

Datu struktūras

| | |
|--|----|
| Saraksts [List] | 2 |
| Rindas | 2 |
| Steks | 4 |
| Kortežs (Tuple)..... | 6 |
| Kopa {Set}: | 7 |
| Vārdnīca {Dictionary}: -..... | 8 |
| Grafi | 8 |
| Meklēšana plašuma (<i>Breadth-first search</i>)..... | 9 |
| Meklēšana dziļumā (Depth-first search, DFS) | 11 |
| Deikstra algoritms..... | 11 |
| Koks ir grafa zars..... | 12 |

Datu struktūra ir klase, kura norāda ar kādām metodēm tiek apvienoti datu vienumi, nodrošinot piekļuvi to apstrādei, uzglabāšanai.

Algoritms + Datu struktūra = Programma (N.Virts).

Izvēloties datu struktūru iegūst efektīvu vai neefektīvu programmu.

Datu tipi:

- ❖ Lineāri : elementi veido lineāru sarakstu. Piemēram, saraksts, steki un rindas.
- ❖ Nelineāri, elementu saraksts sazarojas. Piemēram: grafs un koks.

Noteikta uzdevuma veikšanai Python ir iebūvētas 4 datu struktūras:

Saraksts [List]: [sakārtotu, maināmu, dublējamu, viena veida indeksētu elementu kopa (masīvam līdzīgas struktūras)].

Kortežs (Tuple): (sakārtotu, nemaināmu, dublējamu viena veida neindeksētu elementu kopa (nemaināms saraksts))

Kopa {Set}: {nesakārtotu, nemaināmu, nedublējamu neindeksētu objektu kopa, var pielikt/noņemt vienumus}

Vārdnīca {Dictionary}: {nesakārtotu (Līdz Python 3.6 versijai, no 3.7 sakārtotu), maināmu, nedublējamu atslēgu/vērtību pāru kolekcija}

Saraksts [List]

[sakārtotu, maināmu, dublējamu, viena veida indeksētu elementu kopa (masīvam līdzīgas struktūras)].

| | | | | |
|------------------|---|-----|-----|-----|
| elementa numurs | → | [0] | [1] | [2] |
| elementa vērtība | → | 5 | 7 | 9 |

Masīvam līdzīgas struktūras, savstarpēji saistītu objektu uzglabāšanai, kuru bieži nav jāpārskata, pievienot var tikai saraksta beigās,

```
x=[3,5]
print (type(x))
```

<class 'list'>

Populārākās metodes:

len() – nosaka saraksta garumu

append() – pielikt elementu, dinamiski palielinot sarakstu

pop() – noņemt elementu, dinamiski samazinot elementu
Vingrinājumi:

1. Izveidot sarakstu no 3 elementiem, pievienot pēdējo elementu, nodzēst otro elementu. Pēc katras darbības izdrukāt sarakstu funkcijā

```
1 def drukaSarakstu(saraksts):
2     print("saraksts garums = ",len(saraksts))
3     for x in saraksts:
4         print(x)
5     print()
6
7 saraksts = ["Vārna", "Gulbis", "Žagata"]
8 drukaSarakstu(saraksts)
9 # pievienot sarakstam vienību
10 saraksts.append("Kaija")
11 drukaSarakstu(saraksts)
12 #no saraksta izņemt 3.elementu
13 saraksts.pop(2)
14 drukaSarakstu(saraksts)
15
16 skaitli = [1,2,3]
17 drukaSarakstu(skaitli)
18
```

```
saraksts garums = 3
Vārna
Gulbis
Žagata

saraksts garums = 4
Vārna
Gulbis
Žagata
Kaija

saraksts garums = 3
Vārna
Gulbis
Kaija

saraksts garums = 3
1
2
3
```

Uzdevumi:

1. Izveidojiet sarakstu ar 10 blakus stāvošiem veseliem skaitļiem. Tiem, kuriem žurnālā pāra skaitlis – atstāji tikai pāra skaitli, tiem, kuriem nepāra skaitlis atstāji tikai nepāra skaitli.
2. Izdrukāji lielāko un mazāko skaitli.

Padziļināti

3. Izveidojiet divus viena tipa sarakstus un apvienojiet tos.

Rindas

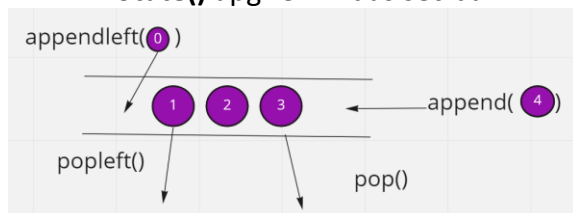
saraksta neindeksēts atvasinājums

Ērti apstrādāt tiešsaistes pasūtījumus, uzglabāt balss pastu, koplietot resursus, piemēram, printeri vai CPU kodolu, jo elementi sakārtoti hronoloģiskā secībā.

Rindas queue (First In First Out) - metodes append() un pop() neefektīvas, jo jāpārbīda visa rinda, tāpēc izmanto divvirzienu rindu deque ar metodēm:

- **append()** pielikt elementus rindas labajā pusē
- **appendleft()** pielikt elementus rindas kreisajā pusē

- **pop()** noņemt rindas labajā pusē
- **popleft()** noņemt rindas kreisajā pusē
- **rotate()** apgriež rindas secību



Vingrinājumi:

1. Divvirzienu rindā ielikt sarakstu ar 3 elementiem, pielikt rindas beigās vienu elementu, rindas sākumā noņemt vienu elementu, pakāpeniski pielikt rindas sākumā 3 elementus vienu pēc otra, pēc katras darbības ar rindu izdrukāt to

```
1 from collections import deque
2 q = deque() #izveido divvirzienu rindu
3
4 q.append(["Vārna", "Gulbis", "Žagata"])
5 print("Rinda= ", q, "\n")
6
7 # pievienot sarakstam elementu
8 q.append("Kaija")
9 print('append("Kaija") = ', q)
10
11 # dzēst no kreisas
12 q.popleft()
13 print("\nq.popleft() = ", q)
14
15 # pievieno kreisajā
16 q.appendleft([1, 2])
17 print("\nq.appendleft([1, 2]) = ", q)
18 q.appendleft(3)
19 print("\nq.appendleft(3) = ", q)
20 q.appendleft(4)
21 print("\nq.appendleft(4) = ", q)
22
```

```
Rinda= deque(['Vārna', 'Gulbis', 'Žagata'])
append("Kaija") = deque(['Vārna', 'Gulbis', 'Žagata', 'Kaija'])
q.popleft() = deque(['Kaija'])
q.appendleft([1, 2]) = deque([1, 2], 'Kaija')
q.appendleft(3) = deque([3, [1, 2], 'Kaija'])
q.appendleft(4) = deque([4, 3, [1, 2], 'Kaija'])
> []
```

2. Turpināt ar to pašu sarakstu: izmest rindas pēdējo elementu, izmest rindas pirmo elementu, apgriež rindas elementus pretējā virzienā, pēc katras darbības ar rindu izdrukāt to

```
21 print("\nPirms dzēšanas = ", q)
22
23 # Dzēsh
24 q.pop()
25 print("\nq.pop() = ", q)
26 q.popleft()
27 print("\nq.popleft() = ", q)
28
29 q.rotate()
30 print("\nq.rotate() = ", q)
```

```
Pirms dzēšanas = deque([4, 3, [1, 2], 'Kaija'])
q.pop() = deque([4, 3, [1, 2]])
q.popleft() = deque([3, [1, 2]])
q.rotate() = deque([1, 2], 3])
> []
```

Uzdevumi:

1. Izveidojiet rindu ar 10 blakus stāvošiem veseliem skaitļiem. Tiem, kuriem žurnālā pāra skaitlis – atstāji tikai pāra skaitli, tiem, kuriem nepāra skaitlis atstāji tikai nepāra skaitli.
2. Izdrukāji lielāko un mazāko skaitli.

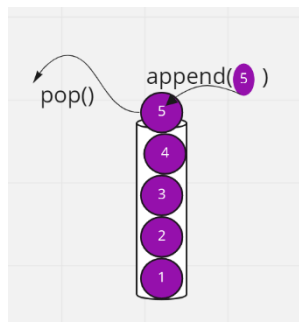
Padziļināti

- <https://clevercode.lv/task/show/kartupelis>
- <https://clevercode.lv/task/show/histogram>

Steks

saraksta neindeksēts atvasinājums

Piedāvā 'Last In First Out' datu pārvaldību, tas kurš pēdējais ienācis – pirmais izies.



Automātiska objekta mērogošana un tīrīšana, uzticama datu uzglabāšanas sistēma, bet ar ierobežotu atmiņu, kura var radīt steka pārpildes kļūdu. Izmanto, veidojot ļoti reaktīvas sistēmas, lai vispirms apstrādātu jaunākos pieprasījumus, iekavu saskaņošanai, Ctrl+Z funkcijas izpilde redaktoros.

Biežāk izmantojamās metodēs:

- **len()** nosaka elementu skaitu stekā
- **append ()** pieliks steka beigās elementu, kurš iekavās (*nevis push()*)
- **pop()** izmetīs augšējo elementu

Vingrinājumi:

1. Stekā ielikt sarakstu no 2 elementiem, pievienot atverošo iekavu, pievienot aizverošo iekavu, izdrukāt visu steku. Pa vienam elementam izņemt no steka, to izdrukājot bez cikliskā operatora.

```
main.py x
1 steks = [7,9] # pinicialize
2 print(steks)
3 steks.append('(')
4 steks.append(')')
5
6 print(steks)
7 print(" stekā iekavu '(' = ",steks.count('('))
8
9 print ("\nStekā ir:")
10 print(steks.pop())
11 print(steks.pop())
12 print(steks.pop())
13 print(steks.pop())
..
```

```
Console Shell
[7, 9]
[7, 9, '(', ')']
stekā iekavu '(' = 1
```

```
Stekā ir:
)
(
9
7
>
```

2. Pārbaudīt vai dotajai simbolu virknei ir pareizi saliktas apaļās iekavas:

```
main.py × Console Shell

1 def parbaudaSteka(v):
2     steks = []
3     rez = "pareizi"
4     for i in v:
5         if i == '(':
6             steks.append('(')
7         elif i == ')':
8             if len(steks) == 0:
9                 rez = "kļūda" ; break
10            elif steks.pop() != '(':
11                rez = "kļūda" ; break
12    print(" stekā iekavu '(' = ",steks.count('('))

13 if steks.count('(')==0:
14     return(rez)
15 else:
16     rez = "kļūda"
17     return(rez)
18

19 v1 = "(7,9)"; print("(7,9) =", parbaudaSteka(v1))
20 v1 = "((7,9)"; print("((7,9) =", parbaudaSteka(v1))
21 v1 = "(7,9))"; print("(7,9)) =", parbaudaSteka(v1))
22
```

```
stekā iekavu '(' = 0
(7,9) = pareizi
stekā iekavu '(' = 1
((7,9) = kļūda
stekā iekavu '(' = 0
(7,9)) = kļūda
```

Uzdevumi:

1. Izmantojot steku, izveidojiet funkciju min().
2. Izveidojiet virkni ar 2 veida iekavām, izmantojot steku, noteikt vai visas iekavas saliktas ievērojot matemātikas likumu.

Padziļināti

- <https://clevercode.lv/task/show/histogram>
- Aplūkot Postfix izmantošanas iespējas.

Kortežs (Tuple)

(sakārtotu, nemaināmu, dublējamu viena veida neindeksētu elementu kopa (nemaināms saraksts))

Korteži strādā ātrāk un drošāk par sarakstiem, jo to datus nevar mainīt, korteža datus ieliek **apaļās iekavās** vai arī var nelikt iekavās.. Izmanto kā atslēgu vārdnīcām.

Vingrinājumi:

1. Izveidot kortežu no 3 elementiem, ielikt divos mainīgajos pirmos 2 mēnešus, trešo var nenorādīt, bet tad jābūt pasvītrojuma zīmei, atrast savu mēnesi kortežā, sakārtot kortežu. Visu izdrukāt.

```
main.py x Console Shell
1 ziema = 'decembris', 'janvāris', 'februāris'
2 m1,m2,_ = ziema
3 print("ši gada mēneši = ",m2,_)
4 print("otrais no labās puses",ziema[-2])
5 print('janv' in ziema) #mekle kortezhaa
6 print("korteža = ",ziema)
7 ziemaSakartota = tuple(sorted(ziema))
8 print("korteža = ",ziemaSakartota)
```

```
ši gada mēneši = janvāris februāris
otrais no labās puses janvāris
False
korteža = ('decembris', 'janvāris', 'februāris')
korteža = ('decembris', 'februāris', 'janvāris')
```

2. Pārvērst korteža datus: simboliskajos, saraksta, vārdnīcas tipā:

```
main.py x Console Shell
1 #Tuple to Str
2 romuTuple = ('I', ' ', 'II', ' ', 'III', ' ', 'IV')
3 print("Tuple= ",romuTuple)
4 romuStr = ''.join(romuTuple)
5 print("Str= ", romuStr)
6 #Tuple to List
7 romuList = list(romuTuple)
8 print("List=",romuList)
9 #Tuple to Dic
10 romu = (('I',1),('II',2))
11 romuDict = dict((x, y) for x, y in romu)
12 print("Dict= ",romuDict)
```

```
Tuple= ('I', ' ', 'II', ' ', 'III', ' ', 'IV')
Str= I II III IV
List= ['I', ' ', 'II', ' ', 'III', ' ', 'IV']
Dict= {'I': 1, 'II': 2}
```

3. Izveidot funkciju, kura atgriež divas vērtības:

```
main.py x Console Shell
1 def drukaa(arguments):
2     return None, f"abi {arguments} strādā!"
3
4 print("funkcijaa tips= ",type(drukaa('stradaa')))
5 print("funkcijaa = ",drukaa('druka'))
6
7 kluda, zina = drukaa('stradaa,')
8 print("kluda= ",kluda)
9 print("zina= ",zina)
```

```
funkcijaa tips= <class 'tuple'>
funkcijaa = (None, 'abi druka strādā!')
kluda= None
zina= abi stradaa, strādā!
```

Uzdevums

1. Izveidot kortežu mēneši un atrodot savu dzimšanas mēnesi tajā izdrukāt to un mēneša kārtas numuru, izmantojot metodi index()

Kopa {Set}:

{ unikālu, nesakārtotu elementu kolekcija, ielikta figūriekavās }

Par kopas elementu var būt jebkurš nemainīgs datu tips (skaitļi, virknes, korteži). Ar kopām var meklēt apvienojumus, šķēlumus,

Vingrinājumi

1. Pārbaudīt elementārās darbības ar kopām

```
main.py x Console Shell
1 k={} #izveido vārdu
2 print(type(k))
3 k=set() #izveido kopu
4 print(type(k))
5 v=[1,23,'jā','nē',1,2,3,'jā','nē']
6 k={1,23,'jā','nē',1,2,3,'jā','nē'}
7 print("v= ", v,"nk= ", k)
8 v= list(set(k)) # saraksts => kopā => saraksts
9 print("\nsarakstā nav dublikātu = ", v)
10 k.add((3,5)) ; print ("pielikt (4,5) = ",k)
11 k.add((3,5)) ; print ("pielikt (4,5) = ",k) # neduble
12 k.update('9','8','7') # skaitliskos nevar
13 print ("pielikt 7,8,9 = ",k)
14 k.discard(5) #drikt dzest neesoshu
15 print ("dzests 5 = ",k)
16 k.pop() #izmet pirm
17 print ("izmet = ",k)
18 print ("3 not in k = ", 3 in k)
```

```
<class 'dict'>
<class 'set'>
v= [1, 23, 'jā', 'nē', 1, 2, 3, 'jā', 'nē']
k= {1, 'nē', 'jā', 2, 3, 23}

sarakstā nav dublikātu = [1, 'nē', 'jā', 2, 3, 23]
pielikt (4,5) = {1, 'nē', 'jā', 2, 3, 23, (3, 5)}
pielikt (4,5) = {1, 'nē', 'jā', 2, 3, 23, (3, 5)}
pielikt 7,8,9 = {1, 'nē', 'jā', 2, 3, '7', '9', '8', 23, (3, 5)}
dzests 5 = {1, 'nē', 'jā', 2, 3, '7', '9', '8', 23, (3, 5)}
izmet = {'nē', 'jā', 2, 3, '7', '9', '8', 23, (3, 5)}
3 not in k = True
>
```

2. Pārbaudīt loģiskās operācijas

```
main.py x Console Shell
1 kopaA = {1,2,3,4,5,6,7}
2 kopaB = {2,4,6,8}
3 print("kopaA =",kopaA, "\nkopaB =",kopaB)
4 #skelums kopigie punktie
5 print("\šķēlums = ", kopaA & kopaB)
6 kopaC = {1,3,5,7}
7 print("\Tukšs šķēlums = ", kopaC & kopaB)
8
9 kopaRez = kopaA.intersection(kopaB)
10 print("\nšķēlumā=",kopaRez)
11 #Apvietoti
12 kopaRez = kopaA | kopaB
13 print("\nApvienoti=",kopaRez)
14 #Atņemsana
15 kopaRez = kopaA - kopaB
16 print("\nkopaA - kopaB =",kopaRez)
17 print("\nkopaB - kopaA =",kopaB - kopaA)
18 #Simetriskie dati
19 print("\nkopaA ^ kopaB =",kopaA ^ kopaB)
20 #Salīdzina vai vienaadi
21 print("\nkopaA == kopaB =",kopaA == kopaB)
22 #Salīdzina vai nevienaadi
23 print("\nkopaA != kopaB =",kopaA != kopaB)
```

```
kopaA = {1, 2, 3, 4, 5, 6, 7}
kopaB = {2, 4, 6, 8}
šķēlums = {2, 4, 6}
Tukšs šķēlums = set()

šķēlumā= {2, 4, 6}

Apvienoti= {1, 2, 3, 4, 5, 6, 7, 8}

kopaA - kopaB = {1, 3, 5, 7}

kopaB - kopaA = {8}

kopaA ^ kopaB = {1, 3, 5, 7, 8}

kopaA == kopaB = False

kopaA != kopaB = True
>
```

Uzdevums

1. Komentēt 2.vingrinājuma kopu apvienošanas īpatnības, salīdzinot ar izņemtajām datu struktūrām.

Vārdnīca {Dictionary}: -

{unikālu, nesakārtotu elementu kopa, kurā piekļuve vērtībām caur atslēgu}

Vingrinājumi

Izpētīt darbu ar vārdnīcu

```
main.py x Console Shell
1 vardnica = {}
2 vardnica = dict.fromkeys([1,2,3])
3 print(vardnica)
4 vardnica = dict.fromkeys([1,2,3],"x")
5 print("\nvienas vērtības",vardnica)
6 tulko = {
7     'list': 'saraksts',
8     'tuple': 'kortežs',
9     'set': 'kopa',
10    'dict': 'vārdnīca'
11 }
12 print("\nvisa vārdnīca= ",tulko)
13 print("\nizdrukā vienu vērtību = ",tulko['set'])
14 tulko['stack'] = 'steks' #pieliek vardnīcai
15
16 print("\nvisa vārdnīca ar for: ")
17 for x in tulko:
18     print(x, " = ",tulko[x])
19 print("\nvārdnīcā ",len(tulko), " ieraksti")
20 print("\nvai ir sarakstā = ", 'list' in tulko)
```

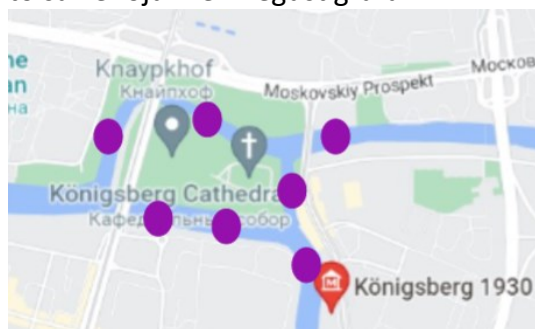
```
{1: None, 2: None, 3: None}
vienas vērtības {1: 'x', 2: 'x', 3: 'x'}
visa vārdnīca= {'list': 'saraksts', 'tuple': 'kortežs', 'set': 'kopa', 'dict': 'vārdnīca'}
izdrukā vienu vērtību = kopa
visa vārdnīca ar for:
list = saraksts
tuple = kortežs
set = kopa
dict = vārdnīca
stack = steks
vārdnīcā 5 ieraksti
vai ir sarakstā = True
>
```

Uzdevums:

Izveidot vārdnīcu: lai uzrakstot atzīmi iegūtu skaidrojumu: 10-izcili, 9-teicami, 8 – ļoti labi, 7 – labi, 6 – gandrīz labi, 5 – viduvēji, 4- gandrīz viduvēji,...

Grafi

Par grafu sauc nelineāru datu struktūru, kura sastāv no objektiem un to savienojumiem. Ļoti daudzas reālās sistēmas matemātiski abstrahējot uz objektiem un to savienojumiem iegūst grafu.

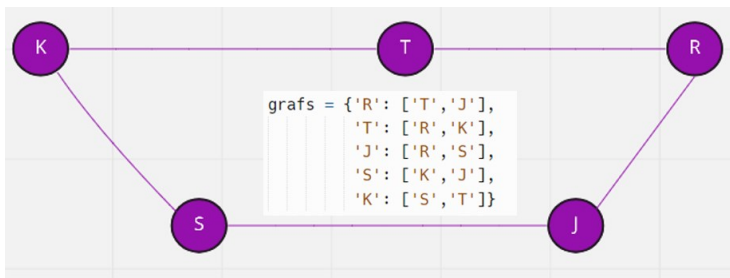
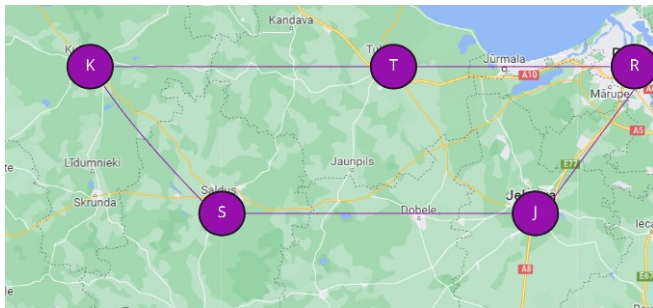
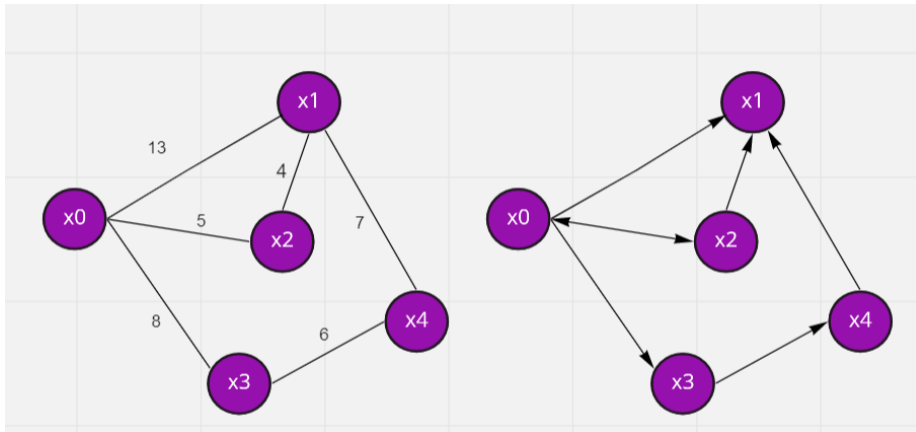


1736. gadā L. Eilers uzsāka grafu teoriju - pierādot, ka pa septiņiem Kēnigsbergas pilsētas tiltiem nevar no viena punkta izejot atgriezties tajā, ja pa katru tiltu iet tikai vienu reizi.

Grafs kā matemātisks objekts sastāv no virsotnēm (V) un malām/šķautnēm (E) veidojot sakārtotu kopu pāri $G = (V, E)$.

Mezgli ir tikai kāds objekts, un mala ir

savienojums starp tiem. Kreisajā pusē ir neorientēts grafs, tas ir, bez skaidriem virzieniem, un malas šeit ir divvirzienu. Labajā pusē ir orientēts grafs, kuru malām ir noteikts virziens.



Meklēšana plašuma (**Breadth-first search**)

nesvērtam grafam:

- ❖ Pareizi aprakstīt kā virsotnes saistītas.
- ❖ Importēt bibliotēkas sadaļu darbam ar rindām
- ❖ Izveidot vārdnīcu, kur virsotne ir atslēga ar kuru iegūt saistītās virsotnes, t.i. vērtību
- ❖ Funkcijā
 - sākuma virsotni ielikt rindā
 - veidot apmeklēto virsotņu vārdnīcu
 - kamēr rinda nav tukša
 - izņemt pirmo virsotni un iegūt saistītās virsotnes, ja
 - tās nav apmeklēto vārdnīcā, tad ieliek vārdnīcā
- ❖ ierakstīt info no kurienes atnāca uz norādīto virsotni

main.py ×



Console Shell

```
1 from collections import deque
2
3 ▼ grafs = {'R': ['T', 'J'],
4           'T': ['R', 'K'],
5           'J': ['R', 'S'],
6           'S': ['K', 'J'],
7           'K': ['S', 'T']}
8 ▼ def bfs(no, uz, grafs):
9     q = deque([no])
10    Bija = {no : None}
11 ▼ while q:
12     virsotne = q.popleft() #virsotne
13 ▼     if virsotne == uz:
14         break
15     nakamVirsothes = grafs[virsotne]
16 ▼     for v in nakamVirsothes:
17 ▼         if v not in Bija:
18             q.append(v)
19             Bija[v]=virsotne
20     return Bija
21
22 no = 'R'
23 uz = 'K'
24 Bija = bfs(no, uz, grafs)
25
26 print(f'no {no} līdz {uz}: \n{uz} ', end=" ")
27 līdz = uz
28 ▼ while līdz != no:
29     līdz = Bija[līdz]
30     print(f'<--- {līdz}', end='')

```

```
no R līdz K:
K <--- T<--- R
```

Meklēšana dziļumā (Depth-first search, DFS)

Parasti izmanto neizpētītu grafu apstrādei, ar rekursijas palīdzību izpēta visas virsotnes, tad atgriežas sākumpunkta un no citas blakus virsotnes pēta rekursijā:

1. Dodieties uz kādu blakus virsotni, kas iepriekš nav apmeklēta.
2. No šīs virsotnes palaist dziļuma pirmās meklēšanas algoritmu
3. Atgriezties uz sākuma virsotni.
4. Atkārtojiet 1.-3. darbību visām iepriekš neapmeklētajām blakus virsotnēm.

Deikstra algoritms

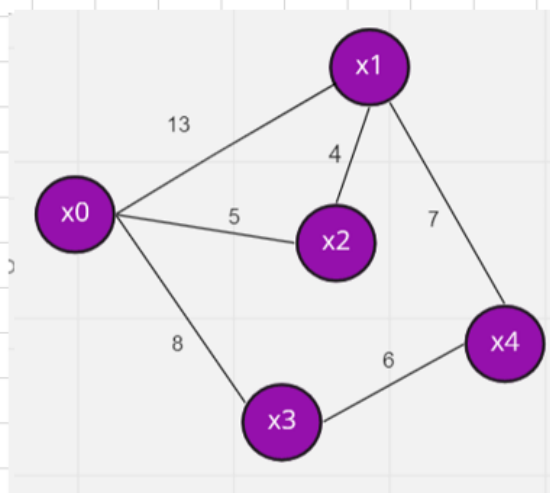
Īsāko ceļu no viena punktu līdz visiem citiem var noteikt ar Deikstra algoritmu, kuru viņš izdomāja 1959.gadā, lai ietaupītu laiku apmeklējot veikalus ar sievu. Ar grafa palīdzību rēķina informācijas paku maršrutizētāja ceļu.

Deikstra algoritms:

1. Izvēlas vienu no sākumu virsotnēm, kurai attālums līdz sevīm 0, pārējiem bezgalība, (šeit 999), doto virsotni pievieno pie izmantotajām virsotnēm
2. Kamēr neizmanto visas virsotnes
 - atrod vismazāko iegūto attālumu, kuru pieskaita visām saistītajām virsotnēm no dotās virsotnes, ja tur lielāks skaitlis, tad to samaina ar iegūto skaitli
 - nesaistītajām virsotnēm pārraksta nemainītu attālumu
 - aplūkoto virsotni pievieno izmantoto virsotņu sarakstam

| | x0 | x1 | x2 | x3 | x4 |
|----|----|----|----|----|----|
| x0 | 0 | 13 | 5 | 8 | 0 |
| x1 | 13 | 0 | 4 | 0 | 7 |
| x2 | 5 | 4 | 0 | 0 | 0 |
| x3 | 8 | 0 | 0 | 0 | 6 |
| x4 | 0 | 7 | 0 | 6 | 0 |

| Nr | x0 | x1 | x2 | x3 | x4 |
|----|----|-----|-----|-----|-----|
| 1 | 0 | 999 | 999 | 999 | 999 |
| 2 | | 13 | 5 | 8 | 999 |
| 3 | | 9 | | 8 | 999 |
| 4 | | | | | 14 |



The screenshot shows a Python IDE with a file named 'main.py'. The code implements a shortest path algorithm (likely Floyd-Warshall) on a graph with 5 nodes. A visualization of the graph is shown as a 5x5 matrix with nodes 1-4 on the left and node 5 on the right. The matrix cells contain edge weights, with some cells highlighted in red or pink. The code includes functions for generating edges, finding the minimum path, and updating the distance matrix.

```

15 inf = 999
16 D = (( 0,13, 5, 8, 0),
17       (13, 0, 4, 0, 7),
18       ( 5, 4, 0, 0, 0),
19       ( 8, 0, 0, 0, 6),
20       ( 0, 7, 0, 6, 0))
21
22 N = len(D) # grafa virsotņu skaits
23 T = [inf]*N # pēdējā tabulas rinda
24 v = 0      # sakuma virsotne = 0
25 Bija = {v} # apskatītas virsotnes
26 T[v] = 0   # sakuma virsotnes svārs = 0
27
28 while v != -1: # kamēr nav visas virsot
29     for j in saites(v,D): #virs saistības
30         if j not in Bija: # ja virsotne ne
31             svārs = T[v] + D[v][j] #pieskaita...
32             saistītajam
33             if svārs < T[j]: # ja svārs mazāks
34                 T[j] = svārs # tad saglabā tikai min
35
36     v = minMala(T, Bija) # nakamais mezglis ar
37     mazāko svāru
38     if v >= 0:          # nakama virsotne
39         Bija.add(v)      # pieliek nakamo
40         virsotni apskatām
41
42 print(T)

```

Visualization of the graph (Nodes 1-4 on the left, Node 5 on the right):

| | | | | | |
|---|---|-----|-----|-----|-----|
| 1 | 0 | 999 | 999 | 999 | 999 |
| 2 | | 13 | 5 | 8 | 999 |
| 3 | | 9 | | 8 | 999 |
| 4 | | | | | 14 |

8, 14]

```

1 def saites(v,D):
2     for i, svārs in enumerate(D[v]):
3         if svārs > 0:
4             yield i
5
6 def minMala(T, Bija):
7     maz = -1 # ja -1, visas izmantojam
8     m = 999 # lielākā vērtība
9     # i - virsotnesNr no 0
10    for i, t in enumerate(T):
11        if t < m and i not in Bija:
12            m = t
13            maz = i
14    return maz

```

Uzdevums:

Ir pieci biroji piecās dažādās pilsētās. Ceļojuma izmaksas starp katru pilsētu pāri ir zināmas. Noteikt lētākais veids, kā sasniegt katru biroju no jebkura izvēlēta biroja.

Koks ir grafa zars.

Katram kokam ir saknes mezglis, no kura atzarojas visi pārējie mezgli. Saknē ir norādes uz visiem tieši zem tās esošajiem elementiem, kas ir zināmi kā tās pakārtotie mezgli. Mezglus bez pakārtotiem mezgliem sauc par lapu mezgliem.

Gan mezgli, gan malas var būt informācijas nesēji. Piemēram, ēkas ar ielām, kuras tās savieno.

Floida Voršala algoritms, lai atrisinātu visu pāru īsākā ceļa problēmu

Floida-Varšala algoritms ir algoritms īsākā ceļa atrašanai starp visiem virsotņu pāriem svērtā grafā. Šo algoritmu var izmantot gan virzītiem, gan nevīrītiem svērtiem grafikiem, bet ne ar negatīviem.