Træer i Python

Træer er en grundlæggende datastruktur i datalogi, der bruges til at repræsentere hierarkiske relationer mellem elementer. Et træ består af noder, hvor hver node kan have nul eller flere underordnede noder (børn), og hver node har præcis én overordnet node (forælder), undtagen roden, som ikke har nogen forælder.

Træer bruges i mange forskellige applikationer, herunder:

- Filssystemer: Mapper og filer organiseres ofte i et træstruktur.
- Databaser: B-træer og andre træstrukturer bruges til at indeksere data for hurtig søgning.
- Syntakstræer: Bruges i kompilatorer til at repræsentere strukturen af kildekode.
- Spiludvikling: Scenetree-strukturer bruges til at organisere objekter i en 3D-verden.

Implementering af et simpelt træ i Python

I det følgende eksempel implementerer vi en simpel træstruktur ved hjælp af klasser i Python:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

    def add_child(self, child_node):
        self.children.append(child_node)

    def __repr__(self, level=0):
        ret = "\t" * level + repr(self.value) + "\n"
        for child in self.children:
            ret += child.__repr__(level + 1)
        return ret
```

Vi bemærker, at `TreeNode`-klassen har en `value`-attribut til at gemme nodens værdi og en `children`-liste til at holde styr på dens underordnede noder. Metoden `add_child` bruges til at tilføje en underordnet node, og `__repr__`-metoden giver en pæn repræsentation af træet.

Eksempel på brug af træstrukturen

Herunder eksempler på initialisering og brug af træstrukturen:

```
# Opret roden af træet
root = TreeNode("Rod")
# Tilføj børn til roden
child1 = TreeNode("Barn 1")
child2 = TreeNode("Barn 2")
child3 = TreeNode("Barn 3")
root.add_child(child1)
root.add_child(child2)
```

```
child1.add_child(child3)

print(root)
```

Visualisering af træet

Når vi kører ovenstående kode, får vi følgende output, der viser træets struktur:

```
'Rod'
    'Barn 1'
        'Barn 3'
    'Barn 2'
```

Dette output viser, at "Rod" har to børn: "Barn 1" og "Barn 2", og "Barn 1" har et barn: "Barn 3".

Man kan også benytte mere avancerede moduler til at visualisere træer, såsom `anytree` eller `networkx`, som tilbyder flere funktioner til at arbejde med og visualisere træstrukturer i Python.

Herunder et eksempel på brug af graphviz til at visualisere træet:

```
from graphviz import Digraph

def visualize_tree(node, graph=None):
    if graph is None:
        graph = Digraph()
    graph.node(str(id(node)), node.value)
    for child in node.children:
        graph.edge(str(id(node)), str(id(child)))
        visualize_tree(child, graph)
    return graph
tree_graph = visualize_tree(root)
tree_graph.render('tree', format='png', cleanup=True)
```

Dette vil generere en PNG-fil, der visuelt repræsenterer træstrukturen.

Øvelser

- Lav et træ der repræsenterer en struktur fra den virkelige verden, fx en organisationsstruktur, et filsystem eller en familie. Implementer funktioner til at tilføje noder, fjerne noder og søge efter noder i træet.
- Overfør træstrukturen til en anden repræsentation, fx en liste eller et dictionary, og implementer funktioner til at konvertere mellem disse repræsentationer.

Gennemløb/traversering af træet

Gennemløb/traversering af et træ betyder at besøge alle noder i træet på en bestemt måde. De mest almindelige metoder til trætraversering er:

- **Præorden gennemløb**: Besøg roden først, derefter rekursivt besøg hver underordnet node.
- **Inorden gennemløb**: Besøg den venstre underordnede node, derefter roden, og til sidst den højre underordnede node (bruges primært i binære træer).
- **Postorden gennemløb**: Besøg alle underordnede noder først, og til sidst roden.

Her er et eksempel på præorden gennemløb:

```
def preorder_traversal(node):
    if node is not None:
        print(node.value)
        for child in node.children:
            preorder_traversal(child)
preorder_traversal(root)
```

Dette vil udskrive værdierne i træet i præorden rækkefølge:

```
Rod
Barn 1
Barn 3
Barn 2
```

Inorden og postorden gennemløb kan implementeres på lignende måde, afhængigt af træets struktur og kravene til applikationen. Herunder er eksempel på inorden og postorden gennemløb for binære træer:

```
def inorder_traversal(node):
    if node is not None:
        if len(node.children) > 0:
            inorder_traversal(node.children[0])  # Venstre barn
        print(node.value)
        if len(node.children) > 1:
            inorder_traversal(node.children[1])  # Højre barn
def postorder_traversal(node):
    if node is not None:
        for child in node.children:
            postorder_traversal(child)
        print(node.value)
inorder_traversal(root)
postorder_traversal(root)
```

Disse gennemløbsmetoder er grundlæggende for mange algoritmer, der arbejder med træstrukturer, såsom søgning, sortering og manipulation af data.

Øvelser

- Implementer en funktion til at beregne højden af træet (det længste vej fra roden til et blad).
- Implementer en funktion til at tælle antallet af noder i træet.

- Implementer en funktion til at finde den dybeste node i træet.
- Implementer en funktion til at finde den laveste fælles forfader (LCA) af to noder i træet.

Forskellige typer af træer

Der findes mange forskellige typer af træer, hver med sine egne egenskaber og anvendelser. Nogle af de mest almindelige typer inkluderer:

- **Binære træer**: Hver node har højst to børn. Bruges ofte i søgetræer og heap-strukturer.
- **B-træer**: En generalisering af binære træer, hvor hver node kan have flere børn. Bruges i databaser og filsystemer til effektiv datahåndtering.
- **AVL-træer**: En type selvbalancerende binært søgetræ, der sikrer, at træet forbliver balanceret efter indsættelser og sletninger.
- **Røde-sorte træer**: En anden type selvbalancerende binært søgetræ, der bruger farver til at sikre balancen.
- **Trie (præfiks træ)**: En træstruktur, der bruges til at gemme en dynamisk mængde strenge, hvor noder repræsenterer tegn i strenge. Bruges ofte i ordbøger og autokomplet funktioner.

Disse forskellige trætyper har forskellige fordele og ulemper afhængigt af den specifikke anvendelse, og valget af trætype kan have stor indflydelse på ydeevnen af algoritmer, der arbejder med træstrukturer.

Binære træer

Et binært træ er en træstruktur, hvor hver node har højst to børn, ofte kaldet venstre og højre barn. Binære træer bruges ofte i søgetræer, hvor de muliggør effektiv søgning, indsættelse og sletning af elementer. Her er en simpel implementering af et binært træ i Python:

```
class BinaryTreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def insert(self, value):
        if value < self.value:
            if self.left is None:
                self.left = BinaryTreeNode(value)
            else:
                self.left.insert(value)
        else:
            if self.right is None:
                self.right = BinaryTreeNode(value)
            else:
                self.right.insert(value)

    def inorder_traversal(self):
        elements = []
        if self.left:
            elements += self.left.inorder_traversal()
        elements.append(self.value)
```

```
        if self.right:
            elements += self.right.inorder_traversal()
        return elements
```

Denne `BinaryTreeNode`-klasse har metoder til at indsætte nye værdier og udføre inorden gennemløb af træet. Inorden gennemløb af et binært søgetræ vil returnere elementerne i sorteret rækkefølge.

Øvelser

- Implementer en funktion til at søge efter en bestemt værdi i et binært søgetræ.
- Implementer en funktion til at slette en bestemt værdi fra et binært søgetræ.
- Implementer en funktion til at finde den mindste og største værdi i et binært søgetræ.
- Implementer en funktion til at kontrollere, om et binært træ er et gyldigt binært søgetræ.

B-træer

B-træer er en generalisering af binære træer, hvor hver node kan have flere børn. De bruges ofte i databaser og filsystemer til effektiv datahåndtering, da de kan holde mange nøgler i hver node og dermed reducere træets højde. Her er en simpel implementering af et B-træ i Python:

```
class BTreeNode:
    def __init__(self, t):
        self.t = t  # Minimum degree
        self.keys = []
        self.children = []
        self.leaf = True

    def insert_non_full(self, key):
        i = len(self.keys) - 1
        if self.leaf:
            self.keys.append(0)
            while i >= 0 and key < self.keys[i]:
                self.keys[i + 1] = self.keys[i]
                i -= 1
            self.keys[i + 1] = key
        else:
            while i >= 0 and key < self.keys[i]:
                i -= 1
            i += 1
            if len(self.children[i].keys) == 2 * self.t - 1:
                self.split_child(i)
                if key > self.keys[i]:
                    i += 1
            self.children[i].insert_non_full(key)

    def split_child(self, i):
        t = self.t
        y = self.children[i]
        z = BTreeNode(t)
        z.leaf = y.leaf
```

```
        z.keys = y.keys[t:(2 * t - 1)]
        y.keys = y.keys[0:(t - 1)]
        if not y.leaf:
            z.children = y.children[t:(2 * t)]
            y.children = y.children[0:t]
        self.children.insert(i + 1, z)
        self.keys.insert(i, y.keys.pop())
```

Denne `BTreeNode`-klasse har metoder til at indsætte nøgler og splitte børn, når en node bliver fuld. B-træer sikrer, at træet forbliver balanceret ved at holde nøglerne fordelt jævnt over noderne.

Øvelser

- Implementer en funktion til at søge efter en bestemt nøgle i et B-træ.
- Overvej hvordan filsystemer kan drage fordel af B-træer til at organisere filer og mapper effektivt.

Røde-sorte træer

Røde-sorte træer er en type selvbalancerende binært søgetræ, der bruger farver (rød og sort) til at sikre, at træet forbliver balanceret efter indsættelser og sletninger. Dette sikrer, at de grundlæggende operationer som søgning, indsættelse og sletning kan udføres hurtigt. Her er en simpel implementering af et røde-sorte træ i Python:

```
class RedBlackNode:
    def __init__(self, value, color='red'):
        self.value = value
        self.color = color  # 'red' or 'black'
        self.left = None
        self.right = None
        self.parent = None

class RedBlackTree:
    def __init__(self):
        self.NIL_LEAF = RedBlackNode(value=None, color='black')
        self.root = self.NIL_LEAF
    def insert(self, value):
        new_node = RedBlackNode(value)
        new_node.left = self.NIL_LEAF
        new_node.right = self.NIL_LEAF
        self._bst_insert(new_node)
        self._fix_insert(new_node)
    def _bst_insert(self, new_node):
        y = None
        x = self.root
        while x != self.NIL_LEAF:
            y = x
            if new_node.value < x.value:
                x = x.left
            else:
                x = x.right
        new_node.parent = y
```

```
        if y is None:
            self.root = new_node
        elif new_node.value < y.value:
            y.left = new_node
        else:
            y.right = new_node
```

Denne `RedBlackNode`-klasse repræsenterer en node i et røde-sorte træ, mens `RedBlackTree`-klassen indeholder metoder til indsættelse og opretholdelse af træets egenskaber. Implementeringen af indsættelse og rebalancering er kompleks og kræver flere hjælpefunktioner.

Øvelser

- Røde-sorte træer er komplekse at implementere fuldt ud. Som en øvelse kan du prøve at implementere sletning af noder i et røde-sorte træ.
- Undersøg og forklar de regler, der styrer farvningen og strukturen af røde-sorte træer, og hvordan de sikrer balancen i træet.
- Overvej anvendelser af røde-sorte træer i real-world applikationer.

Projekt med træer

Som et projekt kan du prøve at implementere et filsystemsimulator ved hjælp af træstrukturer. Hver mappe og fil kan repræsenteres som noder i et træ, hvor mapper kan have underordnede mapper og filer som børn. Implementer funktioner til at oprette, slette, flytte og søge efter filer og mapper i træet. Overvej også at tilføje funktioner til at visualisere filsystemets struktur ved hjælp af grafiske biblioteker som `graphviz` eller `anytree`.

Her kommer der kode, som kan bruges til at gennemløbe en mappe struktur:

```
import os
def traverse_directory(path, level=0):
    indent = " " * (level * 4)
    print(f"{indent}{os.path.basename(path)}/")
    for item in os.listdir(path):
        item_path = os.path.join(path, item)
        if os.path.isdir(item_path):
            traverse_directory(item_path, level + 1)
        else:
            print(f"{indent}    {item}")
# Brug funktionen til at gennemløbe en mappe
traverse_directory("/sti/til/din/mappe")
```

Du kan bruger ovenstående kode til at gennemløbe og udskrive strukturen af en mappe på din computer. Bare erstat `"/sti/til/din/mappe"` med stien til den mappe, du vil gennemløbe.