

```
270     childpos = rightpos
271     # Move the smaller child up.
272     heap[pos] = heap[childpos]
273     pos = childpos
274     childpos = 2*pos + 1
```

Algoritmy a programování

Abstraktní datové struktury

Stromy, binární stromy

```
283 # Follow the path to the root, moving parents down until finding a place
284 # newitem fits.
```

```
285 while pos > startpos: Vojtěch Vonásek
```

```
286     parentpos = (pos - 1) >> 1
```

```
287     parent = heap[parentpos]
```

```
288     if parent < newitem:
```

```
289         heap[parentpos] = newitem
```

```
290         pos = parentpos
```

```
291         continue
```

```
292     break
```

```
293 heap[pos] = newitem
```

```
294
```

```
295 def _siftup_max(heap, pos):
```

```
296     'Maxheap variant of _siftup'
```

```
297     endpos = len(heap)
```

```
298     startpos = pos
```

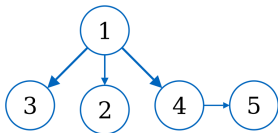
```
299     newitem = heap[pos]
```

```
300     # Bubble up the larger child until hitting a leaf.
```

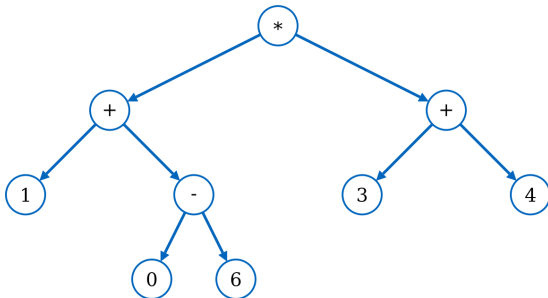
```
301     childpos = 2*pos + 1 # leftmost child position
```

```
302     while childpos < endpos:
```

- Strom je acyklický a souvislý graf $G = (V, E)$
- Jeden uzel je označen jako kořen
- Každý uzel může mít až $m \geq 0$ potomků
- Každý uzel (kromě kořene) má právě jeden vstup
- Přidáním (jakékoliv) hrany vznikne cyklus
- Odebráním (jakékoliv) hrany přestane být graf souvislý
- List (leaf): uzel, který nemá ani jednoho následovníka (potomka)



- Výraz $(1 + (0 - 6)) * (3 + 4)$ reprezentovaný stromem

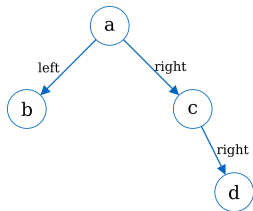
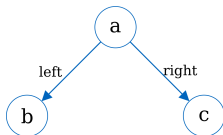


Binární strom

- Souvislý acyklický graf
- Každý uzel má nejvýše dva potomky

Úplný binární strom s n uzly

- každý uzel má 0 nebo 2 potomky
- Počet uzlů v hloubce h je 2^h
- $n = \sum_{i=1}^h 2^i = 2^{h+1} - 1$
- Všechny listy mají hloubku $h = \log_2(n + 1) - 1$
- Počet listů je $(n+1)/2$, počet vnitřních uzlů je $(n - 1)/2$

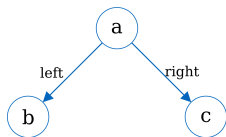


- Uzel je objekt, obsahuje data, referenci na levý a pravý podstrom
- Pokud je podstrom prázdný, je reference None

```
1 class Node:
2     def __init__(self, data=None, left=None, right=None):
3         self.data = data
4         self.left = left
5         self.right = right
```

```
1 from tree import *
2
3 tree = Node("a", Node("b"), Node("c"))
4 print(tree.data)
5 print(tree.left)
6 print(tree.left.data)
```

```
a
<tree.Node object at 0x7f35edc23160>
b
```



Procházení stromu

- Systematické zpracování každého uzlu stromu
- Zpracování: např. tisk dat, kopírování uzlů, provedení jiné operace podle typu uzlu
- Několik možností pořadí zpracování uzlů

Preorder

- Navštívíme uzel, pak levý podstrom, pak pravý podstrom

Inorder

- Navštívíme levý podstrom, pak uzel, pak pravý podstrom

Postorder

- Navštívíme levý podstrom, pak pravý podstrom, pak uzel

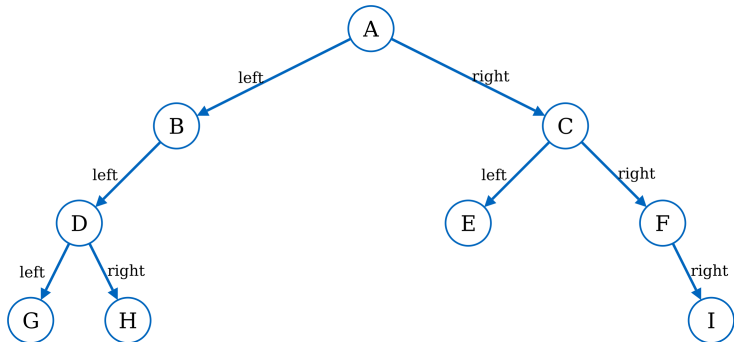
Preorder

- Navštívíme uzel, pak levý podstrom, pak pravý podstrom

```
1 def preorder(node):  
2     if node != None:  
3         print(node.data)      #visit node  
4         preorder(node.left)  #visit left  
5         preorder(node.right) #visit right
```

Preorder

- Navštívíme uzel, pak levý podstrom, pak pravý podstrom
- Uzly jsou navštíveny v pořadí: A B D G H C E F I



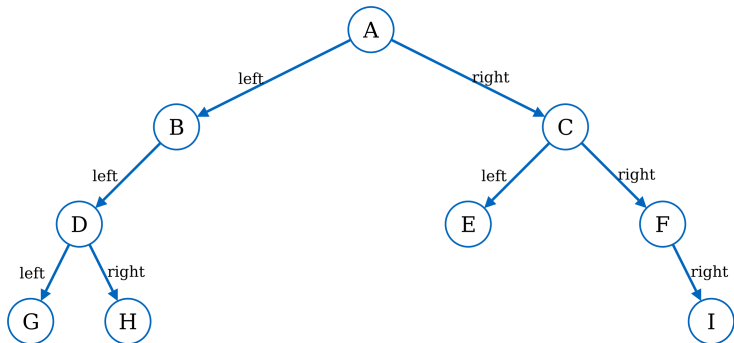
Inorder

- Navštívíme levý podstrom, pak uzel, pak pravý podstrom

```
1 def inorder(node):  
2     if node != None:  
3         inorder(node.left) #visit left  
4         print(node.data)  #visit node  
5         inorder(node.right) #visit right
```

Inorder

- Navštívíme levý podstrom, pak uzel, pak pravý podstrom
- Uzly jsou navštíveny v pořadí: G D H B A E C F I



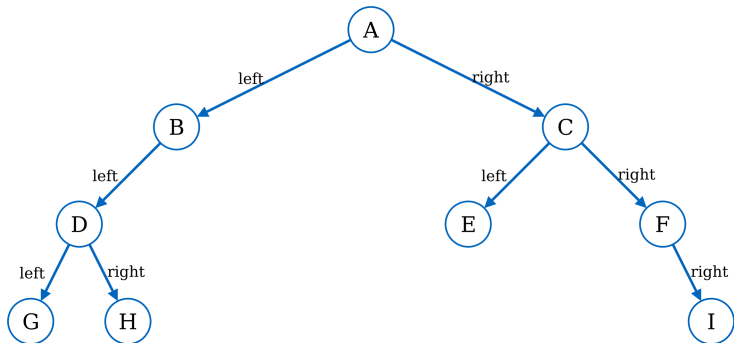
Postorder

- Navštívíme levý podstrom, pak pravý podstrom, pak uzel

```
1 def postorder(node):  
2     if node != None:  
3         postorder(node.left) #visit left  
4         postorder(node.right) #visit right  
5         print(node.data) #visit node
```

Postorder

- Navštívíme levý podstrom, pak pravý podstrom, pak uzel
- Uzly jsou navštíveny v pořadí: G H D B E I F C A



- Převod datových položek stromu na string

```
1 def preorderString(node):
2     if node != None:
3         return (str(node.data) + "␣"
4                 + preorderString(node.left) + "␣"
5                 + preorderString(node.right))
6     return ""
```

```
1 def postorderString(node):
2     if node != None:
3         return (postorderString(node.left) + "␣"
4                 + postorderString(node.right) + "␣"
5                 + str(node.data) )
6     return ""
```

```
1 def inorderString(node):
2     if node != None:
3         return (inorderString(node.left) + "␣"
4                 + str(node.data) + "␣"
5                 + inorderString(node.right) )
6     return ""
```

- Převod datových položek stromu na string

```

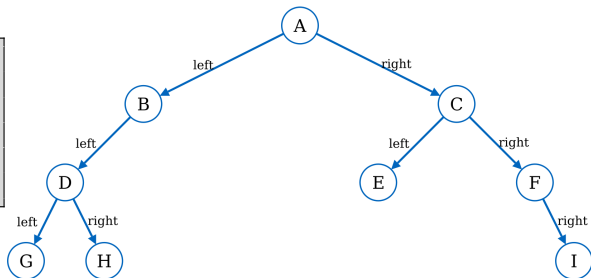
1 from tree import *
2
3 #create the tree
4 node1 = Node("D", Node("G"), Node("H"))
5 node2 = Node("C", Node("E"), Node("F", None, Node("I")))
6 tree = Node("A", Node("B", node1), node2)
7
8 print("Preorder:", preorderString(tree) )
9 print("Postorder:", postorderString(tree) )
10 print("Inorder:", inorderString(tree) )

```

```

Preorder: A B D G H
          C E F I
Postorder: G H D B
          E I F C A
Inorder:  G D H B A
          E C F I

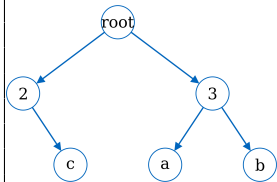
```



- Projdeme strom v inorder postupu, navštívené uzly dáváme do pole

```
1 def inorderList(node):
2     if node != None:
3         return inorderList(node.left) + [node] + inorderList(node.
4             right)
5     return []
```

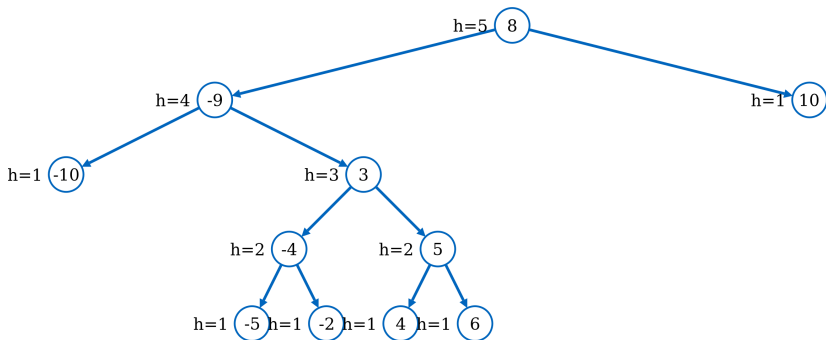
```
1 import tree as TREE
2
3 left = TREE.Node(2, None, TREE.Node("c") )
4 right = TREE.Node(3, TREE.Node("a"), TREE.Node
5     ("b") )
6 tree = TREE.Node("root", left, right)
7
8 nodes = TREE.inorderList(tree)
9 print("Nodes", nodes)
10 for node in nodes: #node is ref to Node
11     print(node.data, end = "␣")
```



```
Nodes [<tree.Node object at 0x7fd1d0c34e80>, <tree.Node object at 0
x7fd1d0c4c160>, <tree.Node object at 0x7fd1d0c1e910>, <tree.
Node object at 0x7fd1d0c1e7f0>, <tree.Node object at 0
x7fd1d0c1ef70>, <tree.Node object at 0x7fd1d0c1ef10>]
2 c root a 3 b
```

- Hloubka v uzlu n je $h(n) = 1 + \max(h(n.left), h(n.right))$

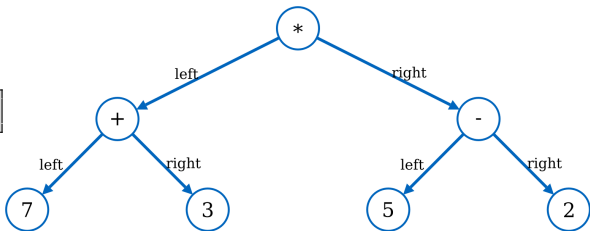
```
1 def countDepth(node):  
2     if node == None:  
3         return 0  
4     return 1 + max(countDepth(node.left), countDepth(node.right))
```



- Repräsentace výrazů: data uzlu obsahují operátor, levý a pravý potomek jsou operandy
- Vyhodnocení výrazů v postorder režimu

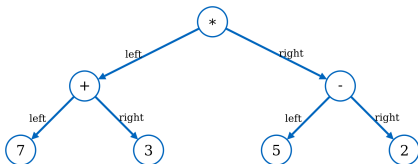
```
1 from tree import *
2 nodePlus = Node("+", Node(7), Node(3))
3 nodeMinus = Node("-", Node(5), Node(2) )
4 tree = Node("*", nodePlus, nodeMinus)
5
6 print( postorderString(tree) )
```

7 3 + 5 2 - *

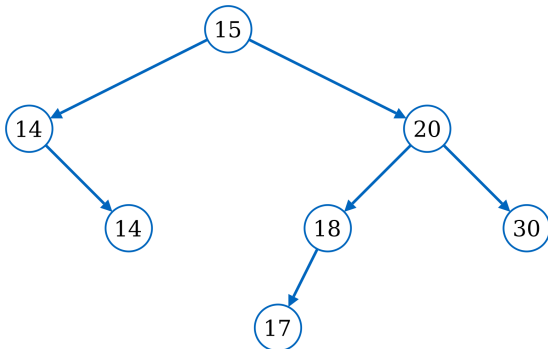


```
1 from tree import *
2 def evaluateTree(node):
3     if node.data=="+":
4         return evaluateTree(node.left) + evaluateTree(node.right)
5     elif node.data=="-":
6         return evaluateTree(node.left) - evaluateTree(node.right)
7     elif node.data=="*":
8         return evaluateTree(node.left) * evaluateTree(node.right)
9     elif node.data == "/":
10        return evaluateTree(node.left) / evaluateTree(node.right)
11    return node.data
12
13 nodePlus = Node("+", Node(7), Node(3))
14 nodeMinus = Node("-", Node(5), Node(2) )
15 tree = Node("*", nodePlus, nodeMinus)
16 print( postorderString(tree) )
17 print( evaluateTree(nodePlus) )
18 print( evaluateTree(tree) )
```

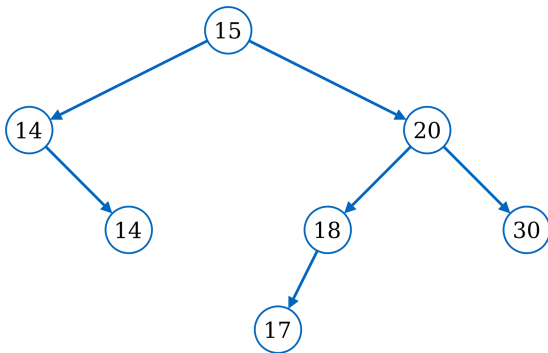
```
7  3 +  5  2 - *
10
30
```



- Složená datová struktura pro rychlé vyhledávání
- Vyhledávání na základě porovnání (předpokládáme $<$ nebo $>$)
- Každý uzel obsahuje klíč
- Klíč v každém uzlu je větší nebo roven než klíče ve všech uzlech v levém podstromu
- Klíč v každém uzlu je menší nebo roven než klíče ve všech uzlech v pravém podstromu

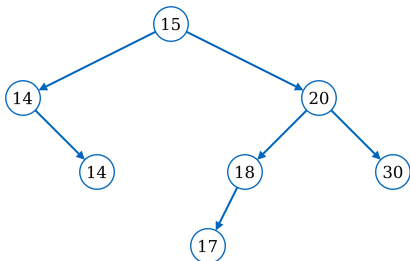


- Vložit prvek
- Smazat prvek
- Vyhledání prvku
 - obsahuje strom hledaný prvek?
 - najít uzel, kde se vyskytuje



Základní operace

- Vložit prvek
- Smazat prvek
- Vyhledání prvku
 - obsahuje strom hledaný prvek?
 - najít uzel, kde se vyskytuje



```
1 class BST:
2     def __init__(self, data=None, left=None, right=None):
3         self.data = data
4         self.left = left
5         self.right = right
```

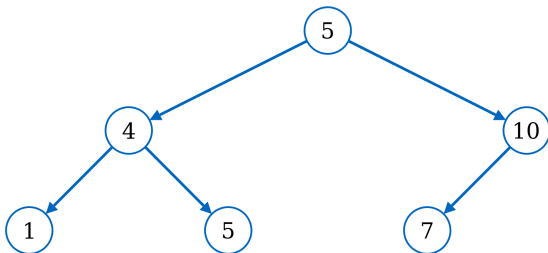
Vytvoření z pole

- Setřídíme pole, z prostředního prvku bude uzel, levý a pravý podstrom z levé a pravé poloviny pole
- Rekurzivní postup

```
1 def buildFromArrayInternal(a):
2     if len(a) == 0:
3         return None
4     if len(a) == 1:
5         return BST(a[0])
6     m = len(a) // 2
7     left = buildFromArrayInternal(a[:m])
8     right = buildFromArrayInternal(a[m+1:])
9     return BST(a[m], left, right)
10
11 def buildFromArray(a):
12     tmp = sorted(a)
13     tree = buildFromArrayInternal(tmp)
14     return tree
```

```
1 from bst import *  
2 from tree import preorderString  
3 a = [4,1,10,5,5,7]  
4 tree = buildFromArray(a)  
5 print( preorderString(tree) )
```

```
5 4 1 5 10 7
```



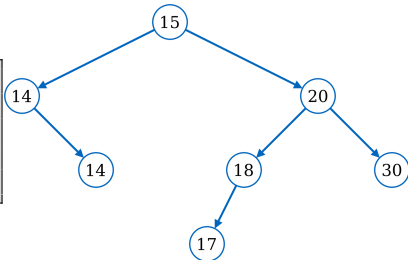
- Pokud uzel obsahuje prvek, vrátíme referenci na uzel
- Jinak prohledáme buď levý nebo pravý podstrom
- Složitost: $\mathcal{O}(\log n)$

```
1 def findNode(node, query):
2     if node:
3         if node.data == query:
4             return node
5         if node.data >= query:
6             return findNode(node.left, query)
7         else:
8             return findNode(node.right, query)
9     return None
```



```
1 from bst import *
2 from tree import inorderString
3 nodeLeft = BST(14, None, BST(14))
4 nodeRight = BST(20, BST(18, BST(17)), BST(30) )
5 tree = BST(15, nodeLeft, nodeRight)
6
7 print( findNode(tree, 17) )
8 print( containsNode(tree, -18) )
9 print( containsNode(tree, 18) )
10 a = findNode(tree, 20)
11 if a:
12     print("Node with key 15:", a.data)
13     print("Subtree of 20 is:", inorderString(a) )
```

```
<bst.BST object at 0x7f8d9d773850>
False
True
Node with key 15: 20
Subtree of 20 is: 17 18 20 30
```

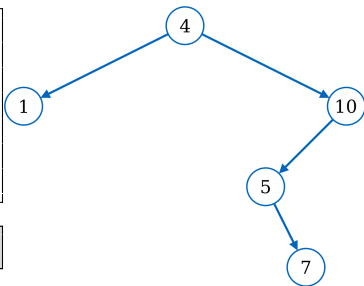


- Rekurzivní hledání vhodného místa, kam vložit (dle velikosti klíče)

```
1 def addKey(node, key):
2     if node == None:
3         return BST(key)
4     if key < node.data:
5         node.left = addKey(node.left, key)
6     elif key > node.data:
7         node.right = addKey(node.right, key)
8     return node
```

```
1 import bst as BST
2 import tree as T
3 tree = None
4 for i in [4,1,10,5,5,7]:
5     tree = BST.addKey(tree, i)
6
7 print( T.inorderString(tree) )
```

```
1 4 5 7 10
```

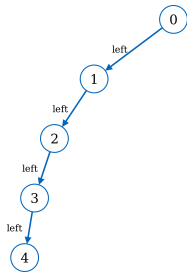


Průměrná složitost

- Dokonale vyvážený strom: rozdíl počtu uzlů podstromů se liší nejvýše o jedna
- Hloubka je $h = \log(n)$
- Vkládání, vyhledávání, mazání: $\mathcal{O}(h) = \mathcal{O}(\log n)$

Nejhorší případ

- Strom je nevyvážený
- Nejhorší případ: degenerovaný strom, hloubka $h = n - 1$
- Vkládání, vyhledávání, mazání: $\mathcal{O}(n)$



Vyvažování

- Cílem je mít stejnou velikost levého a pravého podstromu (rozdíl max. jedna)
- Operace zápisu do stromu (vkládání a mazání) se upraví o operaci vyvažování
- Poskytují nejlepší složitosti operací vkládání/vyhledání/mazání
- Vyvážené stromy: např. AVL-tree a další

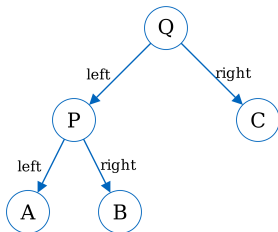
- Faktor vyvážení $BF(node) = h(node.right) - h(node.left)$
- Pokud je $BF(x) < 0$, je uzel x “left-heavy”, obdobně je right-heavy pokud $BF(x) > 0$
- Operace procházení/vyhledávání jsou stejné jako u BST
- Při vkládání uzlu se rekurzivně updatuje hodnota hloubek od listu ke kořeni
- Pokud $BF(x) \notin \{0, -1, 1\}$, provede se vyvážení v uzlu x a aktualizují se hloubky
- Vyvažovací operace: rotace vlevo/vpravo

Rotace vpravo

- Změna uzlu za jeho levého potomka
- Používá se, pokud je uzel left-heavy

```
1 def rotateLeft(node):  
2     pivot = node.left  
3     node.left = pivot.right  
4     pivot.right = node  
5     return pivot
```

Před rotací



Po rotaci vpravo

