Ime, priimek / Name : Urban Potocnik

LV 01: Podatkovne strukture in algoritmi / Data structures and algorithms

1 Uvod / Introduction

Pri implementaciji različnih algoritmov in postopov je pogosto ključnega pomena tudi, kako se bodo podatki in rezultati shranjevali ter prikazovali. Medtem ko pri osnovnem programiranju pogosto srečamo primitivne ter kompozitne tipe podatkovnih struktur, kot so char, double, integer ali array, se pri naprednejših pristopih uporabljajo abstraktne podatkovne strukture, kot so drevesa, vrste in grafi. Za te strukture je značilno, da imajo definirana pravila in osnovne funkcije, s pomočjo katerih jih uporabljamo, dokončna implementacija strukture pa je odvisna od sistema / programerja / opreme, ki nam je na voljo.

V okviru te laboratorijske vaje se boste tako spoznali z vrsto (queue) in skladom (stack). Na ti strukturi se bomo osredotočili, ker sta pogosto prisotni v simulacijah telekomunikacijskega prometa (omrežni predpomnilnik) oziroma programskih elementov (medpomnilnik).

> *When designing algorithms, the selection of proper data structures is of crucial importance. Basic programming tasks often use primitive and composite data types such as char, double, integer or array. Advanced tasks, however, often rely on more complex (abstract) data structures such as trees, queues, stackss and graphs. The difference from primitive data types is that most of these data structures are simply a collection of rules and functions that can be implemented in many ways - mostly depending on the environment / programmer / equipment used in the project.*
>
> *The scope of this exercise covers the stack and the queue and examples of their use.*

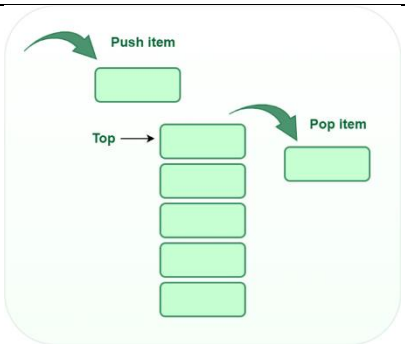
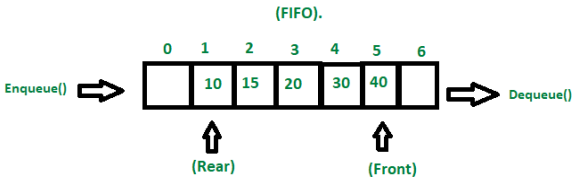
1.1 Sklad in vrsta / Stack and queue data structures

Sklad in vrsta sta linearni podatkovni strukturi, ki omogočata shranjevanje (vnašanje) poljubnih podatkov (objektov, item), ter njihovo branje (in odstranjevanje) v predpisanem vrstnem redu.

- **Sklad**: dodajanje in odstranjevanje poteka na istem koncu (na vrhu sklada), vrstni red je **LIFO : Last In First Out**
- **Vrsta**: dodajanje poteka na enem koncu (end) in odstranjevanje na drugem (front), vrstni red odstranjevanja je **FIFO: First In First Out.**

Osnovne metode:

- **push(item) / enqueue(item):** doda (shrani) nov objekt v podatkovno strukturo. V primeru vrste se ta metoda imenuje tudi enqueue(item).
- **pop() / dequeue(): iz** podatkovne strukture zbriše podatek, ter ga vrne kot rezultat. V primeru vrste je enaka metoda dequeue(). Če je struktura prazna, vrne None.

A stack *is a linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.* *LIFO (Last In First Out): This strategy states that the element that is inserted last will come out first.* *More info*	
A queue *is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.* *We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end.* *More info*	

1.2 Sklad / Stack test

DataStructures.py implements Stack and Queue classes.

Import Stack and test all provided methods. Example:

```
from DataStructures import Stack, Queue

sklad = Stack('TestStack')
sklad.push('A')
sklad.push('B')
sklad.push('C')
sklad.print()
print("Pop() returned: ", sklad.pop() )
sklad.print()
```

Try other methods, and insert resulting printout:

TestStack size: 3 , data: ['A', 'B', 'C']

TestStack size: 2 , data: ['A', 'B']

Pop() returned: B

TestStack size: 1 , data: ['A']

1.3 Vrsta / Queue test

DataStructures.py implements Stack and Queue classes.

Import Queue and test all provided methods.

Results:

Queue se uvozi v datoteko.

1.4 Izpiši tekst nazaj / Print text in reverse order

Task: create algorithm to print given text in reverse order. Select either stack or queue.

```
text = 'Danes je lep sončen dan'
output = ''
s = Stack()
q = Queue()

print(output)
```

Results (example):

nad nečnos pel ej senaD

1.5 Izpiši soglasnike / Remove vowels and print consonants

Task: create algorithm to remove vowels (a,e,i,o,u) from text and print resulting consonants. Use either stack or queue.

```
text = 'Danes je lep sončen dan in vidim mavrico na nebu.'
output = ''
s = Stack()
q = Queue()

print(output)
```

Results:

.bn crvm dvn nčns pl j snD

2 Algoritmi / Algorithms

Algoritmi so po definiciji navodilo, s katerim rešujemo problem, podano kot seznam korakov, ki nas pripeljejo do rešitve problema. V praksi to hkrati pomeni, da lahko za vsak algoritem obstaja poljubno mnogo rešitev (programov), ki ta algoritem izvajajo. Kompleksnejši algoritmi tako ponavadi tudi zahtevajo kompleksnejše tipe podatkov oziroma podatkovnih struktur, s pomočjo katerih lahko sledimo (vmesnim) rezultatom ter prikazujemo napredek postopka. Algoritme pogosto prikažemo tudi z grafom poteka korakov.

Pravilna izbira podatkovne strukture ima tako velik vpliv na pravilno interpretacijo rezultatov. V sklopu laboratorijske vaje si bomo tako pogledali dva algoritma.

> *By definition algorithms are a set of rules / instructions used to solve a set problem. These instructions are a series of steps which lead us to the solution of the problem. This means that we can implement each algorithm in many different ways, depending on the environment constraints (RAM, CPU speed etc.). Complex algorithms usually require complex data structures which enable us to monitor and report the progress. Algorithms can also be presented graphically. The choice of data structures plays an important part in the interpretation of results.*

2.1 Pretvorba števil / Number base conversion

We will use Stack to store binary representation of the decimal number (bits). The binary conversion algorithm will create and return stack data structure.

```
from DataStructures import Stack
# Conversion to binary
def decToBinary( x ):
    out = Stack('Bits')

    return out
```

Algorithm: while input decimal number (x) is larger than 0, calculate bit as x modulo 2, and push to stack. Calculate new x by integer division by 2.

Results (example):

Bits size: 4 , data: [0, 1, 0, 1]

```
# binary to decimal
def binaryToDecimal( st ):
    out = 0
    potenca = 2**(st.size()-1)
    while not st.empty():

    return out
```

Algorithm: while stack is not empty, multiply bit (from stack) with corresponding number 2 ^ exponent.

Test conversion methods with different numbers, and print binary representation (bits) in correct order (MSB first).

Results:

Decimal value: 10

2.2 (Dodatno) Pretvorba v poljubno bazo / Optional task: convert to any base

Modify algorithms of number conversion by adding additional parameter base, so that they can convert decimal number to any base (such as 8, 16 etc.).

def decToBase(x, base): out = Stack('Bits') while x > 0: bit = x % base out.push(bit) x = x // base return out

def baseToDecimal(st, base): out = 0 potenca = base ** (st.size() - 1) while not st.empty(): bit = st.pop() out += bit * potenca potenca = potenca // base return out

base_stack = decToBase(255, 16) base_stack.print() print("Decimal value:", baseToDecimal(base_stack, 16))

(tukaj sem samo nalepi kodo, lepsa identirana koda je v mojem notebooku)

2.3 Sprehod po drevesu v globino / Graph (tree) traversal: Depth First Search

Vhodni podatek je drevo, vozlišča (node, vertice) lahko predstavljajo poljubni podatek, v našem primeru je to mesto. Povezave (edges) predstavljajo ceste in razdalje med kraji.

> *Input data: graph (tree) with nodes (vertices) representing cities of Slovenia, and edges representing roads between cities.*
>
> *Load and plot input graph:*

```
import matplotlib.pyplot as plt
import networkx as nx

# Load graph
G = nx.read_graphml('Slo_cities_tree.graphml')
print(G.nodes)
print(G.edges)

# prepare an empty figure
plt.figure()

#Plot the graph
nx.draw_networkx(G, with_labels=True, font_size = 8)
```

Algoritem Sprehod v globino / Depth first search Algorithm:

Description: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/?ref=lbp

Code:

```
from DataStructures import Stack

def visit_node(node, visited, visited_order, stack, graph, counter_list):
    visited.add(node)
    visited_order.append((counter_list[0], node))
    print("Visit: ", counter_list[0], node)
    counter_list[0]+=1
    for sosed in graph.adj[node]:
        if sosed not in visited:
            stack.push(sosed)
    # preglej kaj je na skladu
    stack.print()


# Create a set to store visited vertices
visited = set()
# Create stack to store nodes to visit
stack = Stack()
# iterator over visits
counter_list = [0]
# Resulting sequence
node_order = []

# Select your input graph (networkx Graph):
inputGr = G
# Select your starting point:
start_node = 'Ljubljana'

print("Graph nodes:", inputGr.nodes)

# ALGORITHM DFS
node = start_node
visit_node(node, visited, node_order, stack, inputGr, counter_list)
while stack.size() > 0:
    node = stack.pop()
    if node not in visited:
        visit_node(node, visited, node_order, stack, inputGr, counter_list)

print("Visited nodes: ", node_order)
```

Test algorithm by selecting different starting points (cities) and check order of visited cities. Example result:

Graph nodes: ['Ljubljana', 'Maribor', 'Celje', 'Koper', 'Kranj', 'Novo mesto']

Visit: 0 Ljubljana

Stack 'Stack': ['Maribor', 'Celje', 'Kranj']

Visit: 1 Maribor

Stack 'Stack': ['Celje', 'Kranj', 'Koper']

Visit: 2 Celje

Stack 'Stack': ['Kranj', 'Koper']

Visit: 3 Kranj

Stack 'Stack': ['Koper', 'Novo mesto']

Visit: 4 Koper

Stack 'Stack': ['Novo mesto']

Visit: 5 Novo mesto

Stack 'Stack': []

Visited nodes: [(0, 'Ljubljana'), (1, 'Maribor'), (2, 'Celje'), (3, 'Kranj'), (4, 'Koper'), (5, 'Novo mesto')]

Q: Opiši na kratko delovanje algoritma, in kako uporablja sklad. ***Q: Describe shortly how algorithm works with stack.***

Algoritem DFS (Depth First Search) uporablja sklad za shranjevanje vozlišč, ki jih še ni obiskal. Začne v izbranem začetnem vozlišču, ga obišče in vse njegove neobiskane sosede doda na sklad. Nato vzame naslednje vozlišče s sklada in postopek ponavlja, dokler sklad ni prazen. Sklad omogoča, da se najprej obiščejo vozlišča, ki so bila dodana zadnja (LIFO).

Q: Ugotovite, od česa je odvisen vrstni red obiskov sosednjih mest ? ***Q: Order of traversal of adjacent nodes (from the starting node) depends on what ?***

Vrstni red obiskov sosednjih mest je odvisen od vrstnega reda, v katerem so sosedi dodani na sklad. V našem primeru je to odvisno od vrstnega reda, v katerem so sosedi definirani v grafu.

2.4 Sprehod po drevesu v širino / Graph (tree) traversal: Breadth First Search

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/?ref=lbp

> *Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited. To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.*

Kopiraj kodo DFS algoritma in jo prilagodi z uporabo vrste, da bo sprehod po drevesu šel v širino.

> *Task: Modify DFS code by replacing Stack with Queue to perform BFS search.*

Results (example):

Graph nodes: ['Ljubljana', 'Maribor', 'Celje', 'Koper', 'Kranj', 'Novo mesto']

Visit: 0 Ljubljana

Queue 'Queue': ['Maribor', 'Celje', 'Kranj']

Visit: 1 Maribor

Queue 'Queue': ['Celje', 'Kranj', 'Koper']

Visit: 2 Celje

Queue 'Queue': ['Kranj', 'Koper']

Visit: 3 Kranj

Queue 'Queue': ['Koper', 'Novo mesto']

Visit: 4 Koper

Queue 'Queue': ['Novo mesto']

Visit: 5 Novo mesto

Queue 'Queue': []

Visited nodes: [(0, 'Ljubljana'), (1, 'Maribor'), (2, 'Celje'), (3, 'Kranj'), (4, 'Koper'), (5, 'Novo mesto')]

Q: Kakšna je razlika v zaporedju obiskovanja vozlišč? What is the difference between BFS and DFS algorithm?

Razlika med BFS (Breadth First Search) in DFS (Depth First Search) je v tem, da BFS obiskuje vozlišča po nivojih (najprej vsa vozlišča na enem nivoju, nato na naslednjem), medtem ko DFS gre čim globlje v drevo, preden se vrne nazaj. BFS uporablja vrsto (FIFO), medtem ko DFS uporablja sklad (LIFO).

3 Simulacija mrežnega medpomnilnika z vrsto / Network buffer simulation using queue

Na področju telekomunikacij se prej ali slej srečamo z omrežji, prometom in povezavami med posameznimi točkami omrežja. Medtem ko se bomo z geometrijo omrežja ukvarjali v kasnejših laboratorijskih vajah, bomo v sklopu te laboratorijske vaje izdelali simulator mrežnega medpomnilnika.

> *One of the most common uses of a queue is shown directly in the data structures name - queuing data packages in ICT (or any other data-collecting field). A queue can be used as a buffer structure that stores incoming data packages and forwards them to the server when it can process them.*
>
> *We will cover network connections and graphs in our future exercises. In this case we will only simulate a single traffic source and destination with a buffer between them.*

```
import numpy as np

# SIMULATION PARAMETERS
input_data_size = 1000
add_probability = 0.5
remove_probability = 0.45

time = 0
input_number = 1

# Outputs
buff_size = []
time_list = []
out_list = []

buffer = Queue('Buffer')

while not (input_number > input_data_size and buffer.empty()):

    time_list.append(time)
    time += 1

    # add input data to queue
    if np.absolute(np.random.random()) < add_probability and input_number <= input_data_size :
        buffer.enqueue(input_number)
        input_number += 1

    # remove data
    if np.absolute(np.random.random()) < remove_probability:
        out = buffer.dequeue()
        if out != None:
            out_list.append(out)

    buff_size.append(buffer.size())

print('Total time: ', time)
print(buff_size)
print(out_list)

plt.figure()
plt.plot(buff_size)
plt.show()
```

Spreminjaj parametre simulacije in ugotovi, kaj pomenijo. V tabeli predstavi nekaj kombinacij parametrov, ter komentiraj, kako vplivajo na zasedenost medpomnilnika ter čas prenosa podatkov.

Task: run simulation and change simulation parameters. Describe the meaning of three parameters, and how their values affect the buffer occupancy (size), and the total time to transfer all input data. You can include examples of graphs, or present results in a table.

Total time: 2000
[0, 1, 2, 1, 2, 3, 2, 3, 4, 3, ...]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...]

add_probability	remove_probability	input_data_size	Zasedenost medpomnilnika	Čas prenosa podatkov
0.5	0.45	1000	Nihanje med 0 in 5	2000
0.6	0.5	1000	Nihanje med 0 in 10	1500
0.4	0.4	1000	Nihanje med 0 in 2	2500