PROJEKT F 28. 11.2018

TERMIN ROZLICZENIA 10.12.2018

ALGORYTMIKA:
ABSTRAKCYJNE STRUKTURY DANYCH: STOS, KOLEJKA, LISTA

PYTHON:
PROGRAMOWANIE ZORIENTOWANE OBIEKTOWO, KLASY

https://www.python-course.eu/python3_object_oriented_programming.php

0_class

http://interactivepython.org/runestone/static/pythonds/index.html

https://www.cs.auckland.ac.nz/compsci105s1c/lectures/

1_class

Programowanie zorientowane obiektowo w Pythonie

Klasa *Robot* o dwóch metodach: *__init__* oraz *say_hi*

- Nazwa klasy : rozpoczęcie definicji klasy : slowo class, tworzy przestrzeń nazw

```
class Robot:

    def __init__(self, name=None):
        """ konstruktor obiektu klasy """
        self.name = name

    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")
```

Zmienna *self* odnosi się do samego obiektu

2_class

3_class

```
class Robot:

    def __init__(self,
                 name=None,
                 build_year=None):
        self.name = name
        self.build_year = build_year

    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")

    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_build_year(self, by):
        self.build_year = by

    def get_build_year(self):
        return self.build_year
```

Metody getter

Metody specjalne:

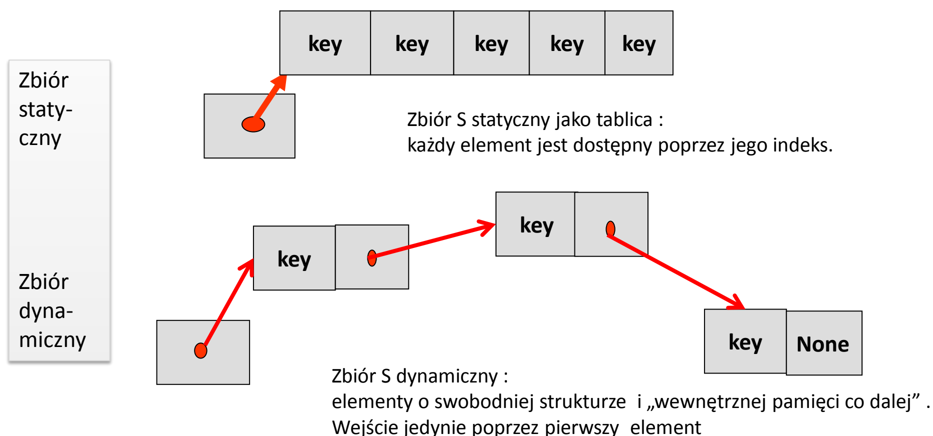
__init__ : konstrukcja obiektu klasy
__str__ : oficjalna reprezentacja tekstowa obiektu
__repr__ : reprezentacja robocza obiektu

http://users.uj.edu.pl/~ufkapano/algorytmy/lekcja06/intro.html

Cechy programowania zorientowanego obiektowo:
- Programy są zbudowane z (1) definicji obiektów i (2) definicji funkcji. Ponadto większość obliczeń wyrażona jest w postaci operacji na obiektach.
- Każda definicja obiektu odpowiada obiektowi lub koncepcji w świecie rzeczywistym. Funkcje działające na obiektach odpowiadają sposobowi działania obiektów w świecie rzeczywistym

Koncepcje związane z programowaniem zorientowanym obiektowo:
- *dziedziczenie* - w Pythonie oparte na wyszukiwaniu atrybutów.
- *polimorfizm* - znaczenie metody (operacji) uzależnione jest od typu (klasy) obiektu, na którym się tą operację wykonuje.
- *hermetyzacja* (enkapsulacja) - ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Można modyfikować implementację interfejsu bez wpływania na użytkowników tego obiektu.
- *kompozycja* - osadzanie innych obiektów w obiekcie pojemnika.
- *wzorce projektowe* (ang. design patterns) - często wykorzystywane struktury programowania zorientowanego obiektowo.

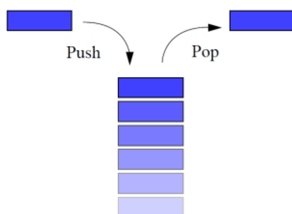
Abstrakcyjne typy danych: zbiór statyczny a dynamiczny?



Dynamiczny zbiór danych to zestaw danych i operacji umożliwiających sprawne modyfikacje zawartości zbioru: dokładania i kasowania elementu, wydajne przeglądanie zawartości : wyszukiwania elementu

Abstrakcyjne typy danych: co to jest i po co to jest?

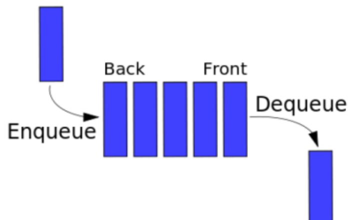
Stos: uporządkowana struktura danych, w których dane są dokładane i pobierane z wierzchołka stosu.



Realizuje bufor LIFO: Last In, First Out

Operacje realizowane w tej strukturze:
- dokładanie elementu: push(item)
- zdejmowanie elementu: pop()
- dostęp do wierzchołka: peek()
- rozmiar : size()
- test czy pusty: isEmpty()

Kolejka: uporządkowana struktura danych, w których nowe dane są dokładane na końcu, natomiast z początku pobiera się dane do dalszego przetwarzania.

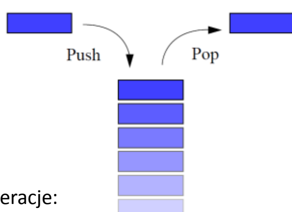


Realizuje bufor FIFO: First In, First Out

Operacje realizowane w tej strukturze:
- dokładanie elementu: enqueue(item)
- usuwanie elementu: dequeue()
- rozmiar : size()
- test czy pusty: isEmpty()

Stos_koleka_na liscie

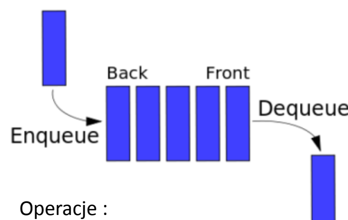
Abstrakcyjne typy danych: stos, kolejka i Python



Operacje:
push(item), pop() , peek(), size(), isEmpty()
realizowane na Pythonowych listach.

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

Nasz_stos



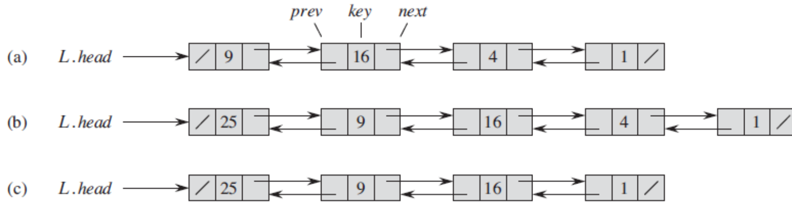
Operacje :
enqueue(item), dequeue(), size(), isEmpty()
realizowane na Pythonowych listach

Nasza_kolejka

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

Abstrakcyjne typy danych: lista

Lista:
Dynamiczny zbiór danych uporządkowanych liniowo (czyli jak w tablicy), przy czym porządek jest określony poprzez wskaźniki.



LIST-SEARCH(L, k)
1 $x = L.head$
2 **while** $x \neq$ NIL and $x.key \neq k$
3 $\quad x = x.next$
4 **return** x

LIST-INSERT(L, x)
1 $x.next = L.head$
2 **if** $L.head \neq$ NIL
3 $\quad L.head.prev = x$
4 $L.head = x$
5 $x.prev =$ NIL

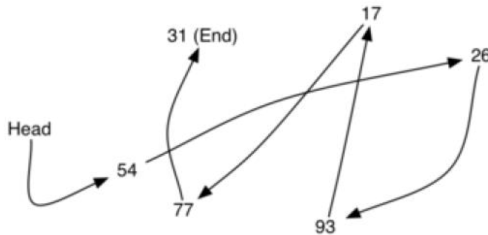
LIST-DELETE(L, x)
1 **if** $x.prev \neq$ NIL
2 $\quad x.prev.next = x.next$
3 **else** $L.head = x.next$
4 **if** $x.next \neq$ NIL
5 $\quad x.next.prev = x.prev$

Ale też isEmpty() i size(),

Pytanie: jak się ma lista typ zmiennej w Pythonie a wyżej opisana lista?

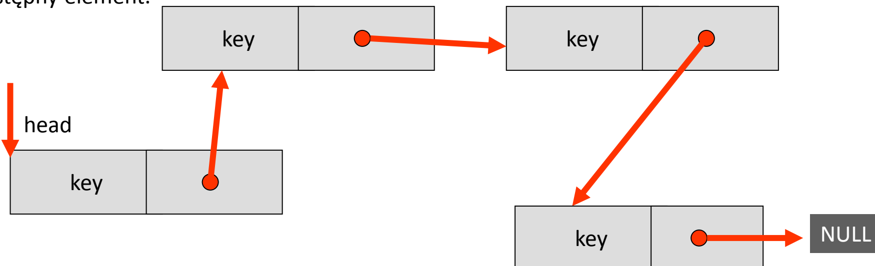
Abstrakcyjne typy danych: lista

Elementy zawierają informację na temat pozycji kolejnego elementu

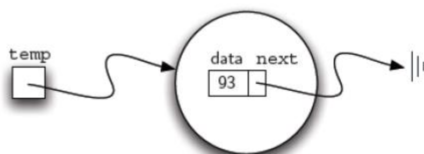


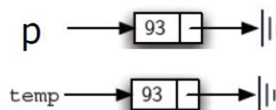
Kluczowa jest wiedza co do położenia pierwszego elementu. Informacja w nim zawarta pozwoli odczytać całą listę.

Wszystkie elementy są podobne: niosą wpisane dane (key) oraz informację, gdzie jest następny element.



Węzeł – podstawowy blok budujący listę wiązaną i łańcuch węzłów

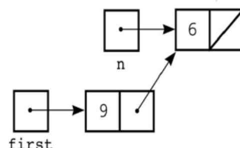




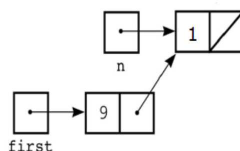

```
p = Node(93)
temp = Node(93)
```

Wiązanie węzłów w łańcuch

```
n = Node(6)
first = Node(9)
first.set_next(n)
```




```
n.set_data(1)
print(first.get_next().get_data())
```



Implementacja klasy lista w Pythonie

Dlatego w pierwszym kroku definiujemy klasę Node, która będzie reprezentowała pojedynczy element.

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

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

0_lista

1_lista

2_lista

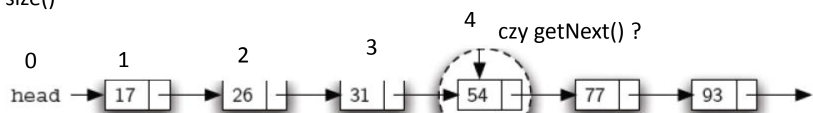
Następnie definiujemy klasę:

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

Niezbędne metody do obsługi tej klasy:
is Empty(), add (item), size(), search(), remove()

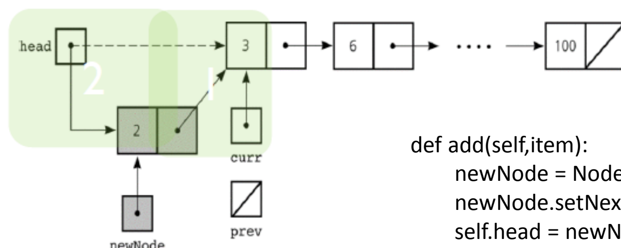
size()

0 1 2 3 4 czy getNext() ?




```
def size(self):
    current = self.head
    count = 0
    while current != None:
      count = count + 1
      current = current.getNext()
    return count
```

add(2)




```
def add(self,item):
    newNode = Node(item)
    newNode.setNext(self.head)
    self.head = newNode
```

PROJEKT F

Zaimplementować klasy odpowiadające kolejce priorytetowa na:
- liście Pythona,
- kopcu max opartym o listę Pythona
- liście z dowiązaniami,

Przygotować moduły z wielorakimi testami poprawności działania klas. Przeprowadzić testy wydajności obsługi kolejki, czyli zmierzyć czas tworzenia i czas obsługi każdej z wyżej przygotowanej implementacji. Testy wydajności przeprowadzić dla rozmiarów: 2**i z i=10… 16

Wyniki przedstawić graficznie(wykres log-log) i skomentować.