6. Abstraktní datové typy

Abstraktní datové typy a jejich rozdíl od strukturovaných datových typů (obecně)

- ADT jsou koncepty používané v programování, které umožňují abstraktní reprezentaci dat a operací s nimi. Tyto typy jsou definovány pomocí rozhraní a poskytují programátorovi určitou úroveň abstrakce, což umožňuje snazší práci s daty
- Na rozdíl od strukturovaných datových typů, jako jsou pole nebo záznamy, které mají pevně definovanou strukturu a reprezentaci dat, ADT umožňují programátorovi abstrahovat se od konkrétní implementace a zaměřit se na funkcionalitu a chování. To umožňuje snadnější úpravu a rozšíření kódu bez nutnosti měnit implementaci datové struktury

Souvislost s rozhraním

- ADT jsou úzce spojeny s rozhraním, protože rozhraní definují metody a operace, které jsou součástí ADT. Programátor může vytvořit konkrétní implementaci ADT pomocí těchto metod a operací, což poskytuje konzistentní rozhraní pro práci s daty

Souvislost s generickým programováním

- Generické programování umožňuje programátorovi psát kód, který pracuje s různými typy dat, aniž by byl specificky navržen pro konkrétní typ. To je užitečné v případě ADT, protože může být vytvořena obecná implementace, která pracuje s různými typy dat, a tato implementace může být použita pro různé druhy dat

Princip LIFO a FIFO

- Princip LIFO (Last In First Out) a FIFO (First In First Out) jsou typy front, které se používají v rámci některých ADT
- LIFO znamená, že poslední prvek, který byl vložen do fronty, je první, který je odebrán
    - To se používá například v zásobníku (stack)
- FIFO znamená, že první prvek, který byl vložen do fronty, je také první, který je odebrán. To se používá například ve frontě (queue)
    - Tyto principy jsou důležité pro porozumění fungování některých ADT a jsou často používány v algoritmech a datových strukturách

Prakticky:

- Vyjmenování ADT, poznání konkrétního dle kódu
    - Zásobník (Stack)
    - Fronta (Queue)
    - Strom (Tree)
    - HashTable (Hashovací tabulka)
    - LinkedList

```java
// Pushing element on the top of the stack
static void stack_push(Stack<Integer> stack)
{
    for(int i = 0; i < 5; i++)
    {
        stack.push(i);
    }
}

// Popping element from the top of the stack
static void stack_pop(Stack<Integer> stack)
{
    System.out.println("Pop Operation:");

    for(int i = 0; i < 5; i++)
    {
        Integer y = (Integer) stack.pop();
        System.out.println(y);
    }
}

// Displaying element on the top of the stack
static void stack_peek(Stack<Integer> stack)
{
    Integer element = (Integer) stack.peek();
    System.out.println("Element on stack top: " + element);
}
```

- Zásobník

```java
public class Queue<T> {
    11 usages
    private T[] data;
    3 usages
    private int dataTop;

    1 usage
    public Queue(int size){
        this.data = (T[]) new Object[size];
        dataTop = 0;
    }

    3 usages
    public void enqueue(T item){
        T[] copy = (T[]) new Object[data.length];
        for (int i = 0; i < data.length; i++) {
            copy[i] = data[i];
            if (data[i] == null){
                copy[i] = item;
                break;
            }
        }
        data = copy;
        dataTop++;
    }

    2 usages
    public T dequeue(){
        if (!isEmpty()){
            T temp = data[0];
            for (int i = 0; i < data.length-1; i++) {
                data[i] = data[i + 1];
            }
            return temp;
        }
        return null;
    }
```

- Fronta

```java
public class NodeController {
    1 usage
    public void build(){
        Node RootNode = new Node( number: 25);
        System.out.println("Working...\n==========");


        insert(RootNode,  value: 70);
        insert(RootNode,  value: 5);
    }

    4 usages
    protected void insert(Node node, int value){
        if (value < node.value)
        {
            if (node.left != null)
            {
                insert(node.left, value);
            } else
            {
                System.out.println("  Inserted " + value + " to left of Node " + node.value);
                node.left = new Node(value);
            }
        }
        else if (value > node.value)
        {
            if (node.right != null)
            {
                insert(node.right, value);
            } else
            {
                System.out.println("  Inserted " + value + " to right of Node " + node.value);
                node.right = new Node(value);
            }
        }
    }
```

- Strom

```java
public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
    System.out.println(capitalCities);
  }
}
```

- Hashtable

```java
public class Main {
  public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

- LinkedList