Typy datových struktur - Pole, Spojový seznam, Strom, Fronta, Zásobník, Halda

Pole

Pole je základní datová struktura, která ukládá prvky stejného typu (homogenní) seřazené za sebou blízko v paměti. Jedná se o statickou strukturu - velikost je pevně daná při inicializaci. Definováno jako souvislý blok paměti rozdělený na stejně velké části. Identifikátory jednotlivých prvků jsou indexy, které korelují s jejich umístěním v paměti.

Varianty

Dynamické pole (ArrayList) - automaticky zvětšuje svou kapacitu
Vícerozměrné pole - matice

Operace a jejich složitosti

- přístup k prvku (index) - O(1)
- vyhledání prvku O(n)
- přidání/smazání na konci O(1)
- přidání/smazání uprostřed O(n)

Spojový seznam

Datová struktura složená z uzlů (node), kde každý uzel obsahuje data a odkaz (pointer) na další uzel v sekvenci. K prvkům musíme přistupovat postupně od začátku. Velikost spojového seznamu se může měnit za běhu programu.

Typy spojových seznamů

1. Jednosměrný seznam
    - každý prvek obsahuje právě jednu referenci na následující prvek
    - poslední prvek ukazuje na neplatnou adresu
2. Obousměrný seznam
    - každý prvek obsahuje referenci na následující a předchozí prvek
    - seznam lze procházet obouma směrama
    - následník posledního prvku a předchůdce prvního je nastavena na neplatnou adresu
3. Cyklický seznam
    - poslední prvek má následovníka první prvek
    - při procházení seznamu nikdy nedojdeme na konec

Operace se spojovým seznamem

- Přístup k prvku: O(n)
- Vyhledání prvku: O(n)
- Vložení/odstranění na začátku: O(1)
- Vložení/odstranění na konci: O(n) nebo O(1) s odkazem na konec
- Vložení/odstranění uprostřed: O(n) na nalezení pozice, samotná operace O(1)

```
# Jednoduchá implementace jednosměrného spojového seznamu

class Node:
    def __init__(self, data):
        self.data = data  # Hodnota (data) uzlu
        self.next = None  # Odkaz na další uzel (none = žádný)


class LinkedList:
    def __init__(self):
        self.head = None  # Hlava seznamu (počáteční uzel)

    # Přidání nového uzlu na konec seznamu
    def append(self, data):
        new_node = Node(data)

        # Pokud je seznam prázdný
        if self.head is None:
            self.head = new_node
            return

        # Jinak najdeme poslední uzel
        last = self.head
        while last.next:
            last = last.next

        # A připojíme nový uzel na konec
        last.next = new_node

    # Přidání nového uzlu na začátek seznamu
    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    # Výpis seznamu
    def print_list(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
```

```
# Příklad použití
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
linked_list.append(3)
linked_list.prepend(0)

linked_list.print_list()  # Výstup: 0 -> 1 -> 2 -> 3 -> None
```

Strom

Strom je hierarchická, nelineární datová struktura, která se skládá z uzlů propojených hranami. Formálně je strom definován jako acyklický, souvislý graf. Root je vrcholem a nemá žádného předchůdce. Všechny ostatní uzly mají právě jednoho rodiče a mohou mít libovolný počet potomků. Uzly, které nemají žádné potomky se nazývají listy. Uzly které nejsou kořenem ani listem označujeme jako vnitřní uzly. Hloubka uzlu je délka cesty od kořene k danému uzlu. Výška stromu je maximální hloubka libovolného uzlu.

Podstrom je část stromu tvořená uzlem a všemi jeho potomky. Rekurzivní vlastnost stromů - každý podstrom je sám o sobě platným stromem

Binární strom je prázdný strom nebo vrchol, který má právě dva syny, resp. levý a pravý podstrom. Tyto podstromy jsou binární stromy. Každý uzel ve stromu může tvořit kořen nového podstromu.

<u>Průchody stromem</u>

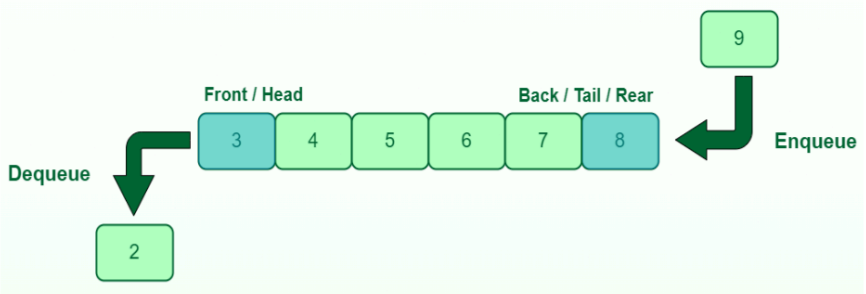
preorder traversal - začíná se od kořene, poté levý podstrom a nakonec pravý podstrom
inorder traversal - nejprve levý strom podstrom, potom kořen a nakonec pravý podstrom
postorder traversal - nejprve levý podstrom, poté pravý podstrom a nakonec kořen.

```
# Jednoduchá implementace obecného stromu
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []  # Seznam potomků

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

```
# Vytvoření stromu
root = TreeNode(1)  # Kořen s hodnotou 1

# Přidání potomků kořene
child1 = TreeNode(2)
child2 = TreeNode(3)
child3 = TreeNode(4)

root.add_child(child1)
root.add_child(child2)
root.add_child(child3)

# Přidání potomků uzlu child1
child1.add_child(TreeNode(5))
child1.add_child(TreeNode(6))

# Jednoduchý výpis stromu - hodnoty kořene a jeho přímých potomků
print(f"Kořen: {root.value}")
print("Potomci kořene:", end=" ")
for child in root.children:
    print(child.value, end=" ")
```

Fronta

Fronta je nelineární datová struktura, která funguje na principu FIFO (first in first out). Funguje podobně jako fronta lidí čekající v řadě - první do, první ven. Má dynamickou velikost



Queue Data Structure

Typy front

Prioritní fronta - prvky jsou seřazeny podle priority

Kruhová fronta - efektivnější využití paměti, po dosažení konce se vrací na začátek, řeší dequeue

Obousměrná fronta - vkládání/odebírání z obou konců

```
# Nejjednodušší implementace fronty
class Queue:
    def __init__(self):
        self.items = []
```

```
    def enqueue(self, item):
        # Přidání prvku na konec
        self.items.append(item)

    def dequeue(self):
        # Odebrání prvku ze začátku
        if self.items:
            return self.items.pop(0)
        return None

    def is_empty(self):
        # Kontrola prázdnosti
        return len(self.items) == 0


# Použití
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())  # Výstup: 1
print(q.dequeue())  # Výstup: 2
print(q.is_empty())  # Výstup: False
```

Zásobník

Zásobník je lineární datová struktura, která funguje na principu LIFO (last in first out). Má dynamickou velikost. Používá se k DFS algoritmům, vyhodnocování výrazů a kontroly syntaxe.

```
# Nejjednodušší implementace zásobníku
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        # Přidání prvku na vrchol
        self.items.append(item)

    def pop(self):
        # Odebrání prvku z vrcholu
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        # Náhled na vrchní prvek
        if not self.is_empty():
            return self.items[-1]
        return None
```

```
    def is_empty(self):
        # Kontrola prázdnosti
        return len(self.items) == 0


# Použití
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.pop())  # Výstup: 3
print(s.pop())  # Výstup: 2
print(s.is_empty())  # Výstup: False
```

Halda

Halda je úpnž binární strom, ve kterém je hodnota v každém uzlu větší nebo rovna (MAXHEAP) nebo menší nebo rovna (MINHEAP) hodnotám v jeho potomcích. Využívá se k implementaci heapsortu, nebo vyhledávání nejkratší cesty.

Max heap - Kořen obsahuje maximální hodnotu v haldě
Min heap - Kořen obsahuje minimální hodnotu v haldě