Structuri de Date și Algoritmi
Structuri de date în alte limbaje

Mihai Nan

Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București

Anul Universitar 2022–2023

Conținutul cursului

Programare orientată pe obiecte

- Presupunem că dorim **să descriem**, utilizând un limbaj de programare, un **obiect** carte.
- În general, o carte poate fi caracterizată prin titlu, autor și editură.

Cum am putea realiza această descriere formală în cadrul unei implementări?

Programare orientată pe obiecte

- Presupunem că dorim **să descriem**, utilizând un limbaj de programare, un **obiect** carte.
- În general, o carte poate fi caracterizată prin titlu, autor și editură.

Cum am putea realiza această descriere formală în cadrul unei implementări?

Dacă descriem acest obiect, tip abstract de date, într-un limbaj de programare structural, spre exemplu limbajul C, atunci vom crea o structură `Carte` împreună cu o serie de funcții cuplate de această structură.

- Cuplajul este realizat prin faptul că orice funcție care operează asupra unei cărți conține în lista sa de parametri o variabilă de tip `Carte`.

Programare orientată pe obiecte

```
1  typedef struct carte {
2      char *titlu, *autor;
3      int nr_pagini;
4  }*Carte;
5
6  void initializare(Carte this, char* titlu, char* autor,
7       int nr_pagini) {
8      this->titlu = strdup(titlu);
9      this->autor = strdup(autor);
10     this->nr_pagini = nr_pagini;
11 }
12
13 void afisare(Carte this) {
14     printf("%s, %s - %d\n", this->autor, this->titlu,
15         this->nr_pagini);
16 }
```

Programare orientată pe obiecte

- Dacă modelăm acest obiect într-un limbaj orientat pe obiecte (în acest caz, Java), atunci vom crea o **clasă** `Carte`.
- Se poate observa în cadrul exemplului de mai jos, că atât datele cât și **metodele** (funcțiile) care operează asupra acestora se găsesc în interiorul aceleiași entități, numită **clasă**.

```
class Carte {
    String nume, autor;
    int nr_pagini;
    public Carte(String nume, String autor, int nr_pagini) {
        this.nume = nume;
        this.autor = autor;
        this.nr_pagini = nr_pagini;
    }
    public Carte() {
        this("Enigma Otiliei", "George Calinescu", 423);
    }
}
```

Programare orientată pe obiecte

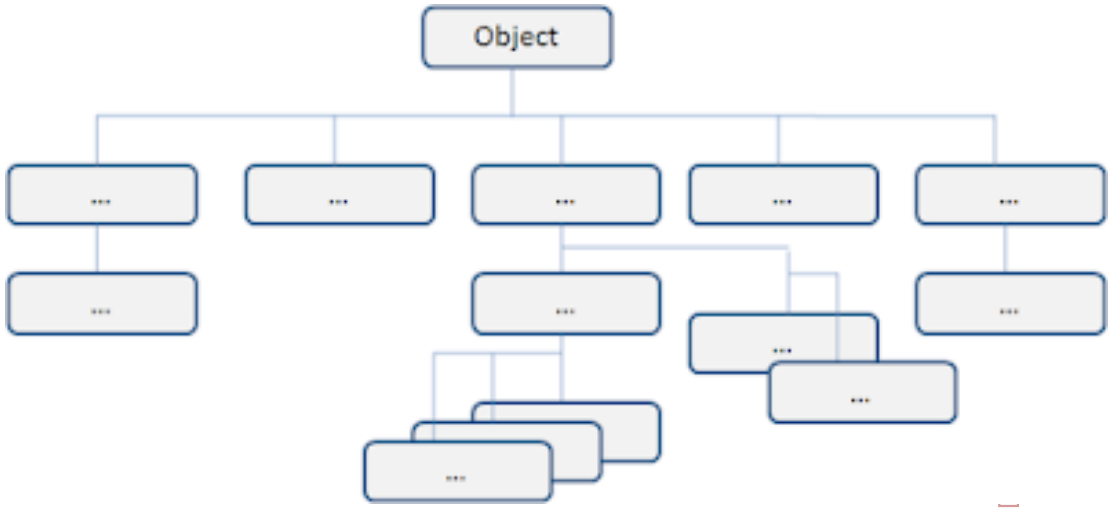
```
class Carte {
    String nume, autor;
    int nr_pagini;
    public Carte(String nume, String autor, int nr_pagini) {
        this.nume = nume;
        this.autor = autor;
        this.nr_pagini = nr_pagini;
    }
    public Carte() {
        this("Enigma Otiliei", "George Calinescu", 423);
    }
    public String toString() {
        String result = "";
        result += this.autor + ", " + this.nume;
        result += " - " + this.nr_pagini;
        return result;
    }
    public static void main(String args[]) {
        Carte carte;
        carte = new Carte("Poezii", "Mihai Eminescu", 256);
        System.out.println(carte.toString());
    }
}
```

Programare orientată pe obiecte

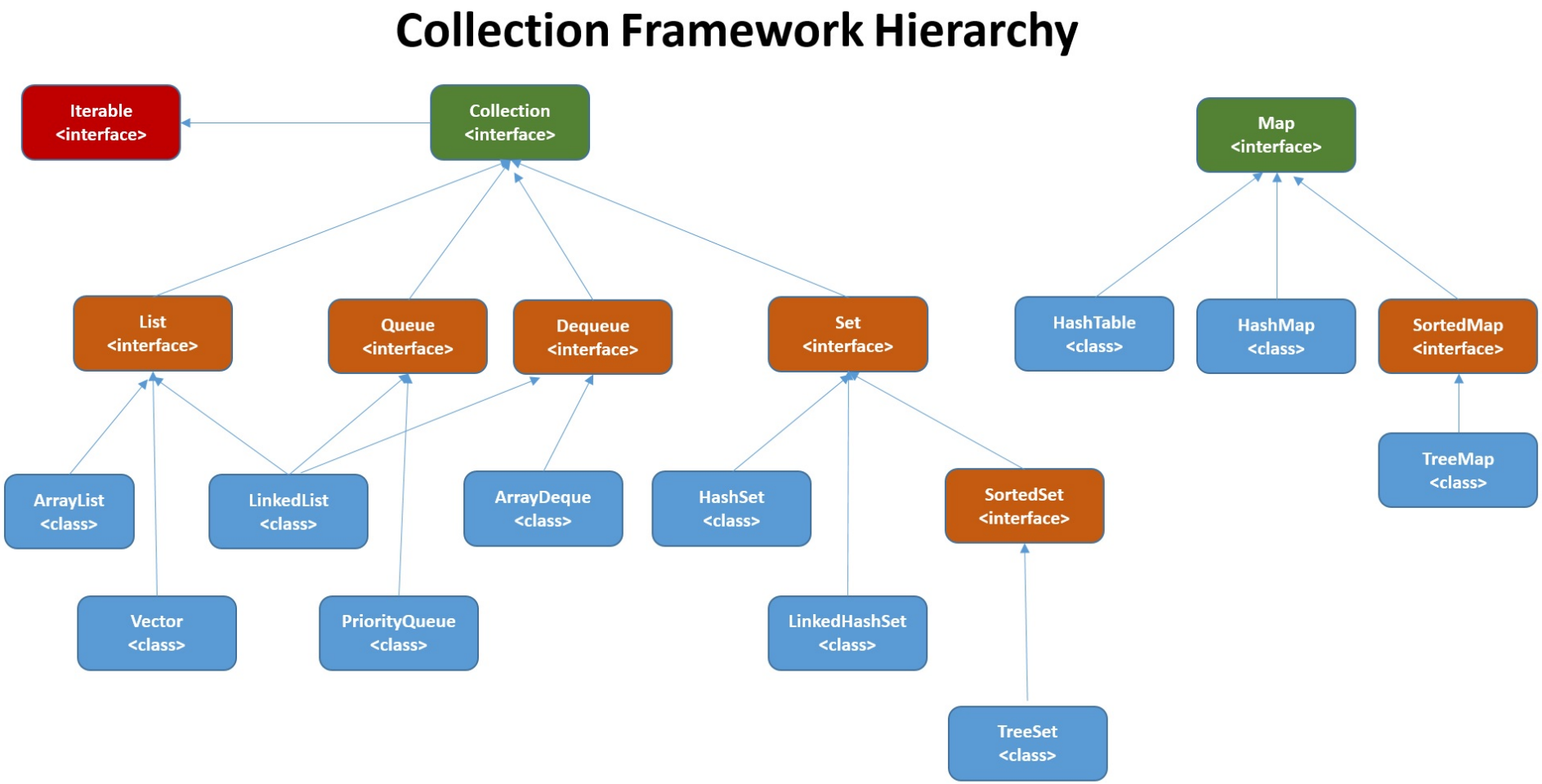
- Putem spune că o clasă furnizează un șablon ce specifică datele și operațiile ce aparțin obiectelor create pe baza șablonului – în documentul de specificații pentru un calculator se menționează că acesta are un monitor și o serie de periferice.
- **Programarea orientată pe obiecte** este o metodă de implementare a programelor în care acestea sunt organizate sub formă de colecții de obiecte care cooperează între ele, fiecare obiect reprezentând instanța unei clase.
- Putem deduce că o clasă descrie un obiect, în general, un nou tip de date. Într-o **clasă** găsim **date** și **metode** ce operează asupra datelor respective.

Colecții în Java

- O **colecție** este un obiect care grupează mai multe elemente într-o singură unitate. Prin intermediul colecțiilor, vom avea acces la diferite tipuri de structuri de date, cum ar fi vectori, liste înlănțuite, stive, mulțimi matematice, tabele de dispersie, etc. Colecțiile sunt folosite atât pentru memorarea și manipularea datelor, cât și pentru transmiterea unor informații de la o metodă la alta.
- Tipul de date al elementelor dintr-o colecție este `Object`, ceea ce înseamnă că mulțimile reprezentate sunt eterogene, putând include obiecte de orice tip.



Colecții în Java



Colecții în Java

- Interfețele reprezintă nucleul mecanismului de lucru cu colecții, scopul lor fiind de a permite utilizarea structurilor de date independent de modul lor de implementare. Toate clasele concrete care au ca super-tip interfața `Collection` implementează conceptul de colecție de obiecte.
- `Collection` modelează o colecție la nivelul cel mai general, descriind un grup de obiecte numite și elementele sale.
- Unele implementări ale acestei interfețe permit existența elementelor duplicate, alte implementări nu.
- Unele au elementele ordonate, altele nu.
- Platforma Java nu oferă o implementare directă a acestei interfețe, ci există doar implementări ale unor sub-interfețe mai concrete, cum ar fi `Set` sau `List`.

Colecții în Java

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    @Override
    boolean equals(Object o);
    @Override
    int hashCode();
}
```

Colecții în Java

Cu ce seamănă această interfață?

Colecții în Java

Cu ce seamănă această interfață?

Cu ceea ce prezentam drept definiție a TAD-ului!

Interfața `Collection` definește o serie de metode pe care o clasă trebuie să le implementeze pentru a fi considerată o colecție în Java. Aceste metode oferă funcționalități de bază pentru gestionarea și manipularea elementelor unei colecții.

Interfața `Collection` extinde interfața `Iterable`, ceea ce înseamnă că o colecție poate fi iterată prin intermediul unui iterator.

Colecții în Java

Class	Map	Set	List	Ordered	Sorted
`HashMap`	x			No	No
`Hashtable`	x			No	No
`TreeMap`	x			Sorted	By *natural order* or custom comparison rules
`LinkedHashMap`	x			By insertion order or last access order	No
`HashSet`		x		No	No
`TreeSet`		x		Sorted	By *natural order* or custom comparison rules
`LinkedHashSet`		x		By insertion order	No
`ArrayList`			x	By index	No
`Vector`			x	By index	No
`LinkedList`			x	By index	No
`PriorityQueue`				Sorted	By to-do order

Colecții în Java – Exemplul I

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {
    public static void main(String[] args) {
        // Crearea unei liste de tip ArrayList
        List<String> lista = new ArrayList<>();
        // Adăugarea elementelor în listă
        lista.add("Ana");
        lista.add("Bogdan");
        lista.add("Cristina");
        // Parcurgerea și afișarea elementelor
        for (String element : lista) {
            System.out.println(element);
        }
    }
}
```

Colecții în Java – Exemplul II

```java
import java.util.HashSet;
import java.util.Set;
public class HashSetExample {
    public static void main(String[] args) {
        // Crearea unui set de tip HashSet
        Set<Integer> set = new HashSet<>();
        // Adăugarea elementelor în set
        set.add(10);
        set.add(30);
        set.add(20);
        set.add(10);
        // Parcurgerea și afișarea elementelor
        for (Integer element : set) {
            System.out.println(element);
        }
    }
}
```

Colecții în Java – Exemplul III

```java
import java.util.HashMap;
import java.util.Map;
public class HashMapExample {
    public static void main(String[] args) {
        // Crearea unui map de tip HashMap
        Map<String, Integer> map = new HashMap<>();
        // Adăugarea perechilor cheie-valoare în map
        map.put("Ana", 25);
        map.put("Bogdan", 30);
        // Parcurgerea și afișarea perechilor cheie-valoare
        for (Map.Entry<String, Integer> entry : map.entrySet
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println(key + " : " + value);
        }
    }
}
```

Colecții în Java – Exemplul IV

```java
import java.util.ArrayList;
import java.util.List;

public class HeterogeneousCollectionExample {
    public static void main(String[] args) {
        // Crearea unei liste de tip ArrayList pentru colecț
        List<Object> colectie = new ArrayList<>();
        // Adăugarea elementelor de diferite tipuri în colec
        colectie.add("Ana");
        colectie.add(25);
        colectie.add(true);
        // Parcurgerea și afișarea elementelor din colecție
        for (Object element : colectie) {
            System.out.println(element);
        }
    }
}
```

Colecții în Java – Exemplul V

```java
// Implementarea unui Comparator personalizat pentru numere întregi în ordine descrescătoare
class CustomComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
    }
}
```

Colecții în Java – Exemplul V

```
1  public class SortedSetExample {
2      public static void main(String[] args) {
3          // Crearea unui TreeSet cu un Comparator
           ↪  personalizat pentru numere întregi
4          TreeSet<Integer> set = new TreeSet<>(new
           ↪  CustomComparator());
5          set.add(5);
6          set.add(2);
7          set.add(8);
8          set.add(5);
9          set.add(4);
10         for (Integer element : set) {
11             System.out.println(element);
12         }
13     }
14 }
```

Implementarea unei liste în Java

```
1  public class LinkedList<T> {
2      private Node<T> head;
3      private int size;
4
5      private static class Node<T> {
6          private T data;
7          private Node<T> next;
8
9          public Node(T data) {
10             this.data = data;
11             this.next = null;
12         }
13     }
14
```

Implementarea unei liste în Java

```
15      public LinkedList() {
16          this.head = null;
17          this.size = 0;
18      }
19
20      public int size() {
21          return size;
22      }
23
24      public boolean isEmpty() {
25          return size == 0;
26      }
27
```

Implementarea unei liste în Java

```
28      public void addFirst(T data) {
29          Node<T> newNode = new Node<>(data);
30          newNode.next = head;
31          head = newNode;
32          size++;
33      }
34      public void addLast(T data) {
35          Node<T> newNode = new Node<>(data);
36          if (isEmpty()) {
37              head = newNode;
38          } else {
39              Node<T> current = head;
40              while (current.next != null)
41                  current = current.next;
42              current.next = newNode;
43          } size++;
44      }
```

Implementarea unei liste în Java

```
45      public void removeFirst() {
46          if (isEmpty())
47              return;
48          head = head.next;
49          size--;
50      }
51      public void display() {
52          Node<T> current = head;
53          while (current != null) {
54              System.out.print(current.data + " ");
55              current = current.next;
56          }
57          System.out.println();
58      }
```

Implementarea unei liste în Java

```
59      public void removeLast() {
60          if (isEmpty())
61              return;
62          if (size == 1) {
63              head = null;
64          } else {
65              Node<T> current = head;
66              Node<T> prev = null;
67              while (current.next != null) {
68                  prev = current;
69                  current = current.next;
70              }
71              prev.next = null;
72          }
73          size--;
74      }
75  }
76
```

Structuri de date în Python

- În Python, există mai multe structuri de date și colecții deja implementate în biblioteca standard, care oferă funcționalități puternice și ușor de utilizat pentru gestionarea și manipularea datelor. Iată câteva dintre cele mai comune structuri de date disponibile în Python:
  1. **Listă** (`list`): Este o structură de date care stochează o colecție ordonată de elemente. Listele pot conține obiecte de tipuri diferite și pot fi modificate (elemente pot fi adăugate, eliminate sau modificate) în mod dinamic.
  2. **Tuplu** (`tuple`): Este o structură de date similară listei, dar este imutabilă, ceea ce înseamnă că nu poate fi modificată după creare. Tuplurile sunt utile când avem date care nu trebuie să fie schimbate, cum ar fi coordonatele unui punct sau valorile constante.
  3. **Dicționar** (`dict`): Este o structură de date care stochează perechi cheie-valoare. Elementele dintr-un dicționar sunt neordonate și accesate prin intermediul cheilor unice. Dicționarele sunt utile atunci când avem nevoie să căutăm sau să actualizăm rapid valori bazate pe o cheie dată.
  4. **Mulțime** (`set`): Este o structură de date care stochează un set neordonat de elemente unice.

Liste în Python

Lista (`list`) este o structură de date flexibilă și modificabilă care poate conține o colecție ordonată de elemente.

1. Crearea unei liste

Listele pot fi create prin includerea elementelor între paranteze pătrate ([]) separate prin virgulă. Elementele dintr-o listă pot fi de orice tip, inclusiv altele liste.

```
lista = [1, 2, 3, 4, 5]
```

2. Accesarea elementelor din listă

Elementele dintr-o listă pot fi accesate utilizând indexul lor. Indexul începe de la 0 pentru primul element.

```
lista = [1, 2, 3, 4, 5]
print(lista[0])    # Output: 1
print(lista[-1])   # Output: 5 (sfârșitul listei)
```

Liste în Python

1. Modificarea elementelor din listă

Listele din Python sunt modificabile, ceea ce înseamnă că putem actualiza valorile elementelor existente utilizând indexul lor.

```
lista = [1, 2, 3, 4, 5]
lista[2] = 10
print(lista)  # Output: [1, 2, 10, 4, 5]
```

2. Adăugarea și eliminarea elementelor

Listele din Python oferă metode și operații pentru adăugarea și eliminarea elementelor.

```
lista = [1, 2, 3]
lista.append(4) # Adăugarea unui element la final
print(lista) # Output: [1, 2, 3, 4]
lista.insert(1, 5) # Adăugarea unui element la un index
print(lista) # Output: [1, 5, 2, 3, 4]
lista.remove(2) # Eliminarea unui element după valoare
print(lista) # Output: [1, 5, 3, 4]
```

Implementarea unei liste în Python

În Python există conceptul de **Clasă**!

Am putea implementa ceva similar ca în Java?

Implementarea unei liste în Python

În Python există conceptul de **Clasă**!

Am putea implementa ceva similar ca în Java?

Da, putem. Ba chiar este mai simplu pentru că nu mai avem tipuri specifice!

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
```

Implementarea unei liste în Python

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
```

`self` în Python este echivalentul lui `this` din Java.

Implementarea unei liste în Python

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def is_empty(self):
        return self.head is None

    def add_front(self, data):
        new_node = Node(data)
        if self.is_empty():
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
```

Implementarea unei liste în Python

```
def add_rear(self, data):
    new_node = Node(data)
    if self.is_empty():
        self.head = new_node
    else:
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
        new_node.prev = current
```

Implementarea unei liste în Python

```
def remove_front(self):
    if self.is_empty():
        return None
    else:
        data = self.head.data
        self.head = self.head.next
        if self.head:
            self.head.prev = None
        return data
```

Implementarea unei liste în Python

```
def remove_rear(self):
    if self.is_empty():
        return None
    else:
        current = self.head
        while current.next:
            current = current.next
        data = current.data
        if current.prev:
            current.prev.next = None
        else:
            self.head = None
        return data
```

Implementarea unei liste în Python

```
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print()

# Exemplu de utilizare:
dll = DoublyLinkedList()
dll.add_front(3)
dll.add_front(2)
dll.add_front(1)
dll.add_rear(4)
dll.add_rear(5)
dll.display()  # Output: 1 2 3 4 5
dll.remove_front()
dll.remove_rear()
dll.display()  # Output: 2 3 4
```

Dicționare în Python

Dicționarul (`dict`) este o structură de date flexibilă și modificabilă care stochează perechi cheie-valoare. Dicționarele permit accesul rapid la valori utilizând cheile asociate și sunt utile atunci când trebuie să căutăm sau să actualizăm valori bazate pe o cheie specifică.

Cu ce structură de date credeți că este echivalent dicționarul?

Dicționare în Python

Dicționarul (`dict`) este o structură de date flexibilă și modificabilă care stochează perechi cheie-valoare. Dicționarele permit accesul rapid la valori utilizând cheile asociate și sunt utile atunci când trebuie să căutăm sau să actualizăm valori bazate pe o cheie specifică.

Cu ce structură de date credeți că este echivalent dicționarul?

Este echivalent cu o tabelă de dispersie!

```
>>> hash("abcd")
2443065157591745621
>>> hash((2,2))
1901736143494378007
```

Dicționare în Python

1. Crearea unui dicționar

Dicționarele pot fi create utilizând parantezele grafe () și specificând perechile cheie-valoare separate prin virgulă.

```
dictionar = {"cheie1": 10, "cheie2": 20, "cheie3": 30}
```

2. Accesarea valorilor din dicționar

Valorile dintr-un dicționar pot fi accesate utilizând cheia asociată.

```
dictionar = {"cheie1": 10, "cheie2": 20, "cheie3": 30}
print(dictionar["cheie1"])  # Output: 10
print(dictionar["cheie3"])  # Output: 30
```

Dacă cheia specificată nu există în dicționar, se va genera o excepție `KeyError`. Pentru a evita această situație, putem utiliza metoda `get()` care returnează valoarea asociată unei chei și returnează un anumit rezultat (de exemplu, None sau o valoare prestabilită) în cazul în care cheia nu există.

Dicționare în Python

1. Modificarea valorilor în dicționar

Valorile dintr-un dicționar pot fi actualizate utilizând cheia asociată.

```
dictionar = {"cheie1": 10, "cheie2": 20}
dictionar["cheie1"] = 30
print(dictionar)  # Output: {"cheie1": 30, "cheie2": 20}
```

Dacă cheia specificată nu există în dicționar, o nouă pereche cheie-valoare va fi adăugată.

2. Adăugarea și eliminarea perechilor cheie-valoare

Putem adăuga perechi cheie-valoare noi într-un dicționar sau putem elimina perechi existente utilizând diverse metode.

```
dictionar = {"cheie1": 10}
dictionar["cheie2"] = 20 # Adăugă o pereche cheie-valoare
print(dictionar) # Output: {"cheie1": 10, "cheie2": 20}
del dictionar["cheie1"]
```

Implementarea unui arbore în Python

```
def creare_nod(eticheta, copii=None):
    if copii is None:
        copii = {}
    return {"eticheta": eticheta, "copii": copii}

def adaugare_nod(parinte, eticheta, copii=None):
    nod = creare_nod(eticheta, copii)
    parinte["copii"][eticheta] = nod

def parcurgere_arbore(nod, nivel=0):
    print("\t" * nivel, nod["eticheta"])
    for eticheta, copil in nod["copii"].items():
        parcurgere_arbore(copil, nivel+1)
```

Implementarea unui arbore în Python

```
# Exemplu de creare a unui arbore:
arbore = creare_nod("A")
adaugare_nod(arbore, "B")
adaugare_nod(arbore, "C")
adaugare_nod(arbore["copii"]["B"], "D")
adaugare_nod(arbore["copii"]["B"], "E")
adaugare_nod(arbore["copii"]["C"], "F")

# Exemplu de parcurgere a arborelui:
parcurgere_arbore(arbore)
```

Implementarea unei stive în Python

```
def create_stack():
    return {"stiva": [], "varf": 0}

def is_empty(stack):
    return stack["varf"] == 0

def push(stack, element):
    stack["varf"] += 1
    stack["stiva"].append(element)

def pop(stack):
    if is_empty(stack):
        raise Exception("Stiva este goală!")
    element = stack["stiva"].pop()
    stack["varf"] -= 1
    return element
```

Implementarea unei stive în Python

```
def peek(stack):
    if is_empty(stack):
        raise Exception("Stiva este goală!")
    return stack["stiva"][-1]

def size(stack):
    return stack["varf"]

stiva = create_stack()
push(stiva, 10)
push(stiva, 20)
push(stiva, 30)
```

Implementarea unei liste în Python

```
def create_node(data):
    return {"data": data, "next": None, "prev": None}

def add_node(head, data):
    new_node = create_node(data)
    if head is None:
        return new_node
    curr_node = head
    while curr_node["next"] is not None:
        curr_node = curr_node["next"]
    curr_node["next"] = new_node
    new_node["prev"] = curr_node
    return head
```

Implementarea unei liste în Python

```
def print_list(head):
    curr_node = head
    while curr_node is not None:
        print(curr_node["data"], end=" ")
        curr_node = curr_node["next"]
    print()

# Exemplu de utilizare
head = create_node(1)
head = add_node(head, 2)
head = add_node(head, 3)
head = add_node(head, 4)

print_list(head)                    # Output: 1 2 3 4
```

Paradigma logică

- **Program** = colecție de reguli logice și fapte care reprezintă **baza de cunoștințe** a programatorului.
- **Rularea unui program** = găsirea de răspunsuri la întrebări puse de programator prin **raționament logic** bazat pe cunoătințele din program.
- **Programarea logică** este un stil de **programare declarativă**.
- **Faptele** sunt afirmații despre proprietățile obiectelor și relațiile existente între ele.
- O **regula**, în limbaj natural, este o frază de forma:
  - **Dacă** *ipoteza*$_1$, ... și *ipoteza*$_n$, atunci concluzie.

Limbajul Prolog

- Un program Prolog este o colecție de definiții ce descriu relații sau funcții de calculat - reprezentări simbolice de obiecte și relații între obiecte. Soluția problemelor nu se mai vede ca o execuție pas cu pas a unei secvențe de instrucțiuni.
- **Fapt** = ceea ce se cunoaște a fi adevărat.
- **Regulă** = ce se poate deduce din fapte date (indică o concluzie care se știe că e adevărată atunci când alte concluzii sau fapte sunt adevărate).
- **Concluzie** - Prolog folosește **rezoluția** pentru a demonstra dacă este adevărată sau nu, pornind de la ipoteza stabilită de faptele și regulile definite.

Sintaxă

Fapte și reguli

1. **Faptele** stabilesc relații între obiectele universului problemei.
   Exemple:

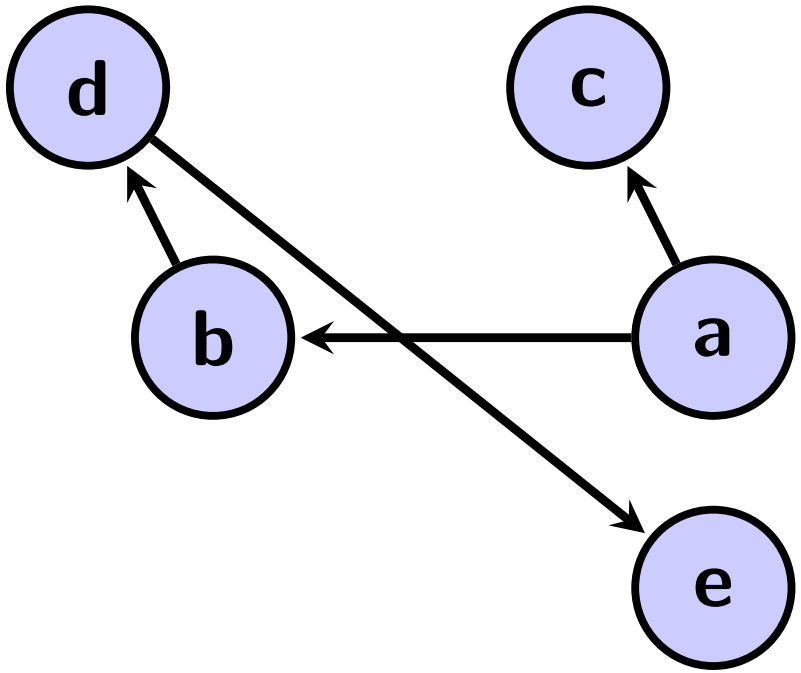
```
frumoasa(ana).   % Ana este frumoasa.
bun(vlad). % Vlad este bun.
cunoaste(vlad, maria). % Vlad o cunoaste pe Maria
cunoaste(vlad, ana). % Vlad o cunoaste pe Ana.
iubeste(mihai, maria). % Mihai o iubeste pe Maria.
```

2. **Regulile**

```
iubeste(X, Y):- bun(X), cunoaste(X, Y), frumoasa(Y).
```

Exemplu de graf în Prolog

```
edge(a,b).
edge(a,c).
edge(b,d).
edge(d,e).
```




```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Exemplu de graf în Prolog

```
% Fapte care descriu relațiile în graf
edge(a, b).   edge(b, c).
edge(b, d).   edge(c, d).
edge(d, e).   edge(e, a).

% Predicat pentru verificarea existenței unui drum între
 ↪  două noduri în graf
path(X, Y, _) :- edge(X, Y).  % Există o muchie directă
 ↪  între X și Y
path(X, Y, Visited) :- edge(X, Z), \+ member(Z, Visited),
 ↪  path(Z, Y, [X | Visited]).

?- path(a, e, []).
true.
?- path(c, d, []).
true.
?- path(a, c, []).
```

Vă mulțumesc pentru atenție!