Obsah

[1. Real-time queue 1](#_Toc70431900)

[1.1 Implementace 3](#_Toc70431901)

[1.2 Element 3](#_Toc70431902)

[1.3 QueueZero 3](#_Toc70431903)

[1.4 QueueOne 5](#_Toc70431904)

[1.5 QueueTwo 8](#_Toc70431905)

[1.6 Uživatelské rozhraní 10](#_Toc70431906)

[1.6.1 Enqueue(q, value) 10](#_Toc70431907)

[1.6.2 Dequeue(q) 10](#_Toc70431908)

[1.6.3 Get\_empty\_queue() 10](#_Toc70431909)

[2. Real time deque 11](#_Toc70431910)

[2.1 Přetáčení prvků 11](#_Toc70431911)

[2.2 Element 12](#_Toc70431912)

[2.3 Stack 12](#_Toc70431913)

[2.4 Deque\_0 13](#_Toc70431914)

[2.5 Deque\_1 15](#_Toc70431915)

[2.6 Deque\_2 17](#_Toc70431916)

[2.7 Deque\_3 18](#_Toc70431917)

[2.8 Uživatelské rozhraní 21](#_Toc70431918)

[2.8.1 push\_left(value, dq) 21](#_Toc70431919)

[2.8.2 push\_right(value, dq) 21](#_Toc70431920)

[2.8.3 pop\_left(dq) 21](#_Toc70431921)

[2.8.4 def pop\_right(dq) 21](#_Toc70431922)

[2.8.5 def new() 22](#_Toc70431923)

[2.8.6 def is\_empty(q) 22](#_Toc70431924)

[3. Stack 23](#_Toc70431925)

[3.1 Element 24](#_Toc70431926)

[3.2 Stack 24](#_Toc70431927)

[3.3 NEW 24](#_Toc70431928)

[3.4 EMPTY 24](#_Toc70431929)

[3.5 TOP 25](#_Toc70431930)

[3.6 POP 25](#_Toc70431931)

[3.7 PUSH 25](#_Toc70431932)

[3.8 LENGTH 25](#_Toc70431933)

[3.9 FIND 26](#_Toc70431934)

[3.10 POP\_MULTI 26](#_Toc70431935)

# 1. Real-time queue

Tato část bakalářské práce se bude zabývat implementací datové struktury fronty, a to konkrétně její real-time implementací. Nejprve si řekneme, co to fronta vlastně je.

Fronta je datová struktura, která je určená pro ukládání a vybírání prvků v takovém pořadí, ve kterém prvek, který byl přidán nejdříve, bude také nejdříve vybrán. Toto dobře vystihuje anglická zkratka FIFO tedy first in, first out.

Definujme pro frontu následující operace:

1. Enqueue – přidání prvku na konec fronty

2. Dequeue – odebrání prvku ze začátku fronty

Pro implementaci real-time fronty potřebujeme, aby obě tyto operace probíhaly v konstantním čase. Pokud bychom reprezentovali frontu jedním seznamem, inhed narazíme na následující problém.

Odebrání prvků jsme sice schopni provést v konstatním čase, ale pro operaci přidání bychom museli nejprve projít celý seznam a až poté připojit nakonec seznamu nový prvek.

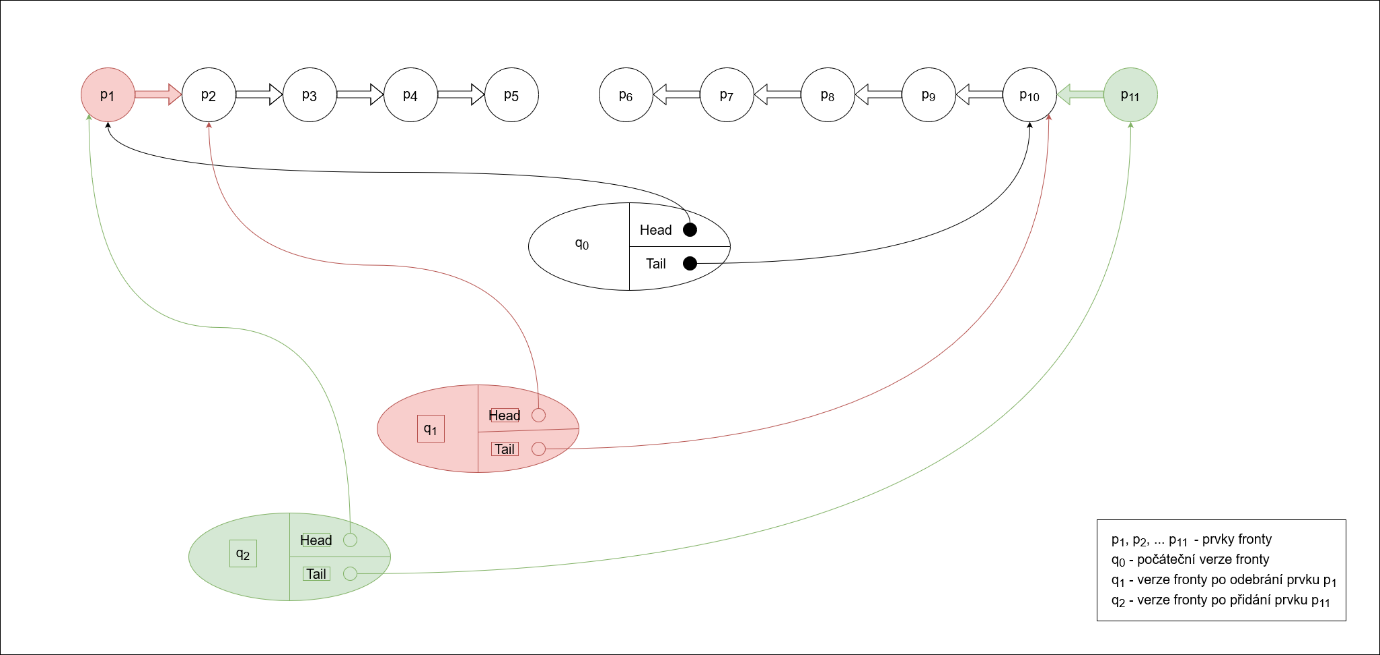
Obrázek 1 – fronta reprezentována jedním seznamem

Diagram

Description automatically generated

Tento problém vyřešíme tak, že frontu budeme reprezentovat dvěma seznamy. Tyto seznamy budeme nazývat head a tail.

Obrázek 2 – fronta reprezentována dvěma seznamy



Díky této změně jsme již schopni provést jak operaci odebírání, tak přidání v konstantním čase.

Pro operaci dequeue budeme používat seznam head a pro operaci enqueue budeme používat seznam tail. Problém ale nastane v případě, pokud by se seznam head vyprázdnil a my bychom dále chtěli odebírat z fronty. Potom by bylo nutné přetočit prvky ze seznamu tail a až poté odebrat prvek z fronty. Toto přetáčení by zabralo n kroků, kde n je délka seznamu tail, což by ovšem porušilo podmínku provedení operací fronty v konstantním čase. Proto v implementaci dovolíme, aby byl head prázdný pouze v případě, že je prázdný i tail, tzn. když je fronta prázdná. Cílem bude tedy rozdělit přetáčení prvků z tail tak, že při operacích enqueue a dequeue provedeme i část tohoto přetáčení.

Přetáčení se bude skládat ze tří kroků:

a) přetočením tail na nový seznam head

b) přetočením head na pomocný seznam head reversed

c) připojením přetočením head reversed na nový head

Všimněme si, že kroky a) a b) lze provádět souběžně. Během tohoto procesu se bude podoba fronty měnit, proto musíme zajistit, aby bylo možné zjistit, jak aktuální podobu fronty, tak aktuální podobu přetáčení. Pro přidávání prvků proto vytvoříme nový seznam tail. Při přetáčení se tedy prvky budou přidávat místo starého tailu na tail nově vytvořený. Odebírání bude řešeno pomocí dvou listů head. Zatímco jeden bude používán pro realizaci přetáčení, druhý bude sloužit pro odebírání prvků. Samozřejmě tím, že budou některé prvky při procesu přetáčení odebrány, ne všechny prvky se tak budou z head reversed na nový head napojovat. Proto budeme používat counter, který nám určí, kolik prvků je třeba při kroku c) připojit.

Obrázek 3 – fronta při přetáčení a,b

Obrázek 4 – fronta po dokončení přetáčení a,b

Obrázek 5 – fronta při přetáčení c

Obrázek 6 – fronta po dokončení přetáčení c

Dále potřebujeme určit, kdy proces přetáčení započne. S přetáčením začínáme v momentě, kdy bude velikost seznamu tail větší než velikost seznamu head. Dokud tedy fronta splňuje podmínku k přetáčení nedochází a fronta je tvořena pouze seznamy head a tail. Po porušení této podmínky začínáme s procesem přetáčení.

Abychom zabránili tomu, že se head vyprázdní dříve než je proces přetáčení dokončen, je třeba vymyslet, jak rozložíme kroky přetáčení mezi n operací fronty (enqueue, dequeue). Dejme tomu, že při uvedení fronty do tohoto procesu obsahuje seznam head n prvků a seznam tail prvků. Pro vykonání kroků a) a b) je tedy potřeba přetočení prvků. Připojení head reversed na nový head docílíme v nejhorším případě přetočením n prvků. Musíme tedy rozdělit 2 kroků mezi n operací fronty. Toho docílíme tak, že při přechodu do procesu přetáčení provedeme jeden krok a při každé další operaci fronty provedeme další dva kroky přetáčení. Po dokončení celého procesu máme připravený nový seznam head a nový seznam tail. podmínka je opět splněna, nahradíme tedy staré head a tail novými a můžeme se vrátit k reprezentaci fronty jen pomocí head a tail.

## 1.1 Implementace

Jelikož fronta má různou vnitřní strukturu podle toho, zda se nachází v procesu přetáčení a také podle toho v jakém jeho kroku se nachází, rozdělíme frontu do stavů. Tyto stavy budou tři a jsou reprezentovány třídami QueueZero, QueueOne, QueueTwo.

## 1.2 Element

Pro potřeby implementace byla vytvořena třída Element, která představuje jeden prvek ve frontě.

Tato třída obsahuje value tedy hodnotu prvku a také next, což je ukazatel na další prvek.



## 1.3 QueueZero

Tato třída je podobou fronty, která se nenachází v procesu přetáčení.

V tomto stavu obsahuje fronta head a tail, tedy dva seznamy tvořící vlastní frontu. Dále lendiff, který vyjadřuje rozdíl mezi velikostí seznamu head a velikostí seznamu tail a který nám bude určovat jak dlouho fronta v tomto stavu zůstane. Díky lendiff nemusíme udržovat velikosti obou seznamů. Podmínku ..... nahradíme podmínkou, kdy lendiff = 0. Dále třída QueueZero obsahuje state, který číselně vyjadřuje stav fronty. Vlastnost state bude také součástí dalších tříd a je použita k rozpoznání, ve kterém stavu se fronta nachází.



Operace enqueue vytvoří nový tail napojením nového prvku na starý tail a vrátí novou verzi fronty. Na začátku se také zkontroluje lendiff a v případě, že je roven nule, se vytvoří fronta třídy QueueOne a operace přidání prvku s value se zavolá až na tento vytvořený objekt.



Operace dequeu vytvoří novou verzi fronty pomyslným odebráním prvku ze seznamu head a vráti ji společně s hodnotou odebraného prvku. Obdobně se na začátku zkontroluje lendiff a případně se operace odebírání provede až v dalším stavu. Také se zkontroluje, zda již fronta není prázdná.



Můžeme si všimnout, že obě operace, jak enqueue, tak dequeue snižují hodnotu lendiff.

## 

## 1.4 QueueOne



V tomto stavu začneme s přetáčením prvků. Budou se provádět následující kroky přetáčení:

a) přetáčením prvků z head na head\_reversed

b) přetáčením prvků z tail na n\_head

Dokud se v head nebo tail vyskytují nějaké prvky, fronta zůstane v prvním stavu.

Operace enqueue probíhá následovně. V první části se zkontroluje podmínka, zda lendiff = 0. Pokud ano, víme, že nastal přechod ze stavu 0 do stavu 1, tedy že funkce byla zavolána z enqueue\_zero. Takže je potřeba po vložení prvku provést jeden krok přetáčení. Krok a) poté provedeme běžným způsobem. Krok přetáčení b) je zjednodušen. Z nově vkládaného prvku rovnou vytvoříme n\_head. Poté vrátíme novou verzi fronty ve stavu 1.

Pokud se nejedná o přechod mezi stavy 0 a 1, postupuje se následnovně, nejprve se na n\_tail napojí nový prvek a sníží se hodnota lendiff. Dále se provedou dvakrát kroky přetáčení a) a b).

V případě, že se během provedení vyprázní oba seznamy head a tail, přejde fronta do stavu 2 a my tedy vrátíme jako novou verzi objekt QueueTwo. Pokud ne, pouze vrátíme novou verzi fronty ve stavu 1.

Nyní operace dequeue. Opět nejprve zjistíme, zda-li se jedná o přechod ze stavu 0 do stavu 1. Pokud ano, provedeme počáteční krok přetáčení. U kroku a) si musíme dát pozor na to, že je potřeba z fronty nejprve odebrat jeden prvek. Tudíž s přetáčením začneme až u následujícího prvku seznamu head tedy head.next. Krok b) potom provedeme běžným způsobem. Pokud se nejedná o přechod mezi stavy, postupuje se následovně. Nejprve vytvoříme nový head\_origin a snížíme hodnotu delta\_for\_copy. Následně provedeme kroky přetáčení a vrátíme novou verzi fronty stejně tak, jako u operace pro přidávání prvků.



## 

## 1.5 QueueTwo



V tomto stavu dochází k poslednímu kroku procesu přetáčení, což je napojení head\_reversed na n\_head, které byly oba vytvořeny v předchozím stavu. Hodnota delta\_for\_copy, určuje kolik prvků je ještě potřeba napojit.

Přidávání prvků je řešeno jednoduše pouze připojením nového prvku na n\_tail a opět zvýšením hodnoty lendiff.



Při odebírání prvků se pouze sníží hodnota delta\_for\_copy a použijeme nový head\_origin přes ukazatel next. Když je proces napojení dokončen, tzn. delta\_for\_copy je rovna 0, seznamy n\_head a n\_tail jsou zcela vytvořené a mohou nahradit původní head a tail. Fronta přejde zpátky do stavu 0.



## 1.6 Uživatelské rozhraní

Pro uživatele fronty byly vytvořeny následující funkce:

### 1.6.1 Enqueue(q, value)

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

Vstupy:

q – objekt třídy QueueZero, QueueOne nebo QueueTwo

value – hodnota prvku

Výstupy:

objekt třídy QueueZero QueueOne nebo QueueTwo

### 1.6.2 Dequeue(q)

Tato funkce vytvoří novou verzi fronty s odebraným prvním prvkem

Vstupy:

q – objekt třídy QueueZero, QueueOne nebo QueueTwo

Výstupy:

value -Hodnota prvku

objekt třídy QueueZero, QueueOne nebo QueueTwo

### 1.6.3 Get\_empty\_queue()

Tato funkce vytvoří prázdnou frontu a vrátí ji uživateli.

Vstupy:

Výstupy:

objekt třídy QueueZero (prázdná fronta)

# 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

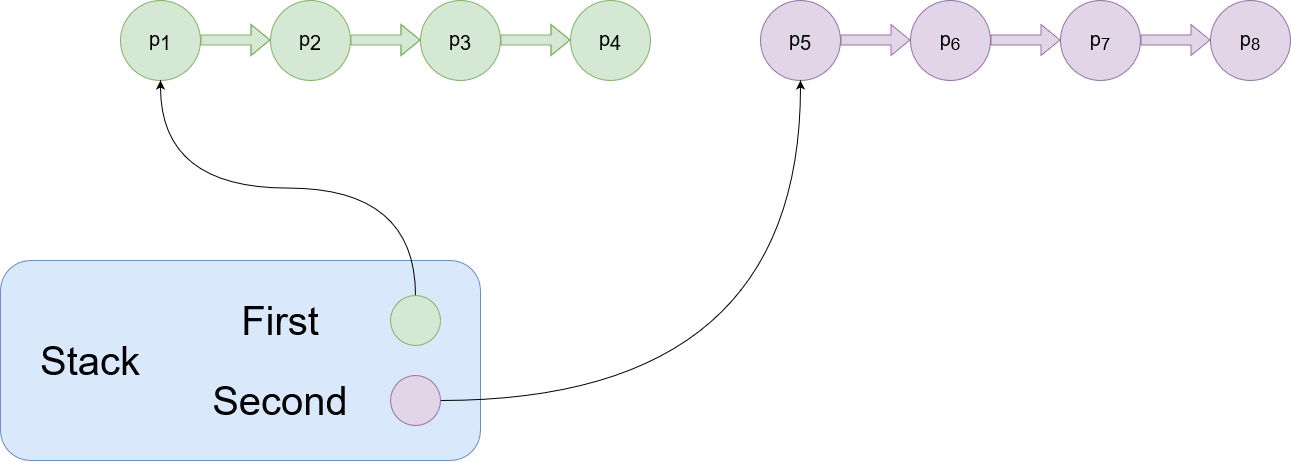
Prvky fronty budou reprezentovány jako objekty třídy Element.



## 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

# 3. Stack

V této kapitole si popíšeme funkcionální implementaci Random-acces stacku.

Stack nebo-li zásobník jako datová struktura se vyznačuje tím, že prvek, který byl zařazen do stacku jako poslední, bude vybrán ze stacku jako první. Toto popisuje anglická zkratka LIFO – tedy Last in First Out.

Pro jednoduchý stack jsou definované tyto operace:

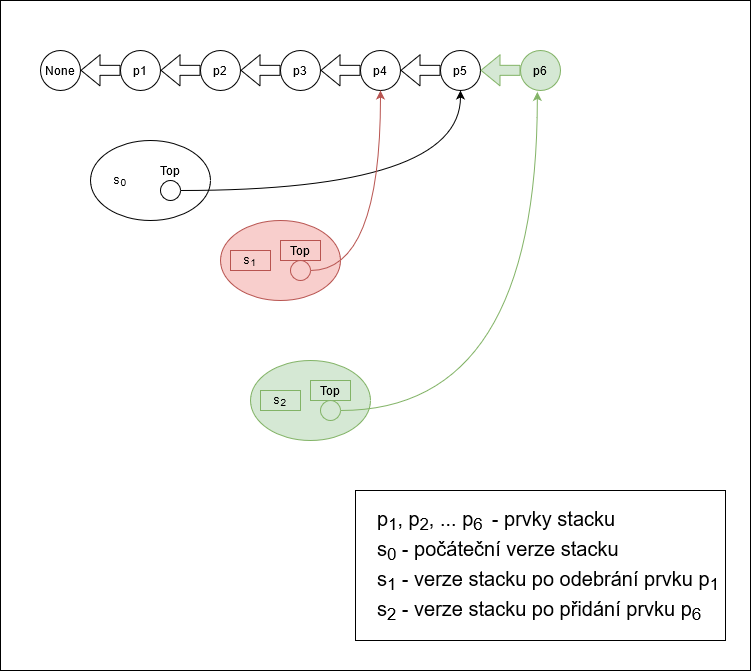
Pop – odstraní poslední prvek ze stacku

Push – vloží nový prvek nakonec stacku

Nejprve začneme implementací jednoduchého stacku. Stack bude tvořen jedním seznamem.

Operaci pop provedeme snadno v konstantním čase. Stejně tak můžeme provést operaci push pouhým napojením nového prvku na vrchol stacku a opět tedy v konstantním čase.

Obrázek 11 – operace pop a push na stacku



Máme tudíž základní podobu stacku a můžeme provést úpravy tak, abychom umožnili přístup ke

k-tému prvku stacku. Za tímto účelem vytvoříme další operaci.

Find – vrátí nám k-tý prvek stacku

Procházíme tedy postupně seznam stacku do té doby, než nalezneme k-tý prvek.

Tato operace nám zabere length – k počet kroků, kde length je délka stacku a k je index hledaného prvku. Proto vylepšíme vnitřní strukturu stacku tak, že prvky seznamu budou mít kromě ukazatele na následující prvek seznamu ještě další ukazatel. Tento ukazatel bude určitá zkratka na konkrétní prvek seznamu. Nový větší skok vždy vytvoříme u prvku .

Obrázek 12 – stack s vytvořenými skoky

## 3.1 Element

Třída Element reprezentuje prvky stacku. Každý tento prvek obsahuje následující. Hodnotu prvku value, index k určující pořadí, ukazatel na prvek následující a ukazatel na skok.



## 3.2 Stack

V samotném stacku uchováváme pouze ukazatel na prvek nacházející se na vrcholu.



## 3.3 NEW

Tato funkce nám bude sloužit k vytvoření nového prázdného stacku.



## 3.4 EMPTY

Tato funkce zkontroluje zda je stack na vstupu prázdný.



## 3.5 TOP

Funkce, která vrátí hodnotu prvku na vrcholu stacku.



## 3.6 POP

Funkce slouží k odebrání prvku z vrcholu stacku.



## 3.7 PUSH

Funkce připojí nový prvek na vrchol stacku. Musí se ale zjistit jaký se má tomuto prvku přiřadit jump.

Pomocí jump přejdeme z vrcholu stacku na prvek p1 a uchováme si velikost tohoto skoku. Poté provedeme z tohoto prvku p1 další skok pomocí jump na prvek p2 a opět si uchováme velikost skoku. Porovnáme velikosti skoků a pokud jsou si rovné, nově přidanému prvku nastavíme jump na prvek p2. Pokud si rovné nejsou jump bude nastaven na vrchol stacku.



## 3.8 LENGTH

Funkce vrátí aktuální velikost stacku. Díky uchovávání indexu každého prvku pouze přečteme index vrcholu stacku a vrátíme tuto hodnotu.



## 3.9 FIND

Funkce vrátí hodnotu k-tého prvku stacku. Algoritmus je podobný algoritmu půlení intervalů.



## 3.10 POP\_MULTI

Funkce vrátí nový stack tvořen zpodními k prvky. Algoritmus identický s funkcí find.

