# 

# 2. Real time deque

V této kapitole se budeme zabývat implementací deque (double-ended queue), u které všechny operace s ní spojené probíhají v reálném čase. Toto je typ fronty, u které můžeme operace přidání prvku a odebírání prvku provádět jak zleva, tak zprava. Funkce, které řeší tyto operace budeme nazývat pop\_left a pop\_right pro odebírání a push\_left, push\_right pro přidávání prvků. Dále bude k dispozici funkce new a is\_empty. Funkce new vrátí uživateli novou prázdnou frontu. Funkce is\_empty přijme na vstupu konkrétní frontu a zjistí, zda je tato fronta prázdná, či nikoli. Jako základ pro tuto implementaci použijeme pro reprezentaci fronty, stejně jako v předchozí kapitole, dva stacky. Pro tyto stacky budeme dále používat označení LHS pro stack levý a RHS pro stack pravý.

Tímto zajistíme, že operace pop i push budou prováděny v čase O(1). Problém ale nastane v případě, že se jeden ze stacků operací pop\_left nebo pop\_right vyprázdní a my budeme chtít odebírat z daného konce fronty. Poté bychom museli přetočit prvky neprázdného stacku a až potom provést odebírání. Cílem této implementace je rozdělit tento přesun prvků do operací prováděných nad frontou. Proto se tedy bude fronta nacházet v různých stavech a každý tento stav bude realizovaný jednou třídou. Tyto stavy budou čtyři.

## 2.1 Přetáčení prvků

Nyní se podíváme na samotné přetáčení prvků. Ve stavech 0 a 1 k přetáčení nedochází. Až při přechodu ze stavu 1 do stavu 2 kroky přetáčení začínají. To, že fronta má přejít do stavu 2 poznáme tak, že dojde k porušení následujícího invariantu.

Pro vytvoření nových stacků je potřebné vykonat následující kroky:

a.

Přetočení 2m + k – 1 prvků stacku B na pomocný seznam značený aux\_B

b.

Přetočení stacku S na pomocný seznam značený aux\_S

c.

Přetočení zbývající části stacku B na new\_S

d.

Přetočení aux\_B na new\_B

e.

Napojení aux\_S na new\_S

Tyto kroky jsou rozděleny mezi stavy fronty následovně. Ve stavu fronty 2 probíhají kroky a, b.

Ve stavu fronty 3 probíhají zbývající kroky c, d a nakonec také e (s krokem e je možno začít až po dokončení kroku c). Po dokončení posledního kroku tedy napojení aux\_S na new\_S máme připraveny seznamy new\_S a new\_B, musíme ale taky počítat s tím, že do fronty budou během přetáčení vkládány nové prvky. Tyto prvky se budou ukládat do seznamu extra\_S a extra\_B. Jelikož spojení seznamů new\_S a extra\_S, new\_B a extra\_B nelze jednoduše zrealizovat, použijeme jako reprezentaci stacku dvojici seznamů. V následující části se podíváme na implementaci jednotlivých tříd.

Obrázek 7 – kroky přetáčení a, b

Obrázek 8 – kroky přetáčení c, d

Obrázek 9 – krok přetáčení e

## 2.2 Element

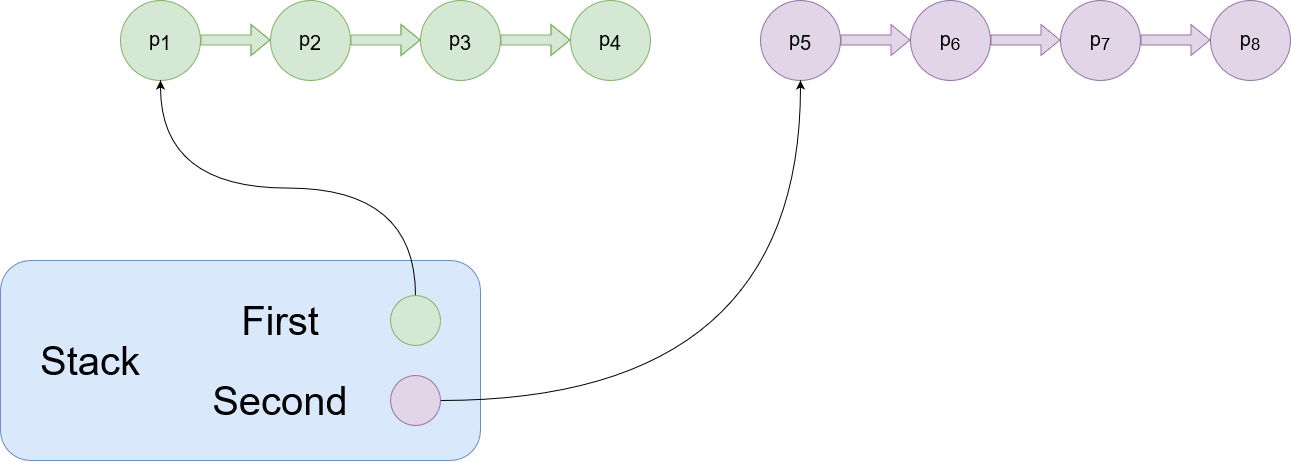
Prvky fronty budou reprezentovány jako objekty třídy Element.



## 2.3 Stack

Stack je složen ze dvou seznamů tvořených prvky třídy Element, přičemž vždy uchováváme jen ukazatel na první prvek. Tyto ukazatelé jsou v implementaci značeny jako first pro první část a second pro část druhou.

Obrázek 10 – struktura stacku



class Stack(object):

    def \_\_init\_\_(self,first,second):

        self.first = first

        self.second = second

Samotný stack má dvě metody:

Metoda **get** vrátí první prvek stacku. Tedy zkontroluje, jestli je první část stacku prázdná. Pokud ano vrátí první prvek druhé části, tedy ukazatel second. V případě, že první část prázdná není, vrátí ukazatel first.

    def get(self):

        if self.first is None:

            return self.second

        return self.first

Metoda **next** vrátí novou verzi stacku s odebraným prvním prvkem stacku.

Pokud první část není prázdná, vezmeme ukazatel first a kontrolujeme, zda za ním existuje další prvek. Pokud ano, vrátíme novou verzi stacku, pričemž první část bude následující prvek za first tedy first.next a druhá část zůstavá identická jako u předchozí verze. Pokud ovšem další prvek neexistuje, znamená to, že další prvek stacku je až v jeho druhé části. Tedy u nové verze pouze použijeme None pro první část a druhá část zůstane stejná.

Nyní probereme případ, kdy je první část prázdná. Zkontrolujeme tedy, zda se něco nachází v druhé části. Pokud ne, stack je prázdný a metoda next nedává smysl, vrátíme None. Pokud se ovšem nějaký prvek v druhé části nachází, vrátíme novou verzi stacku s první částí None a jako ukazatel na druhou část nastavíme následují prvek předchozí verze části second.



## 2.4 Deque\_0

Tato třída je podobou fronty ve stavu 0. V tomto stavu je fronta reprezentována jako jeden seznam. Fronta může obsahovat v tomto stavu nula až tři prvky.

class Deque\_0(object):

    def \_\_init\_\_(self, list):

        self.list = list

        self.state = 0

Operace push zleva realizujeme tak, že vytvoříme nový seznam, vložíme zde požadovaný prvek a poté vložíme všechny prvky z předchozí verze deque. Pokud se bude provádět operace push zprava, budeme postupovat podobně, pouze nejprve do nového seznamu vložíme prvky staré verze a až potom nový prvek. Po provedení těchto operací zkontrolujeme délku nového seznamu. Pokud je délka < 4, pouze vrátíme novou verzi fronty ve stavu 0 s nově vytvořeným seznamem. Pokud ovšem délka = 4, rozpůlíme list na dvě části, z těchto části vytvoříme nový levý stack, pravý stack a vrátíme frontu ve stavu 1.

Operace pop zleva a zprava jsou implementované následujícím způsobem. Pokud odebíráme zleva, vytvoříme nový seznam a z předchozí verze zkopírujeme všechny prvky kromě prvku s indexem 0. Při odebírání zprava zkopírujeme do nového seznamu všechny prvky kromě prvku posledního. Toto lze v pythonu jednoduše zajistit pomoci operátoru [-1]. Jelikož odebíráním nelze překročit maximální velikost fronty, v tomto stavu pouze vrátíme novou verzi fronty ve stavu 0.

def push\_left\_0(value, dq):

    n\_dq\_list = []

    n\_dq\_list.append(value)

    for x in dq.list:

        n\_dq\_list.append(x)

    if len(n\_dq\_list) == 4:

        LHS = Stack( Element(n\_dq\_list[0], Element(n\_dq\_list[1], None)), None )

        RHS = Stack( Element(n\_dq\_list[3], Element(n\_dq\_list[2], None)), None )

        return Deque\_1(LHS, RHS, 2, 2)

    return Deque\_0(n\_dq\_list)

def pop\_left\_0(dq):

    if len(dq.list) == 0:

        return None, dq

    n\_dq\_list = []

    for i in range(len(dq.list)):

        if i == 0:

            continue

        n\_dq\_list.append(dq.list[i])

    return dq.list[0], Deque\_0(n\_dq\_list)

## 2.5 Deque\_1

Tato třída reprezentuje frontu stavu 1.

class Deque\_1(object):

    def \_\_init\_\_(self, LHS, RHS, LHS\_length, RHS\_length):

        self.LHS = LHS

        self.RHS = RHS

        self.LHS\_length = LHS\_length

        self.RHS\_length = RHS\_length

Operace push zleva se provádí následovně.Vytvoří se nový LHS a do něj se na jeho první část *viz Stack*. napojí nový Element s hodnou value, zvětší se délka daného stacku o jeden a poté se zkontroluje následující invariant

Pokud invariant není porušen, vracíme novou verzi fronty ve stavu 1. Pokud ovšem invariant porušen je, dojde k přechodu do stavu 2. Vytvoříme frontu stavu 2 s již přidaným prvkem. Poté je zapotřebí provést na této frontě kroky přetáčení. Obdobně probíhá operace push zprava.

Operace pop zleva. Uložíme si hodnotu prvku vybraného z LHS a vytvoříme nový stack zavoláním metody next() na LHS předchozí verze fronty. Snížíme délku daného stacku a zkontrolujeme invariant stejně jako u operace push. Pokud je porušena podmínka zkontrolujeme podmínku . Je-li porušena tato podmínka, znamená to, že je LHS prázdný, ale současně víme, že v RHS jsou maximálně tři prvky. Vytvoříme tedy z pravého stacku frontu stavu 0 a vrátíme společně s hodnotou vybraného prvku. Pokud první podmínka porušena není, fronta přechází do stavu 2 a je potřeba provést počáteční kroky přetáčení obdobně jako u operace push.



## 2.6 Deque\_2

class Deque\_2(object):

    def \_\_init\_\_(self, side, S, B, S\_origin, B\_origin, aux\_S, aux\_B, extra\_S, extra\_B, copy\_S, copy\_B, extra\_S\_size, extra\_B\_size, aux\_counter):

        self.side = side

        self.S = S

        self.B = B

        self.S\_origin = S\_origin

        self.B\_origin = B\_origin

        self.aux\_S = aux\_S

        self.aux\_B = aux\_B

        self.extra\_S = extra\_S

        self.extra\_B = extra\_B

        self.copy\_S = copy\_S

        self.copy\_B = copy\_B

        self.extra\_S\_size = extra\_S\_size

        self.extra\_B\_size = extra\_B\_size

        self.aux\_counter = aux\_counter

Operace push v tomto stavu napojuje nový prvek na pomocné seznamy extra\_s a extra\_b. Nejprve zjistíme, jak je orientovaná fronta podle atributu side. Pokud je side = True znamená to, že seznam S je vlevo a seznam B vpravo. Pokud je side = False, orientace je B ->S .

Napojíme prvek na příslušný extra seznam a zvětšíme jeho velikost. Provedeme kroky přetáčení a vrátíme novou verzi fronty ve správném stavu.

Operace pop je o něco složitější. V prvním případě je příslušný seznam extra neprázdný. Potom pouze vybereme prvek z tohoto seznamu a vytvoříme novou frontu příslušného stavu a provedeme na ní 4 kroky aktuálního přetáčení.

Pokud je ale daný seznam extra prázdný, musíme vybírat prvky z příslušného seznamu origin.

## 2.7 Deque\_3

class Deque\_3(object):

    def \_\_init\_\_(self, side, B, S\_origin, B\_origin, aux\_S, aux\_B, extra\_S, extra\_B, copy\_S, copy\_B, extra\_S\_size, extra\_B\_size, new\_S, new\_B, S\_size, B\_size):

        self.side = side

        self.B = B

        self.S\_origin = S\_origin

        self.B\_origin = B\_origin

        self.aux\_S = aux\_S

        self.aux\_B = aux\_B

        self.extra\_S = extra\_S

        self.extra\_B = extra\_B

        self.copy\_S = copy\_S

        self.copy\_B = copy\_B

        self.extra\_S\_size = extra\_S\_size

        self.extra\_B\_size = extra\_B\_size

        self.new\_S = new\_S

        self.new\_B = new\_B

        self.S\_size = S\_size

        self.B\_size = B\_size

        self.state = 3

Operace push i pop jsou prováděny obdobně jako ve stavu 2, pouze s jednou změnou, a to u operací pop\_left a pop\_right. Může totiž nastat následující případ, řekněme, že odebíráme zleva nad frontou se side = true tedy uspořádání je S -> B. Měli bychom tedy vybírat ze seznamu Extra\_s. Ten je ovšem prázdný, a tudíž musíme vybírat ze seznamu S\_origin. Může ale nastat situace, že všechny prvky jsou již přesunuty do new\_s, tudíž copy\_S je 0, ale fronta je zároveň stále ve stavu 3, protože seznam new\_b ješte není zcela vytvořený, tzn. copy\_b > 0. Potom musíme postupovat následovně. Odebírání provedeme na již vytvořeném new\_S. Použijeme tedy new\_s.next jako nový new\_S, ale také nesmíme zapomenout snížit velikost new\_S.





## 2.8 Uživatelské rozhraní

Pro uživatele fronty byly vytvořeny následující funkce:

### 2.8.1 push\_left(value, dq)

Tato funkce vytvoří novou verzi fronty s přidaným prvkem s hodnotou value zleva.

Vstupy:

dq – objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

value – hodnota prvku

Výstupy:

objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

### 2.8.2 push\_right(value, dq)

Tato funkce vytvoří novou verzi fronty s přidaným prvkem s hodnotou value zprava.

Vstupy:

dq – objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

value – hodnota prvku

Výstupy:

objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

### 2.8.3 pop\_left(dq)

Tato funkce vytvoří novou verzi fronty odebráním prvku zleva.

Vstupy:

dq – objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

Výstupy:

objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

### 2.8.4 def pop\_right(dq)

Tato funkce vytvoří novou verzi fronty odebráním prvku zprava.

Vstupy:

dq – objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

Výstupy:

objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

### 2.8.5 def new()

Tato funkce vytvoří prázdnou frontu.

Vstupy:

Výstupy:

objekt třídy Deque\_0

### 2.8.6 def is\_empty(q)

Tato funkce zkontroluje, zda je fronta prázdná.

Vstupy:

q – objekt třídy Deque\_0, Deque\_1, Deque\_2 nebo Deque\_3

Výstupy:

True/False