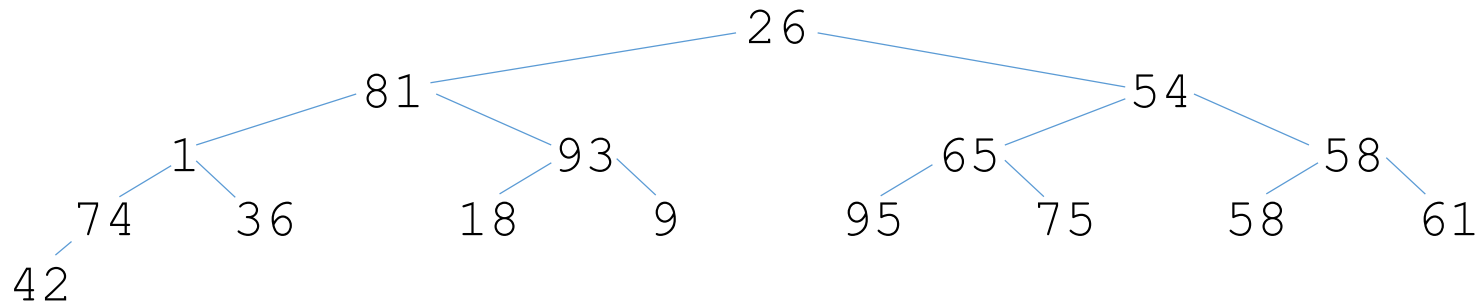
- Halda – pole prvků h_1, \dots, h_n , kde $h_{n/2-1}, \dots, h_n$ jsou listy
- Je třeba zajistit vlastnosti haldy – pro $h_{n/2-2}$ až h_1 kontrolujeme přípustnost – výměna prvků

```
def movedown(array, start, end):  
    root = start  
    while True:  
        child = root * 2 + 1  
        if child > end: break  
        if child + 1 <= end and array[child] < array[child + 1]:  
            child += 1  
        if array[root] < array[child]:  
            array[root], array[child] = array[child], array[root]  
            root = child  
    else:  
        break
```

Binární Halda

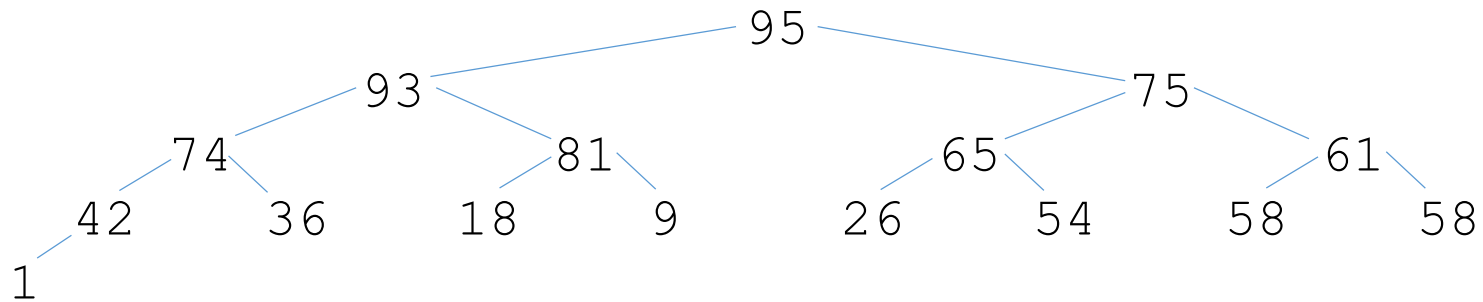
■ Vstup

[26, 81, 54, 1, 93, 65, 58, 74, 36, 18, 9, 95, 75, 58, 61, 42]



■ Heap

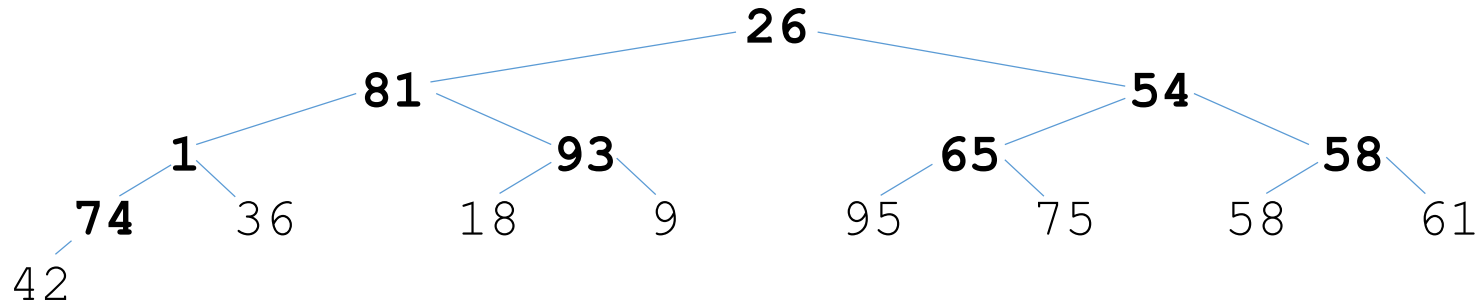
[95, 93, 75, 74, 81, 65, 61, 42, 36, 18, 9, 26, 54, 58, 58, 1]



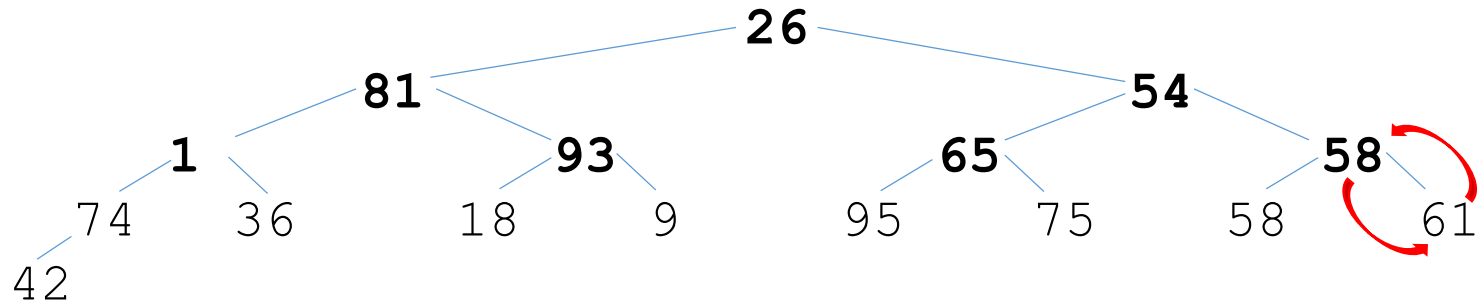
Binární Halda

■ Tvorba haldy ($h_{n/2-2}$ až h_1)

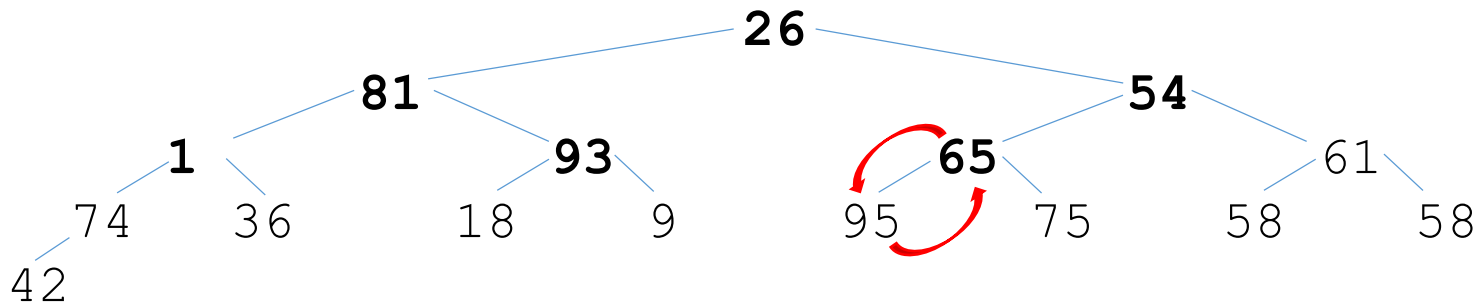
[**26**, **81**, **54**, **1**, **93**, **65**, **58**, **74**, 36, 18, 9, 95, 75, 58, 61, 42]



[**26**, **81**, **54**, **1**, **93**, **65**, **58**, 74, 36, 18, 9, 95, 75, 58, 61, 42]



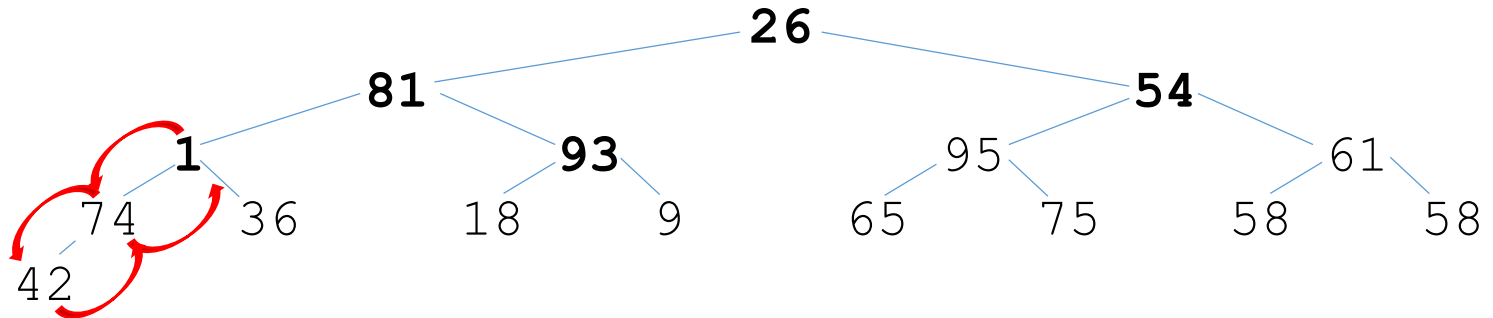
[**26**, **81**, **54**, **1**, **93**, **65**, 61, 74, 36, 18, 9, 95, 75, 58, 58, 42]



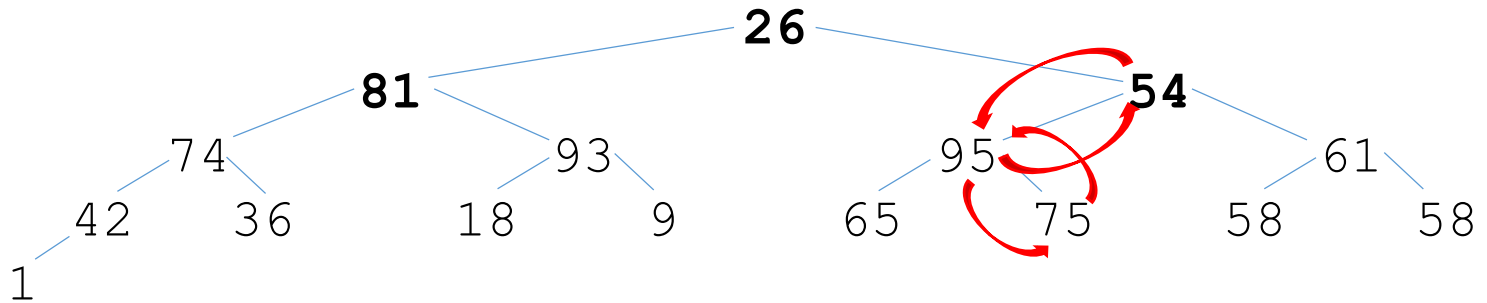
Binární Halda

■ Tvorba haldy ($h_{n/2-2}$ až h_1)

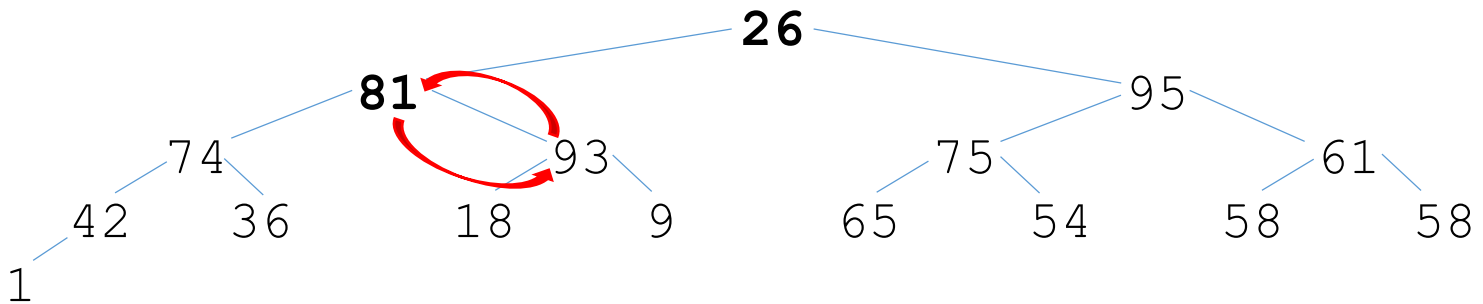
[**26**, **81**, **54**, **1**, **93**, 95, 61, 74, 36, 18, 9, 65, 75, 58, 58, 42]



[**26**, **81**, **54**, 74, 93, 95, 61, 42, 36, 18, 9, 65, 75, 58, 58, 1]



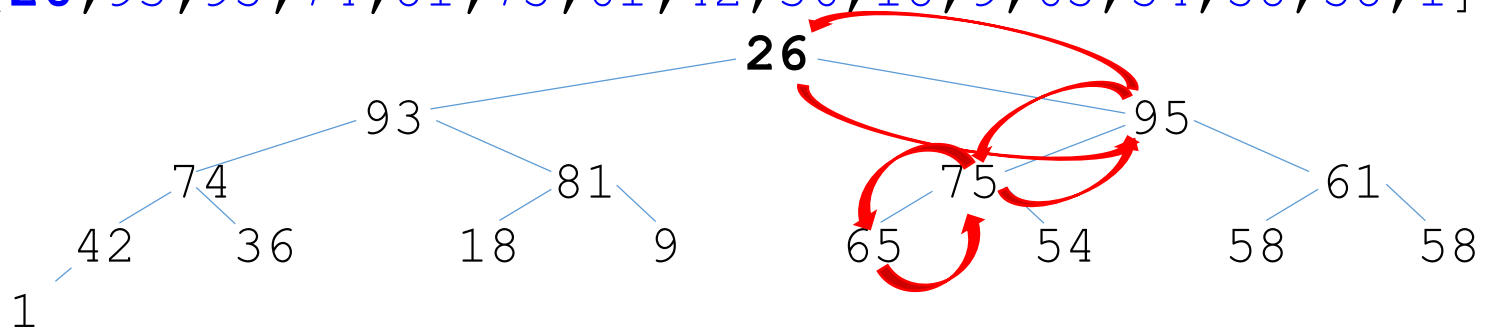
[**26**, **81**, 95, 74, 93, 75, 61, 42, 36, 18, 9, 65, 54, 58, 58, 1]



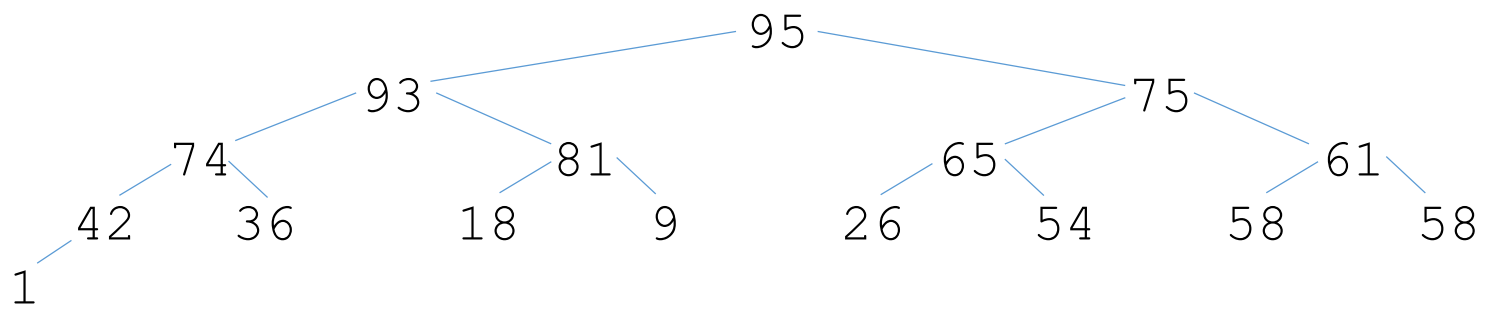
Binární Halda

■ Tvorba haldy ($h_{n/2-2}$ až h_1)

[**26**, 93, 95, 74, 81, 75, 61, 42, 36, 18, 9, 65, 54, 58, 58, 1]



[95, 93, 75, 74, 81, 65, 61, 42, 36, 18, 9, 26, 54, 58, 58, 1]



Heap Sort

Heap Sort

- Řazení pomocí převodu na binární haldu
- Implementujeme pomocí pole, kde potomci prvku n jsou na pozici $2n+1$ a $2n+2$
- Vyjímáním max. elementu budujeme setříděný seznam
 - Pokud není halda prázdná, zaměníme první a poslední prvek
 - „spravíme“ haldu
- Není stabilní

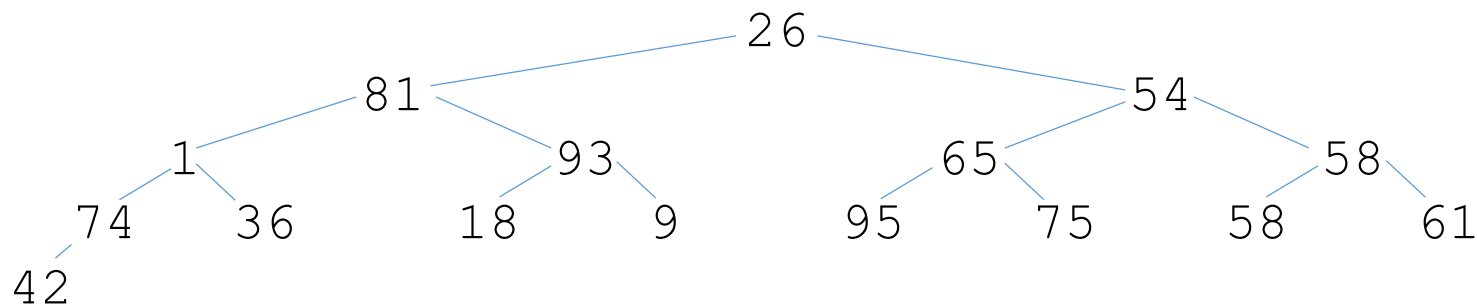
Heap Sort

- Tvorba haldy
 - Halda – pole prvků h_1, \dots, h_n , kde $h_{n/2-1}, \dots, h_n$ jsou listy
 - Je třeba zajistit vlastnosti haldy – pro $h_{n/2-2}$ až h_1 kontrolujeme přípustnost – výměna prvků
- Po výměně prvku je třeba opět kontrola haldy
 - Kontrolujeme přípustnost pro h_1
- Zajištění přípustnosti
 - pokud je daný prvek menší než některý z jeho následníků, vyměníme je a kontrolujeme přípustnost vyměněného následníka

Heap Sort

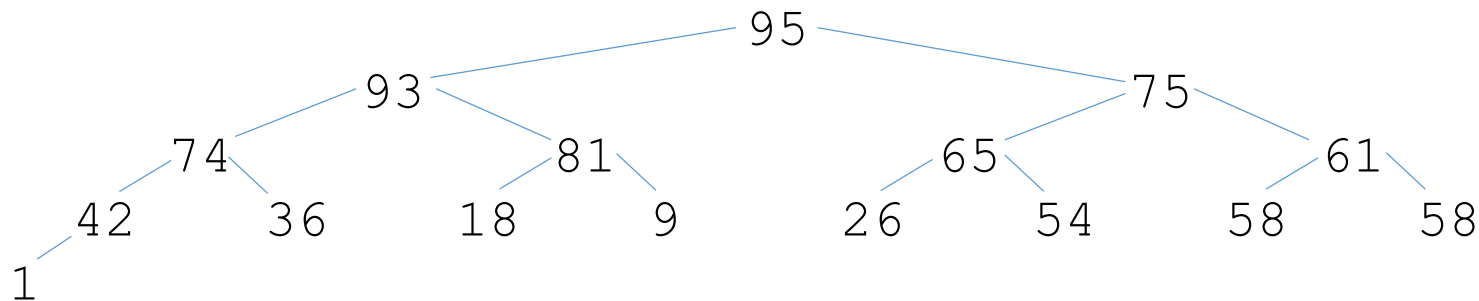
■ Vstup

[26, 81, 54, 1, 93, 65, 58, 74, 36, 18, 9, 95, 75, 58, 61, 42]



■ Heap

[95, 93, 75, 74, 81, 65, 61, 42, 36, 18, 9, 26, 54, 58, 58, 1]



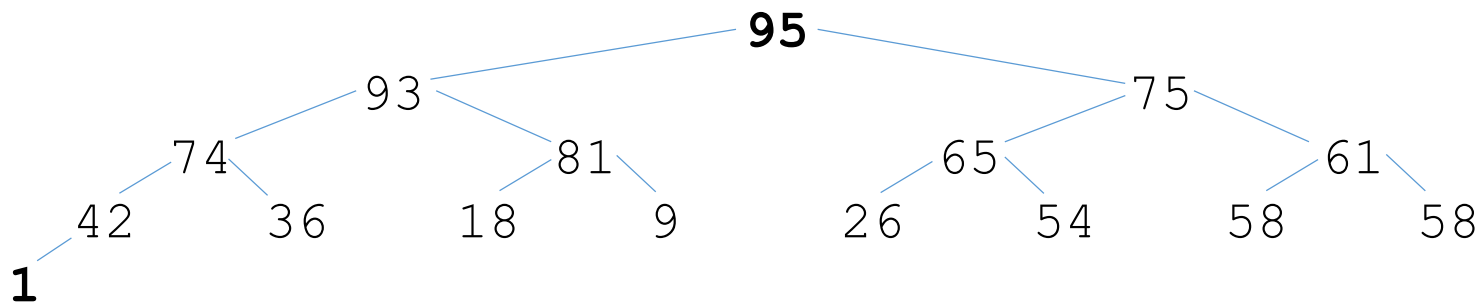
■ Výstup

[1, 9, 18, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]

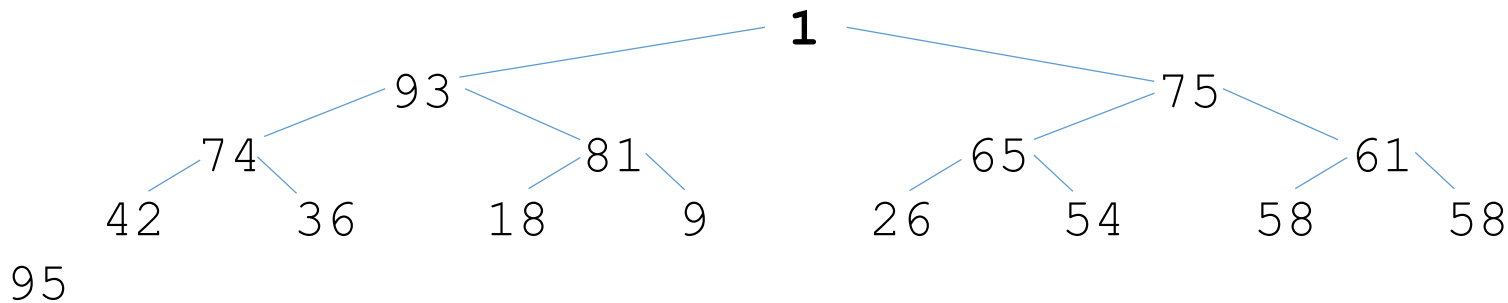
Heap Sort

■ Výběr prvku a kontrola haldy

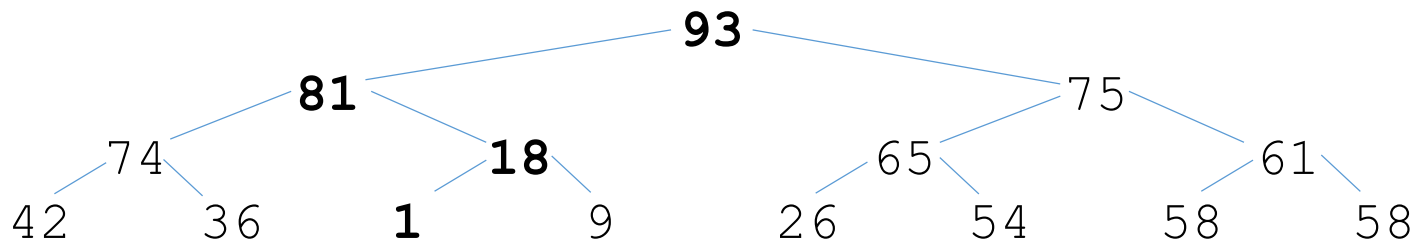
[**95**, 93, 75, 74, 81, 65, 61, 42, 36, 18, 9, 26, 54, 58, 58, **1**]



[1, 93, 75, 74, 81, 65, 61, 42, 36, 18, 9, 26, 54, 58, 58, 95]



[**93**, **81**, 75, 74, **18**, 65, 61, 42, 36, **1**, 9, 26, 54, 58, 58, 95]

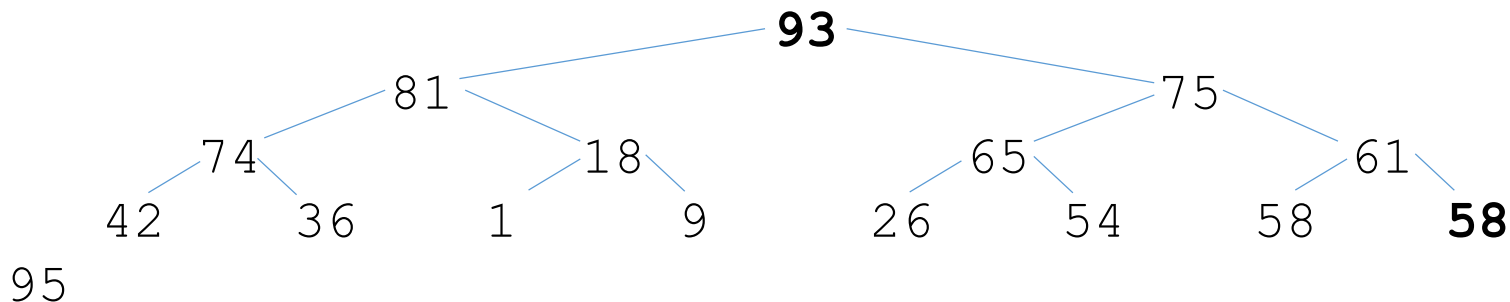


95

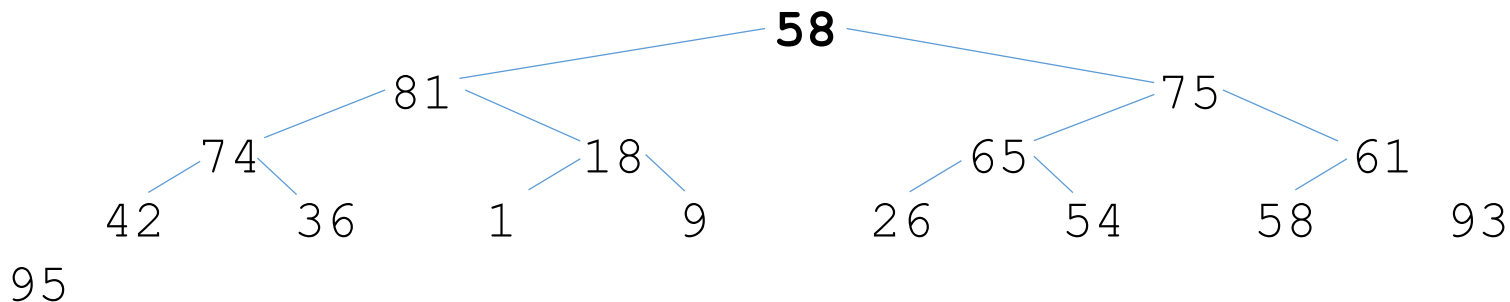
Heap Sort

■ Výběr prvku a kontrola haldy

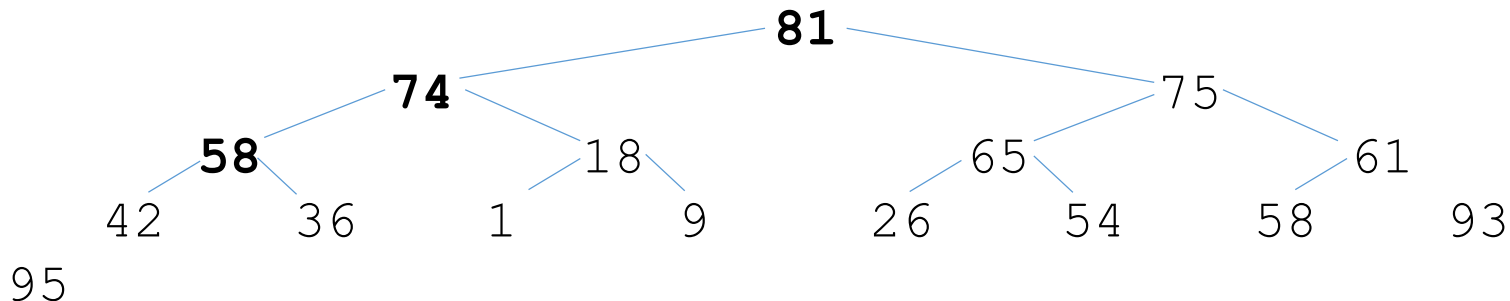
[**93**, 81, 75, 74, 18, 65, 61, 42, 36, 1, 9, 26, 54, 58, **58**, 95]



[58, 81, 75, 74, 18, 65, 61, 42, 36, 1, 9, 26, 54, 58, 93, 95]



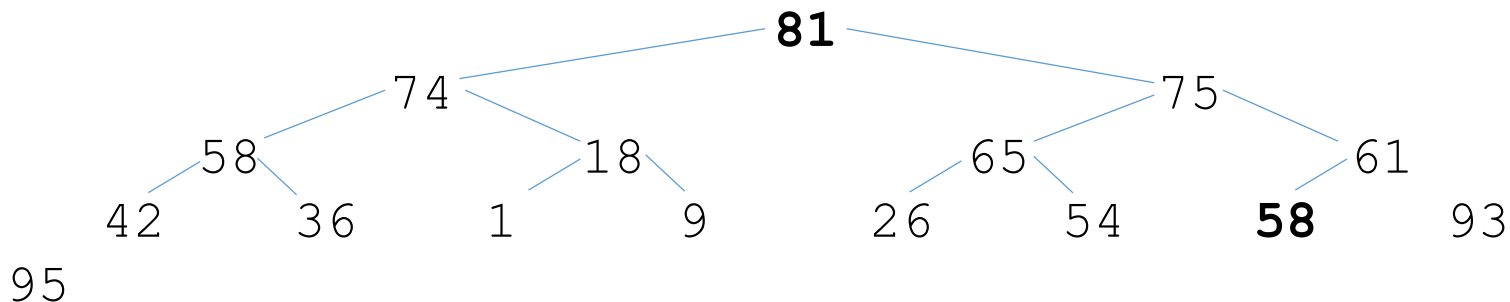
[**81**, **74**, 75, **58**, 18, 65, 61, 42, 36, 1, 9, 26, 54, 58, 93, 95]



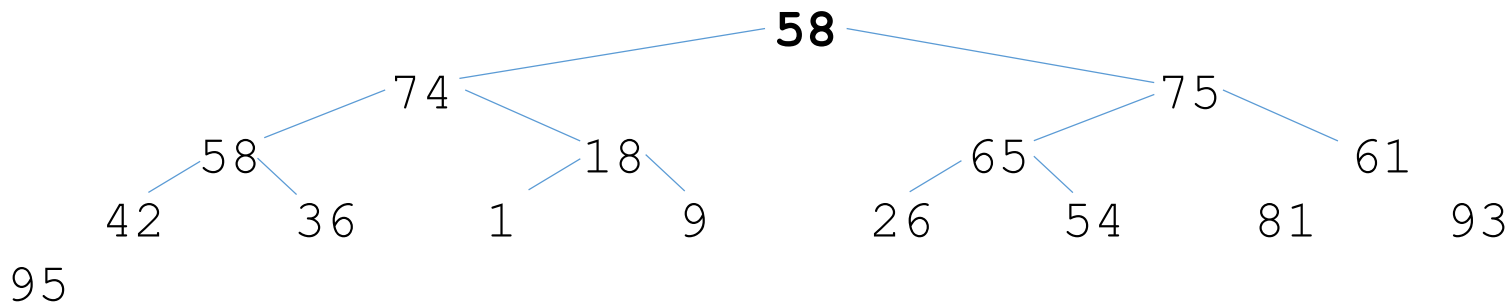
Heap Sort

■ Výběr prvku a kontrola haldy

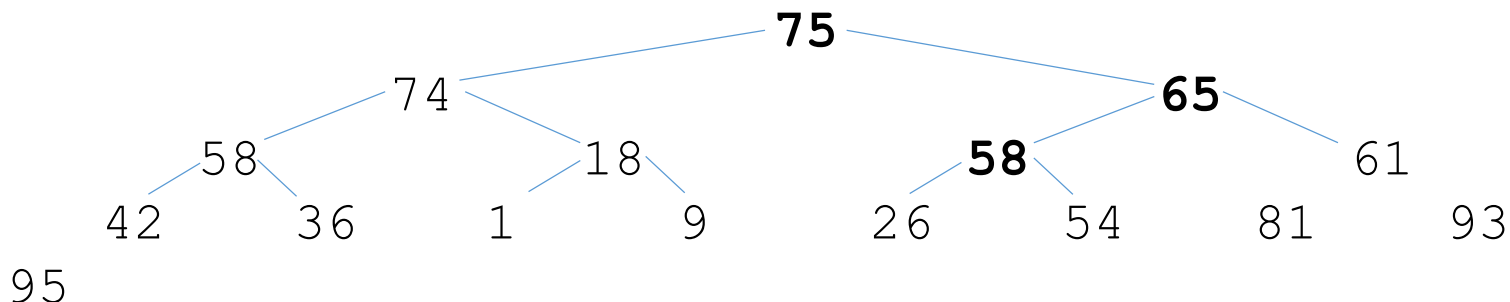
[**81**, 74, 75, 58, 18, 65, 61, 42, 36, 1, 9, 26, 54, **58**, 93, 95]



[58, 74, 75, 58, 18, 65, 61, 42, 36, 1, 9, 26, 54, 81, 93, 95]



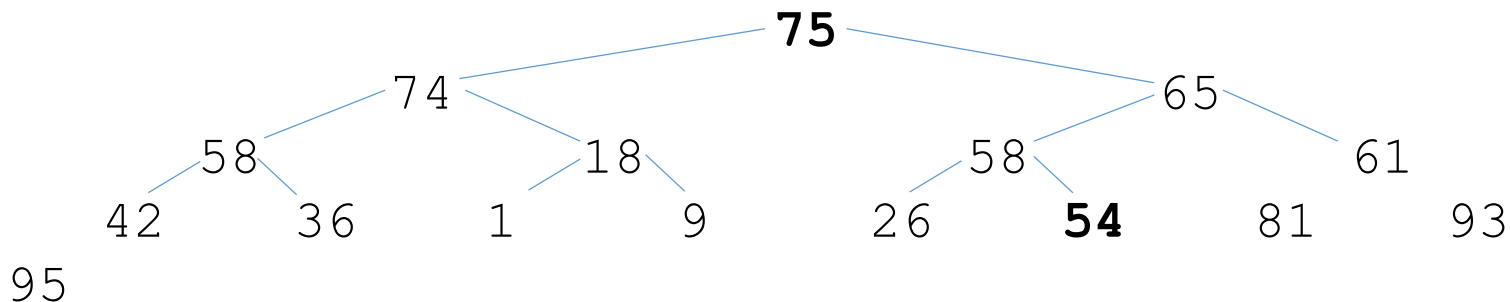
[75, 74, 65, 58, 18, 58, 61, 42, 36, 1, 9, 26, 54, 81, 93, 95]



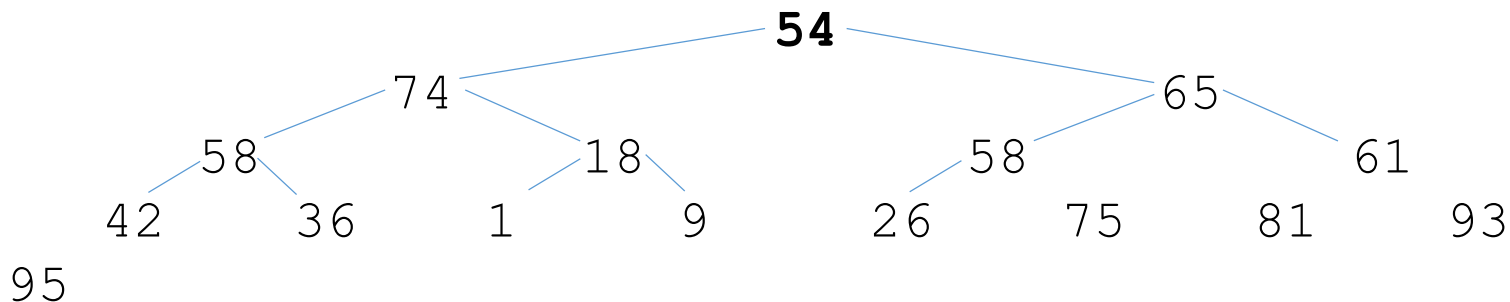
Heap Sort

■ Výběr prvku a kontrola haldy

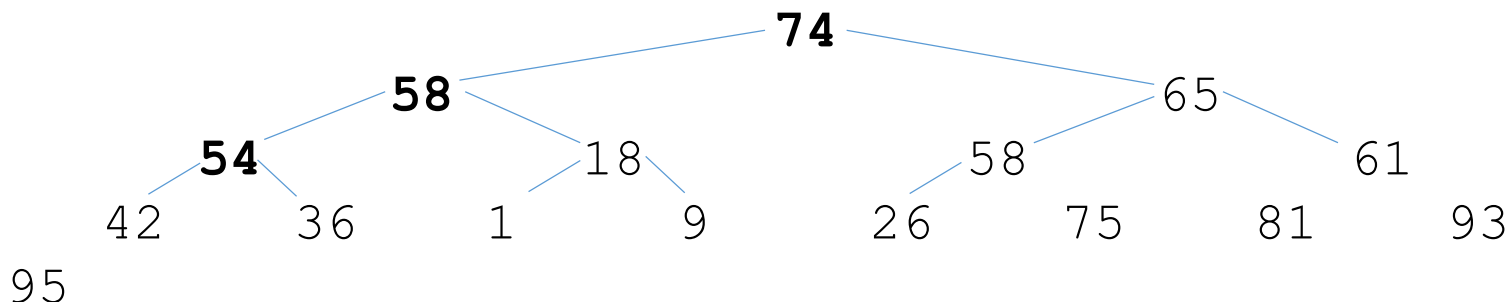
[**75**, 74, 65, 58, 18, 58, 61, 42, 36, 1, 9, 26, **54**, 81, 93, 95]



[54, 74, 65, 58, 18, 58, 61, 42, 36, 1, 9, 26, 75, 81, 93, 95]



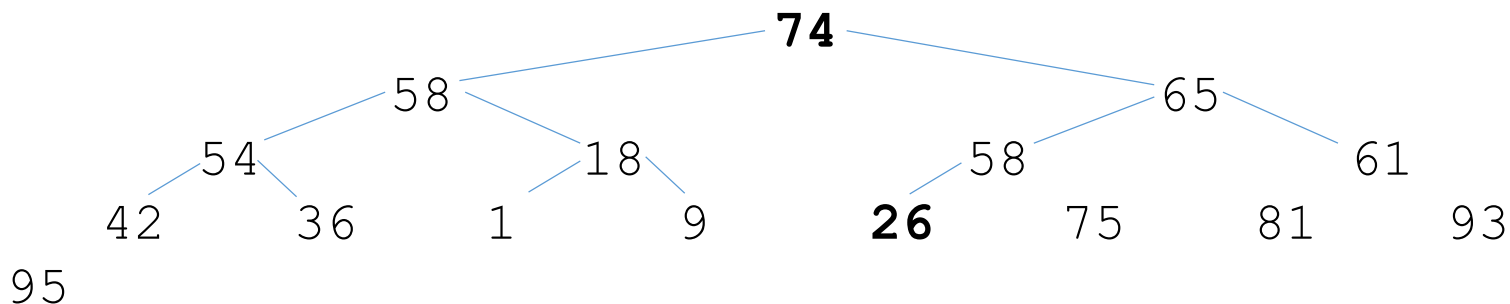
[**74**, **58**, 65, **54**, 18, 58, 61, 42, 36, 1, 9, 26, 75, 81, 93, 95]



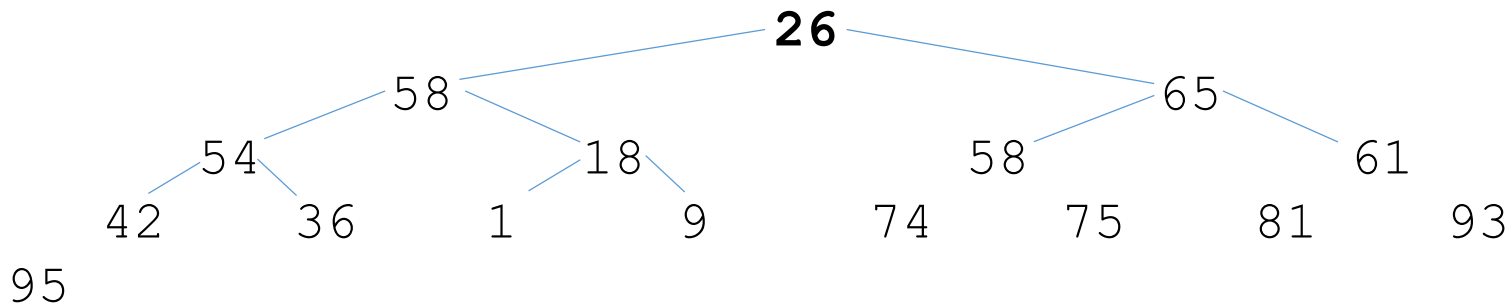
Heap Sort

■ Výběr prvku a kontrola haldy

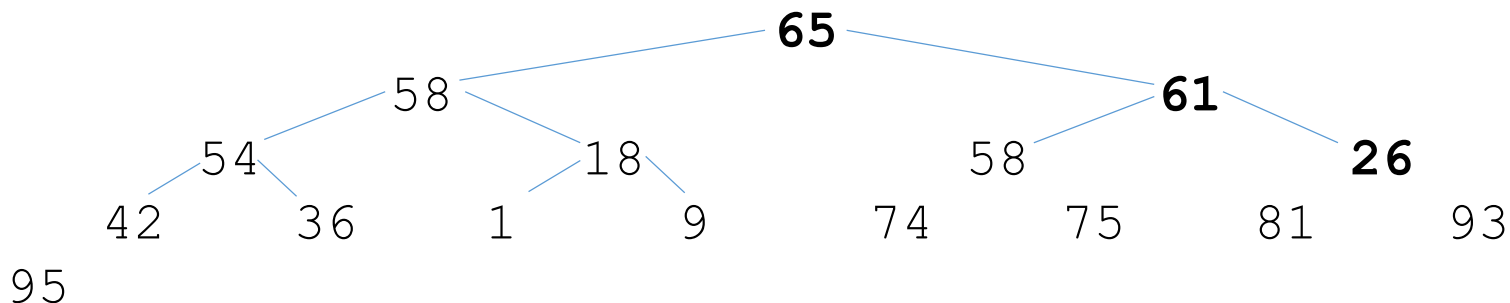
[**74**, 58, 65, 54, 18, 58, 61, 42, 36, 1, 9, **26**, 75, 81, 93, 95]



[26, 58, 65, 54, 18, 58, 61, 42, 36, 1, 9, 74, 75, 81, 93, 95]



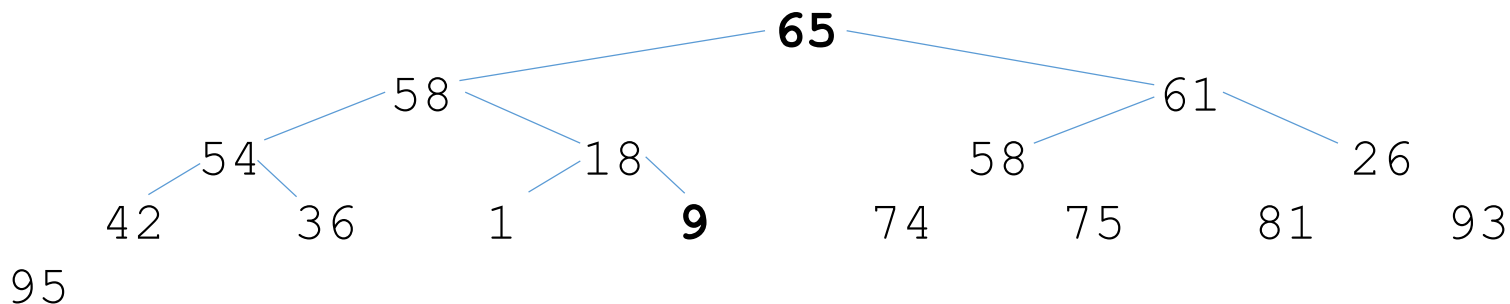
[65, 58, 61, 54, 18, 58, 26, 42, 36, 1, 9, 74, 75, 81, 93, 95]



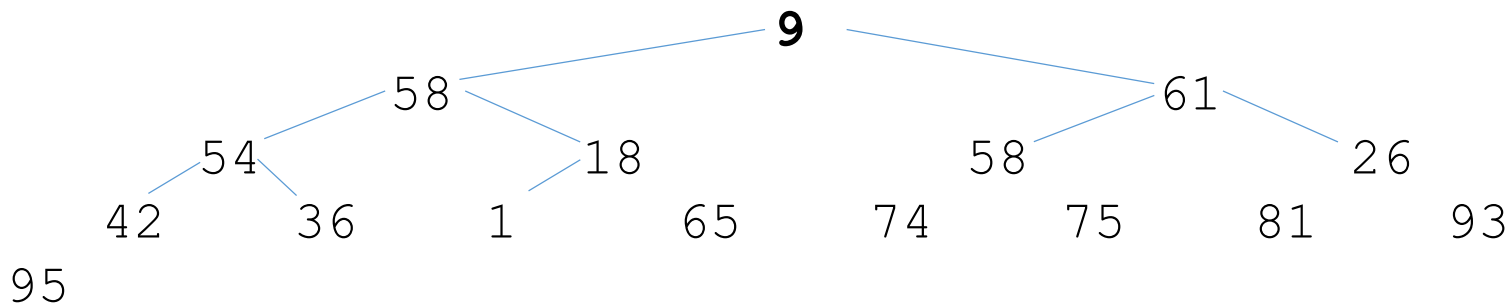
Heap Sort

■ Výběr prvku a kontrola haldy

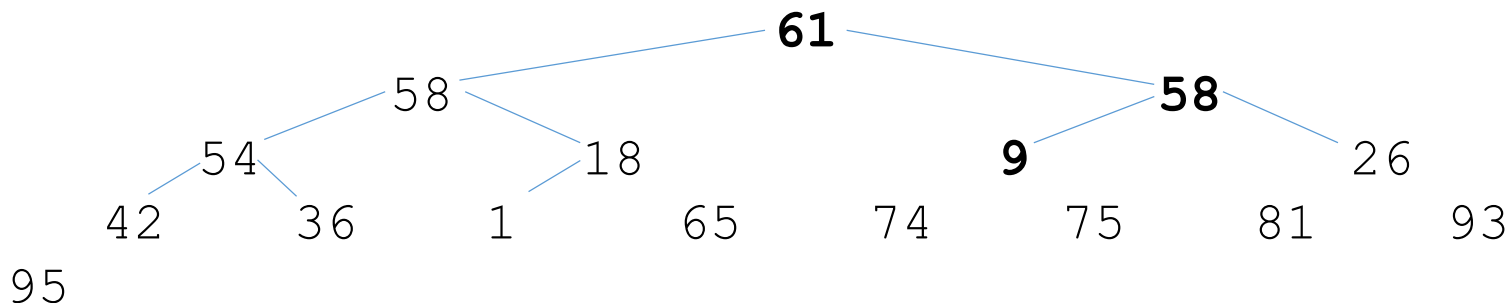
[**65**, 58, 61, 54, 18, 58, 26, 42, 36, 1, **9**, 74, 75, 81, 93, 95]



[65, 58, 61, 54, 18, 58, 26, 42, 36, 1, **9**, 74, 75, 81, 93, 95]



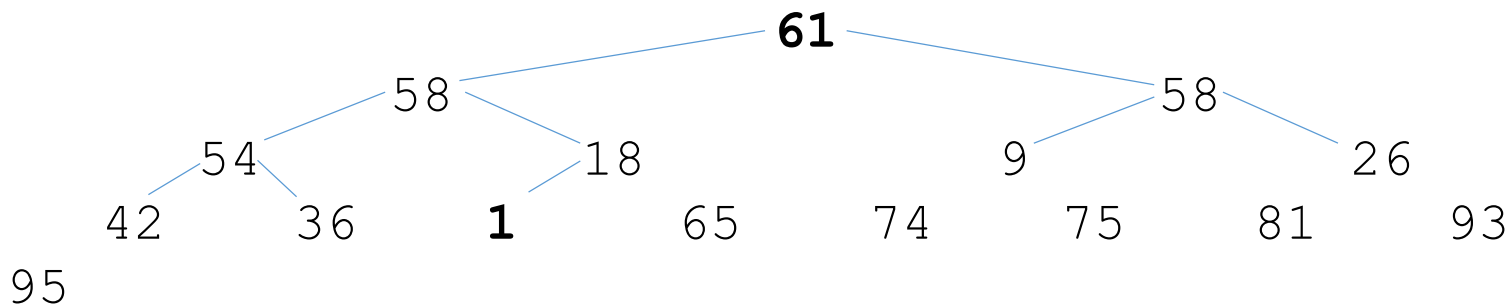
[**61**, 58, **58**, 54, 18, **9**, 26, 42, 36, 1, 65, 74, 75, 81, 93, 95]



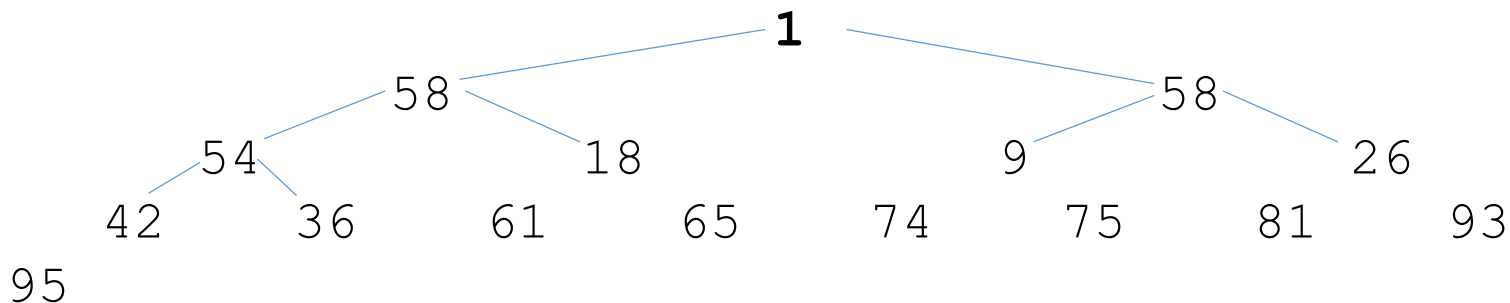
Heap Sort

■ Výběr prvku a kontrola haldy

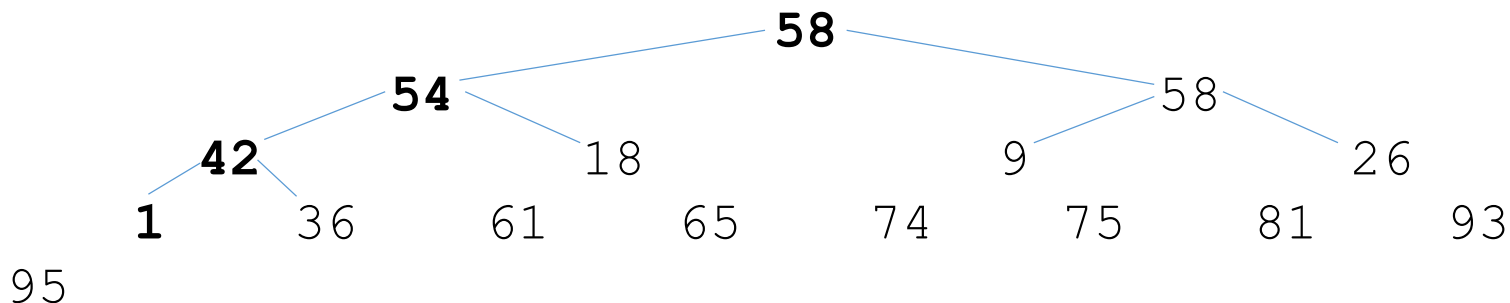
[**61**, 58, 58, 54, 18, 9, 26, 42, 36, **1**, 65, 74, 75, 81, 93, 95]



[**1**, 58, 58, 54, 18, 9, 26, 42, 36, 61, 65, 74, 75, 81, 93, 95]



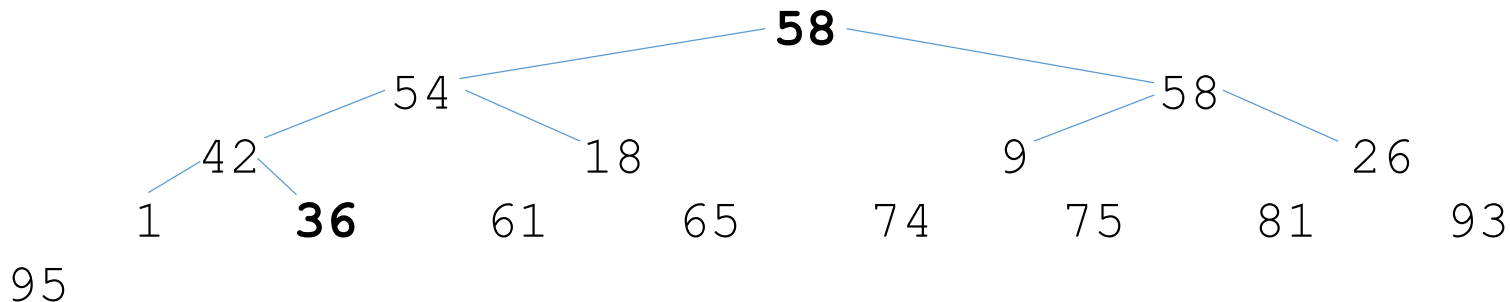
[**58**, **54**, 58, **42**, 18, 9, 26, **1**, 36, 61, 65, 74, 75, 81, 93, 95]



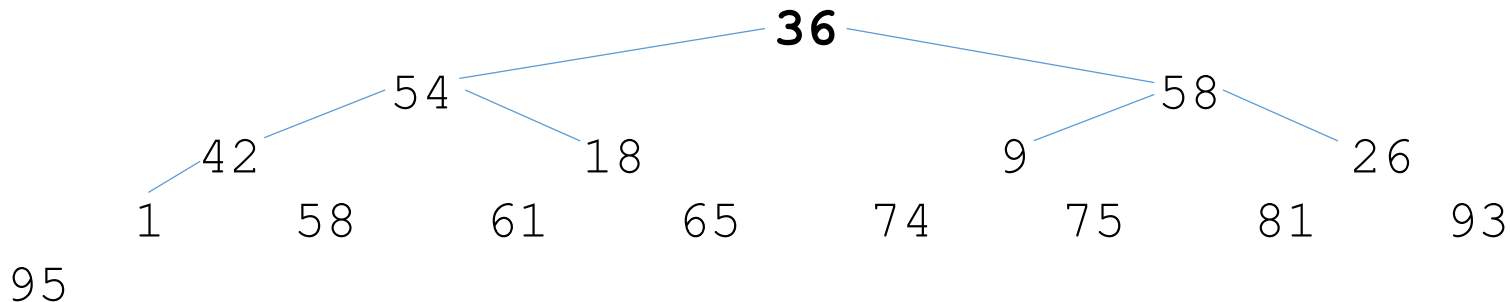
Heap Sort

■ Výběr prvku a kontrola haldy

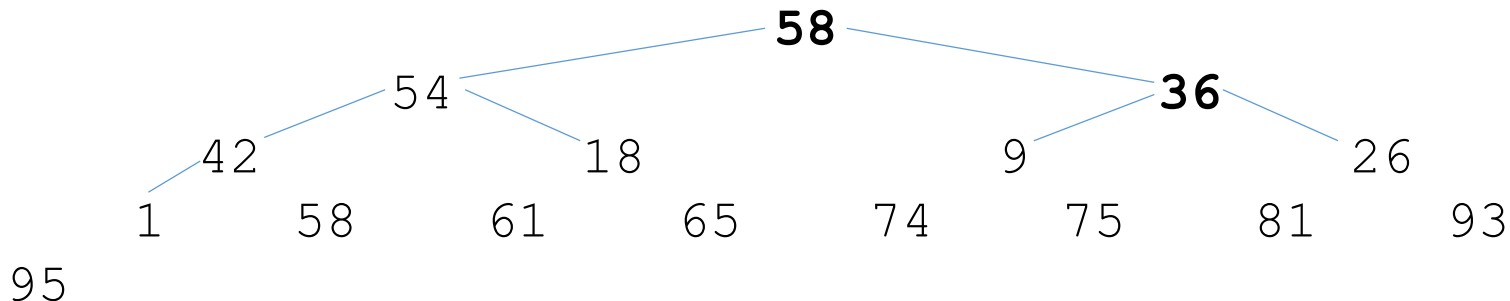
[**58**, 54, 58, 42, 18, 9, 26, 1, **36**, 61, 65, 74, 75, 81, 93, 95]



[**36**, 54, 58, 42, 18, 9, 26, 1, 58, 61, 65, 74, 75, 81, 93, 95]



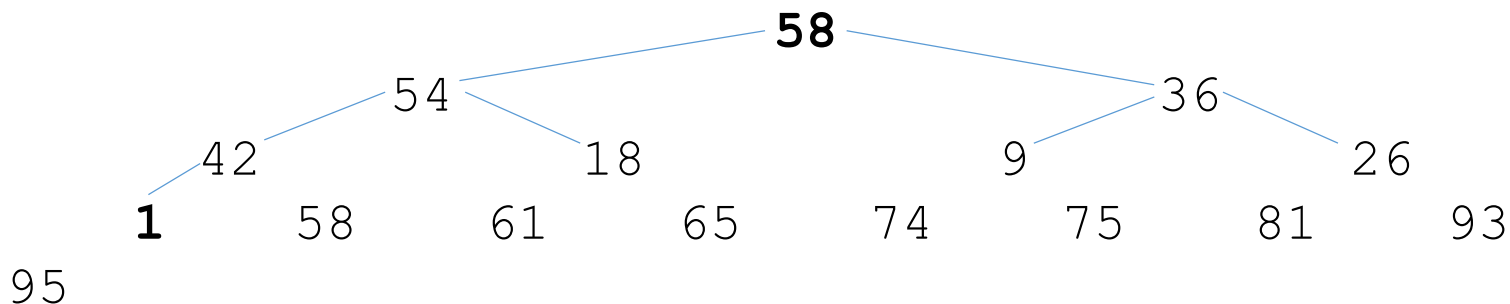
[**58**, 54, **36**, 42, 18, 9, 26, 1, 58, 61, 65, 74, 75, 81, 93, 95]



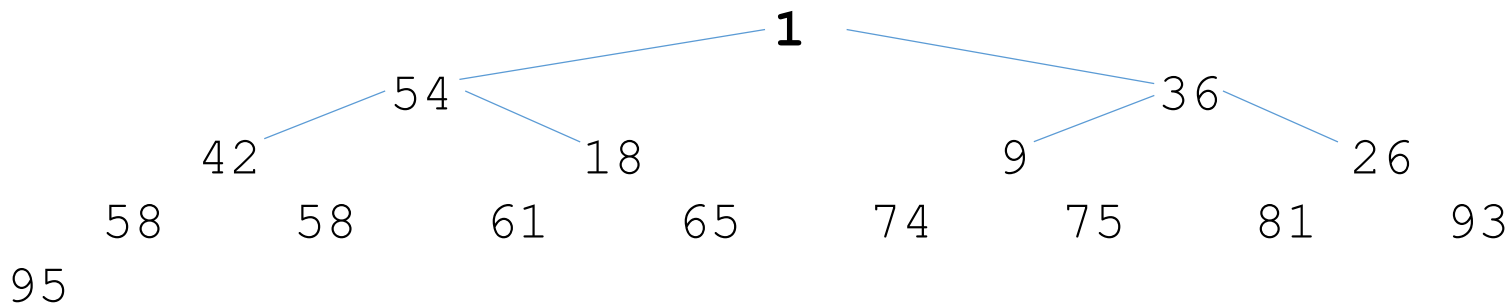
Heap Sort

■ Výběr prvku a kontrola haldy

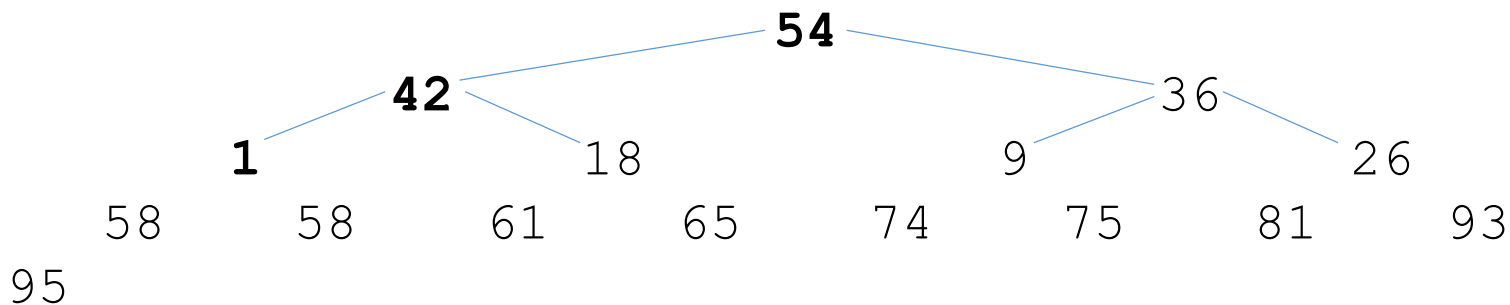
[**58**, 54, 36, 42, 18, 9, 26, **1**, 58, 61, 65, 74, 75, 81, 93, 95]



[1, 54, 36, 42, 18, 9, 26, 58, 58, 61, 65, 74, 75, 81, 93, 95]



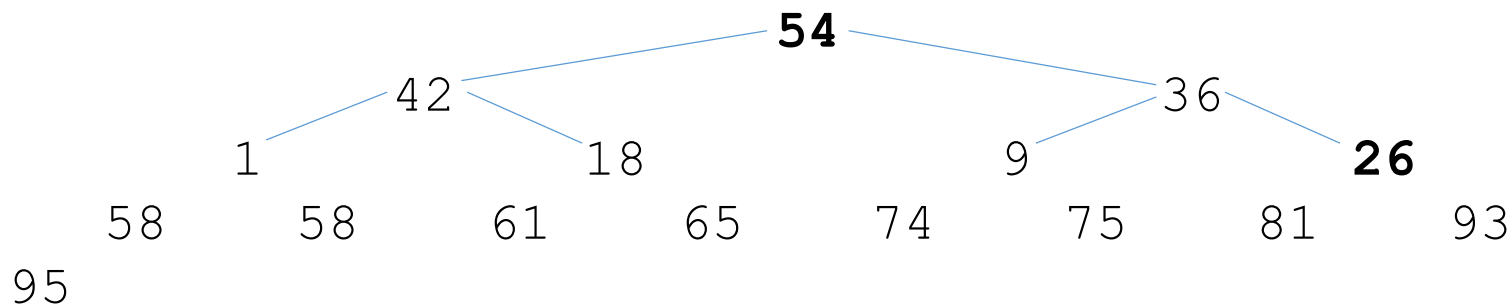
[54, 42, 36, 1, 18, 9, 26, 58, 58, 61, 65, 74, 75, 81, 93, 95]



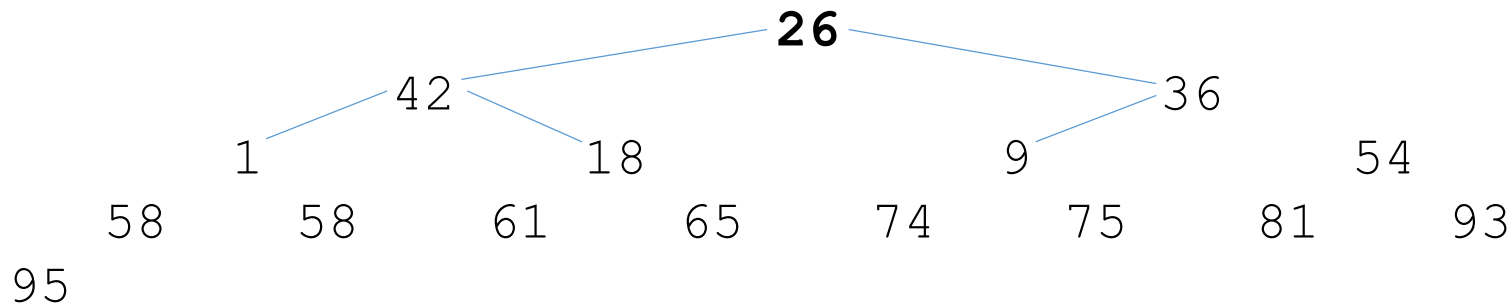
Heap Sort

■ Výběr prvku a kontrola haldy

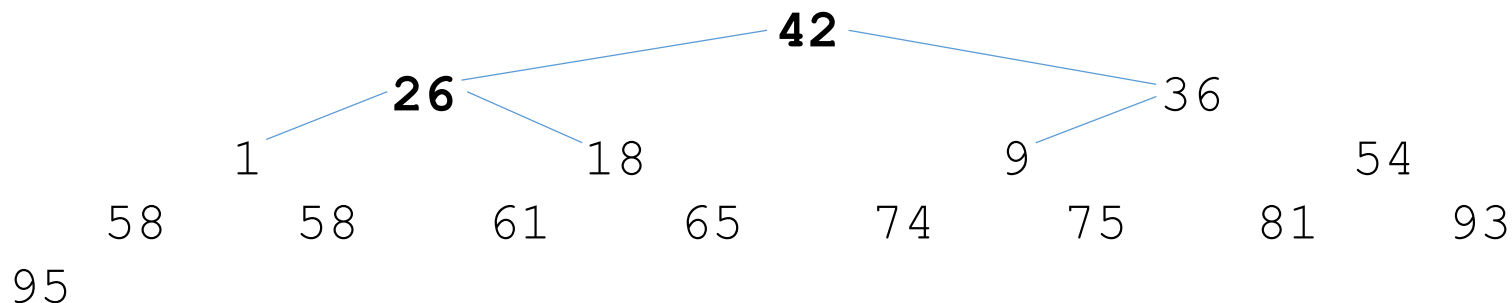
[**54**, 42, 36, 1, 18, 9, **26**, 58, 58, 61, 65, 74, 75, 81, 93, 95]



[**26**, 42, 36, 1, 18, 9, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



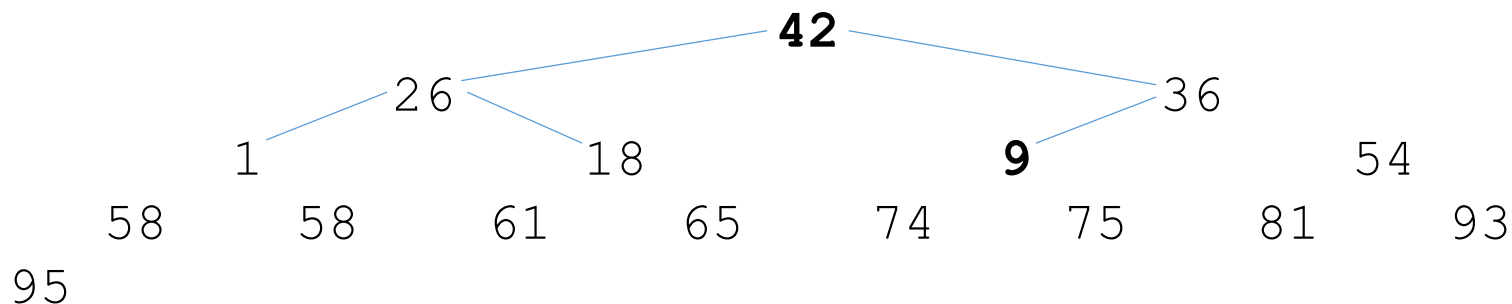
[**42**, **26**, 36, 1, 18, 9, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



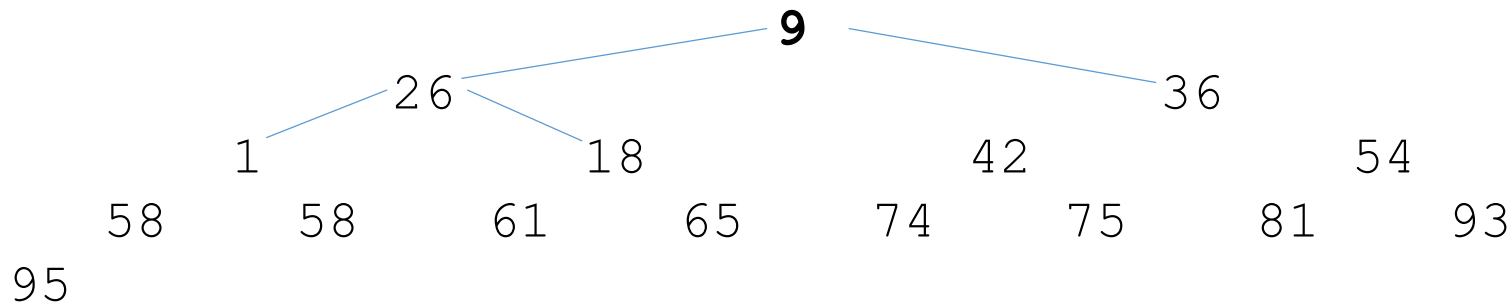
Heap Sort

■ Výběr prvku a kontrola haldy

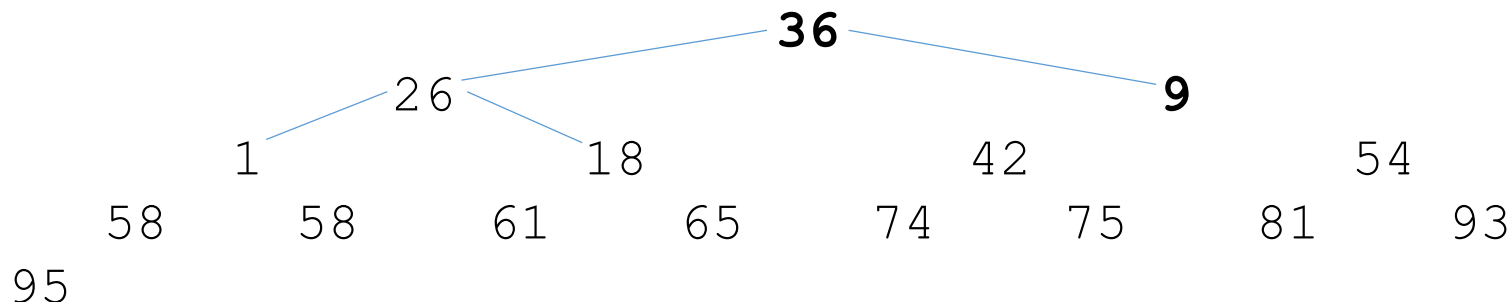
[**42**, 26, 36, 1, 18, **9**, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



[**9**, 26, 36, 1, 18, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



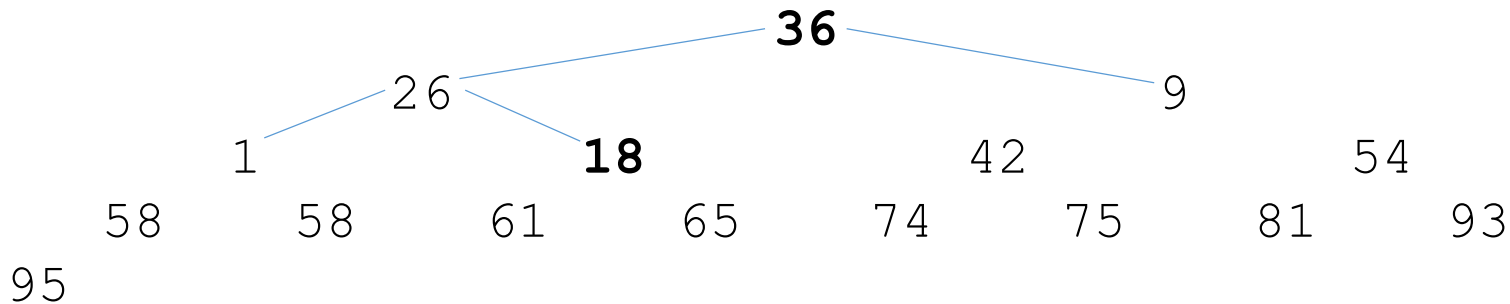
[**36**, 26, **9**, 1, 18, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



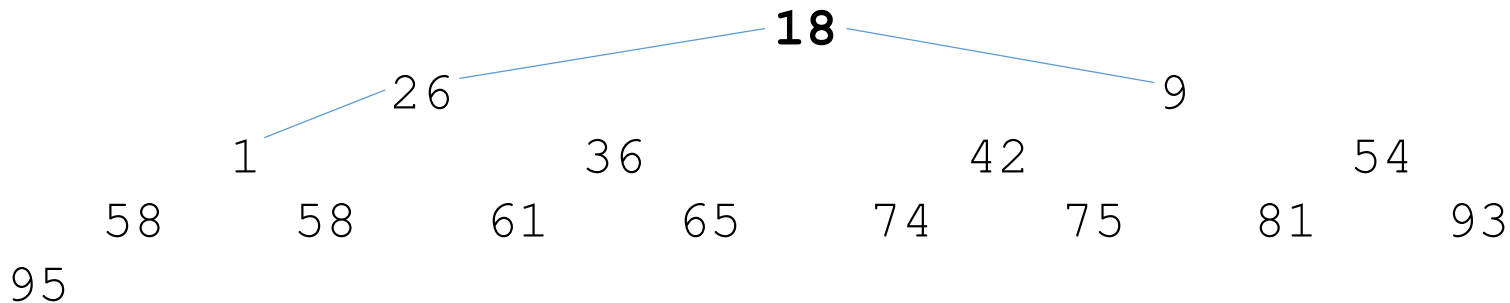
Heap Sort

■ Výběr prvku a kontrola haldy

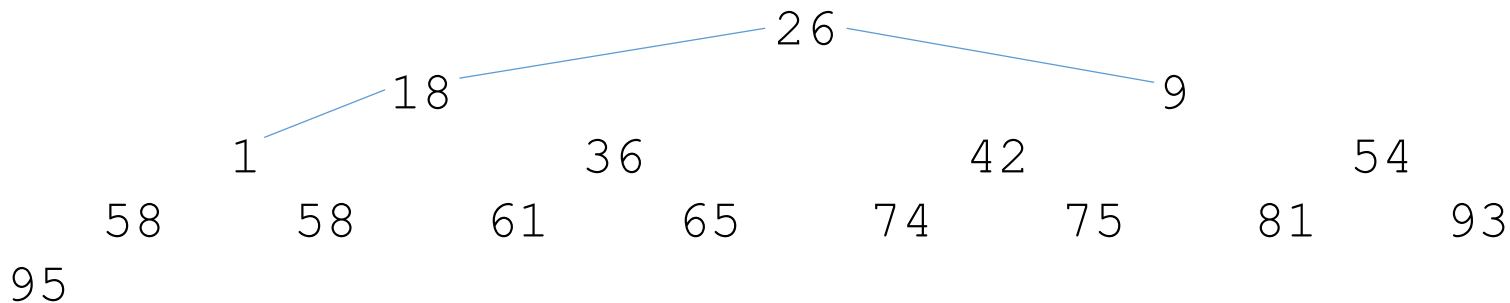
[**36**, 26, 9, 1, **18**, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



[18, 26, 9, 1, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



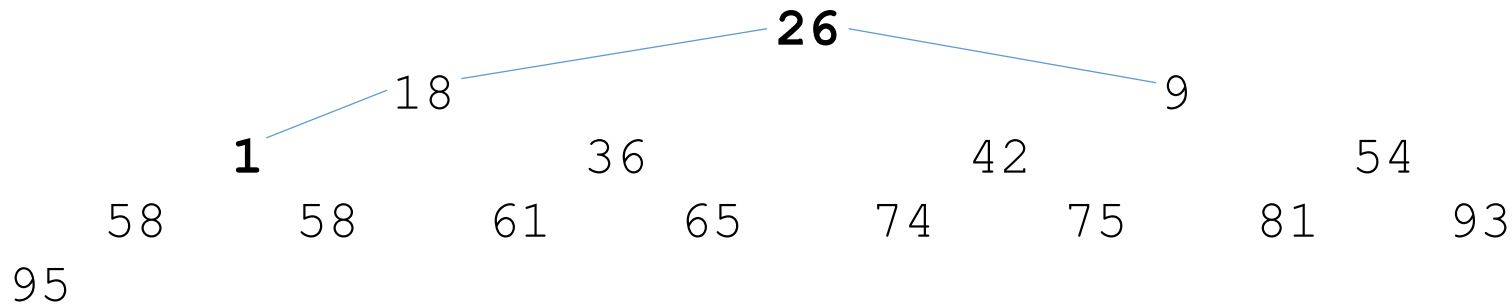
[26, 18, 9, 1, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



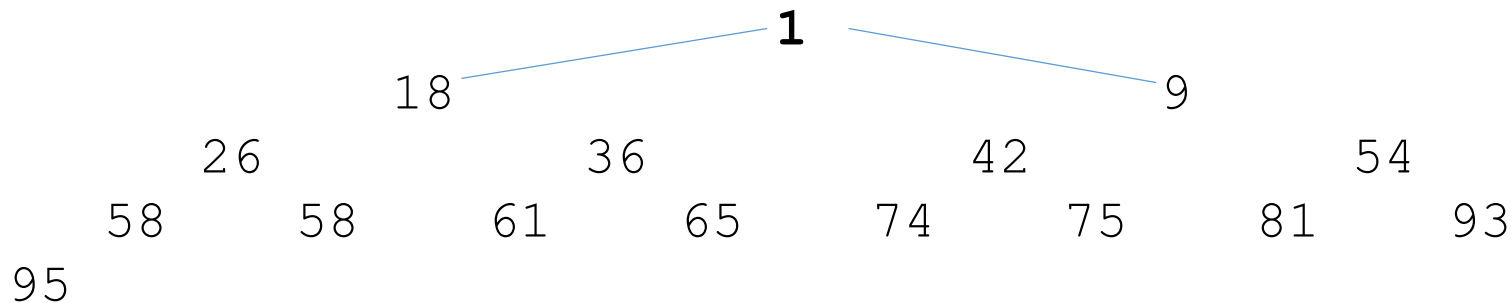
Heap Sort

■ Výběr prvku a kontrola haldy

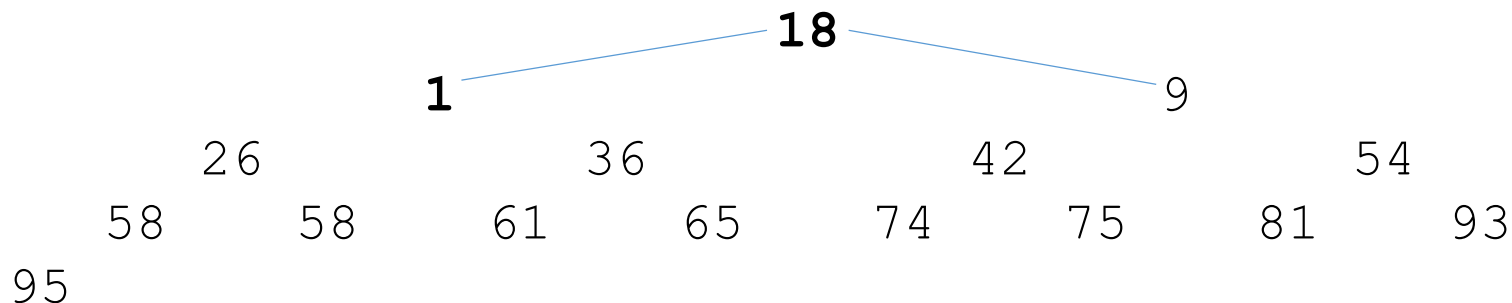
[**26**, 18, 9, **1**, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



[**1**, 18, 9, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



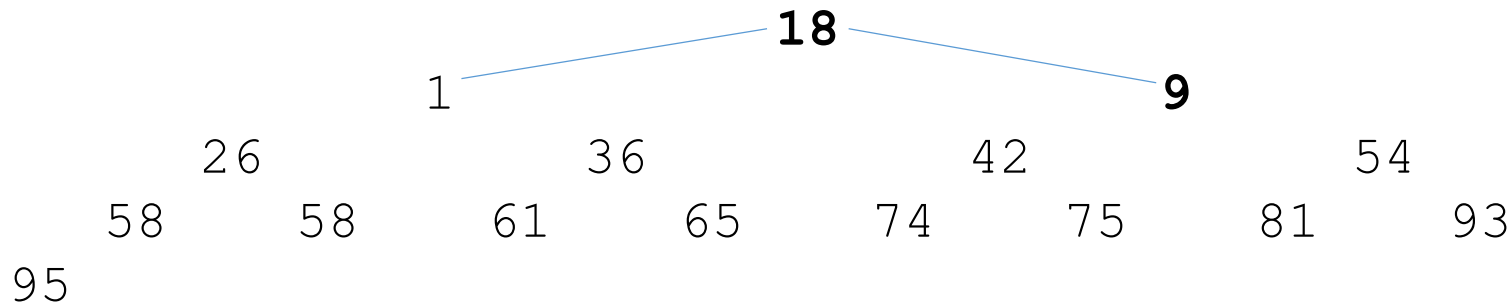
[**18**, **1**, 9, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



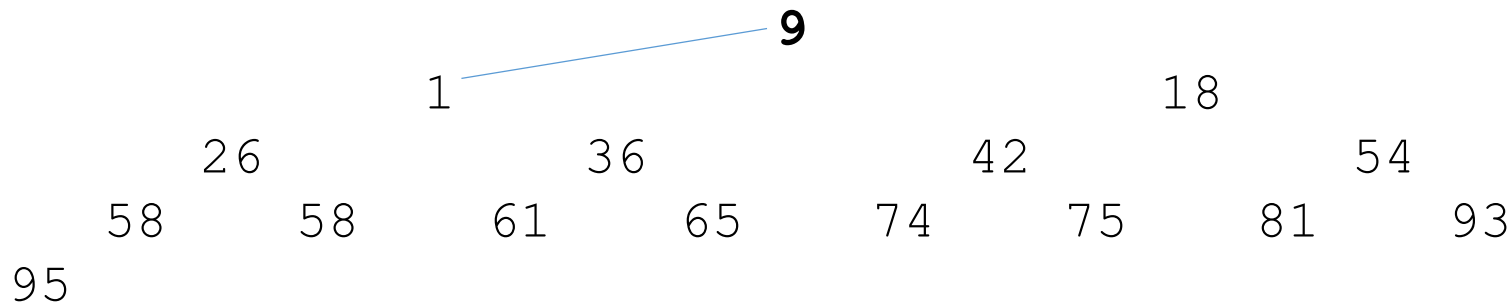
Heap Sort

■ Výběr prvku a kontrola haldy

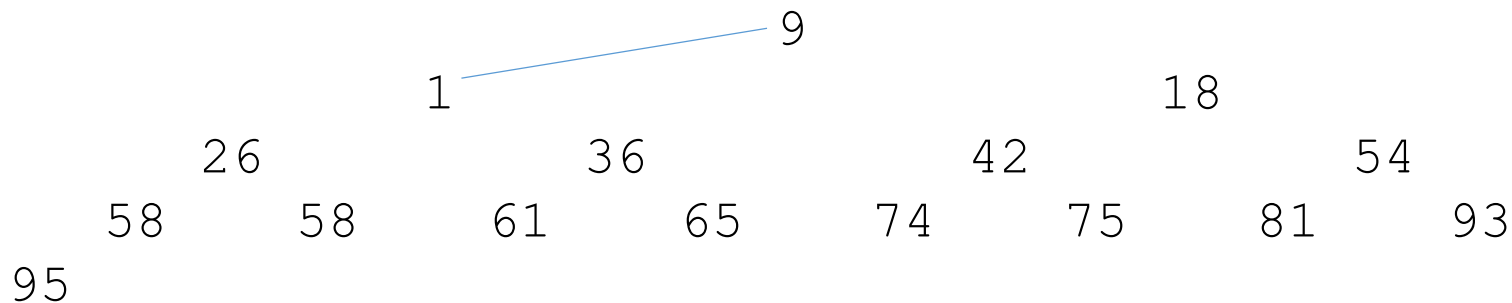
[**18**, 1, **9**, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



[**9**, 1, 18, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



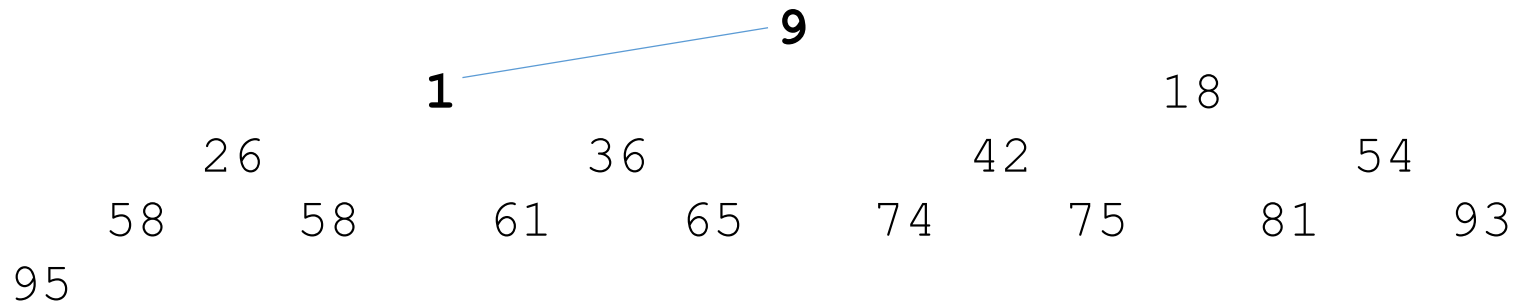
[**9**, 1, 18, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



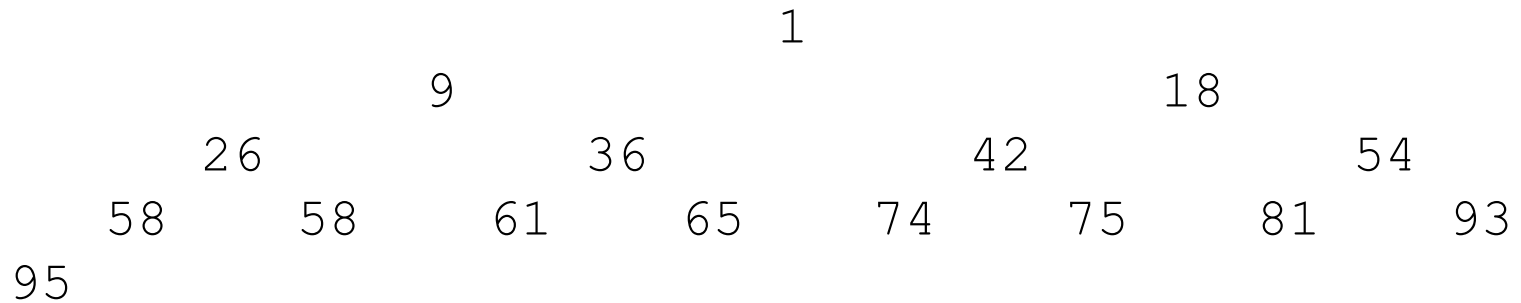
Heap Sort

■ Výběr prvku a kontrola haldy

[9, 1, 18, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



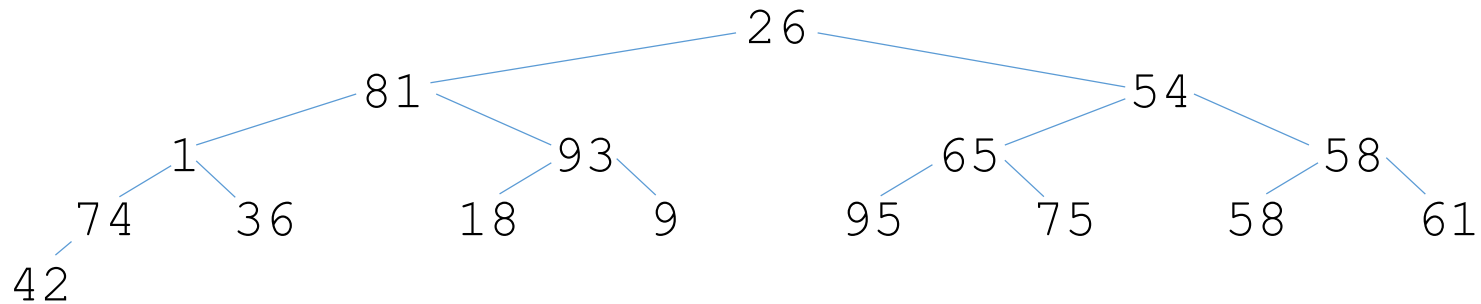
[1, 9, 18, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



Heap Sort

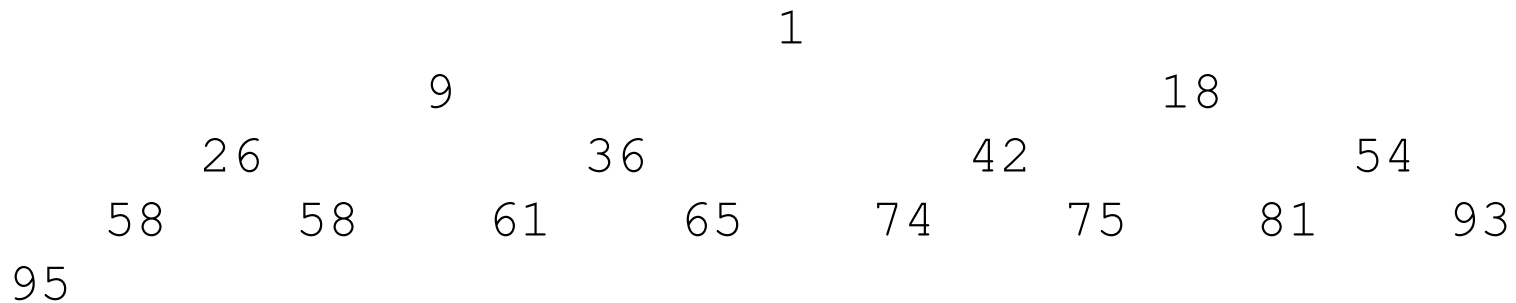
■ Vstup

[26, 81, 54, 1, 93, 65, 58, 74, 36, 18, 9, 95, 75, 58, 61, 42]



■ Výstup

[1, 9, 18, 26, 36, 42, 54, 58, 58, 61, 65, 74, 75, 81, 93, 95]



Heap Sort

tvorba haldy



```
def heapSort(array):  
    for start in range((len(array)-2)//2, -1, -1):  
        movedown(array, start, len(array)-1)
```

```
    for end in range(len(array)-1, 1, -1):  
        array[end], array[0] = array[0], array[end]  
        movedown(array, 0, end - 1)
```

výběr prvku



kontrola haldy



```
def movedown(array, start, end):  
    root = start  
    while True:  
        child = root * 2 + 1  
        if child > end: break  
        if child + 1 <= end and array[child] < array[child + 1]:  
            child += 1  
        if array[root] < array[child]:  
            array[root], array[child] = array[child], array[root]  
            root = child  
    else:  
        break
```

Heap Sort

 tvorba haldy

```
def heapSort(array):
```

```
    myHeap = heapDataType(array) - 1):
```

```
    while myHeap.size():
```

```
        max = myHeap.removeMax()
```

 výběr prvku

```
        result.add(max)
```

kontrola haldy

```
    while True:
```

```
        child = root * 2 + 1
```

```
        if child > end: break
```

```
        if child + 1 <= end and array[child] < array[child + 1]:
            child += 1
```

```
        if array[root] < array[child]:
```

```
            array[root], array[child] = array[child], array[root]
```

```
            root = child
```

```
    else:
```

```
        break
```

Heap Sort

- Vhodný pro kombinaci s binárním vyhledáváním
 - Použití jako „záložní“ algoritmus pro quick sort
 - Garantovaná „worst-case“ náročnost
 - Využívá záměn v rámci vstupní datové struktury (nepotřebuje pomocné uložení dat)
 - Vhodný i pro inkrementální řazení
-
- Zkuste si implementovat pomocí abstraktního datového typu

Řazení

- Porovnání algoritmů

- Časová náročnost (best, average, worst)

Porovnávací algoritmy nemohou být lepší než $n \cdot \log n$

- Paměťová náročnost (memory)

- Stabilita řazení

- Vše v „big O“ notaci

Více se tomuto tématu budeme věnovat v druhé půlce semestru

Algoritmus	Best	Average	Worst	Memory	Stable
Heap sort	$n \log n$	$n \log n$	$n \log n$	1	No

Základy algoritmizace

- Dnes:
 - Abstraktní datové typy
 - Zásobník
 - Fronta
 - Řazení pomocí haldy

Příště spojové seznamy

Základy algoritmizace

- Dodatečné zdroje:

- Implementing a Stack in Python

- <http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaStackinPython.html>

- Implementing a Queue in Python

- <http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaQueueinPython.html>