NAPOJ

UČNA PRIPRAVA

Množica

DATUM NAPOJ

OSNOVNI PODATKI

Šola:

Letnik: 1. Letnik

Datum:

Predmet: Informatika

Učna tema: množica

Učna enota: množica, Python set, implementacija množice, dvojiško drevo, dvojiško iskalno drevo

Učne oblike:

- frontalno, individualno, skupinsko

Učne metode:

- razlaga, pogovor, demonstracija, reševanje problemov

Predznanje:

- poznajo pojme algoritem, funkcija, tabela

Operativni učni cilji

Ob koncu učne ure učenec:

- Razume pojem množice.
- Zna implementirati množico.
- Razume dvojiško drevo
- Razume dvojiško iskalno drevo.
- Pozna osnove balansiranja dvojiškega drevesa.

Učna sredstva:

- Učila: list, računalnik
- Učni pripomočki: računalnik, Python, zvezek, tabla

Didaktične etape učnega procesa:

1. Pripravljanje ali uvajanje
2. Obravnava nove učne snovi ali usvajanje
3. Urjenje
4. Ponavljanje
5. Preverjanje

Medpredmetne povezave: slovenščina, angleščina

Literatura:

- C.-C. Li, An immediate approach to balancing nodes in binary search trees, Journal of Computing Sciences in Colleges 21 (4) (2006) 238–245.
- P. E. Smith, J. H. Graham, A simple balanced search tree, in: Proceedings of the 1993 ACM conference on Computer science, ACM, 1993, pp. 461–465.
- S. Sen, R. E. Tarjan, Deletion without rebalancing in balanced binary trees, in: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2010, pp. 1490–1499.
- M. T. Goodrich, R. Tamassia, Data structures and algorithms in Java, 2001.
- M. A. Weiss, Data Structures and Algorithm Analysis (2nd Edition), Addison Wesley, 1994.
- R. Sedgewick, K. Wayne, Algorithms, Addison-Wesley Professional, 2011.

- D. D. Sleator, R. E. Tarjan, W. P. Thurston, Rotation distance, triangulations, and hyperbolic geometry, Journal of the American Mathematical Society 1 (3) (1988) 647–681. - G. Anželj, J. Brank, A. Brodnik, P. Bulić, M. Ciglarič, M. Đukić, L. Furst, M. Kikelj, A. Krapež, H. Medvešek, N. Mori, M. Pančur, P. Sterle, Računalništvo in informatika 1, Založba Univerze na Primorskem and Založba Fakultete za računalništvo in informatiko and Založba Fakultete za elektrotehniko and računalništvo in informatiko, 2015. URL https://lusy.fri.uni-lj.si/ucbenik/book/index.html.
Novi pojmi: - množica, Python set, dvojiško drevo, dvojiško iskalno drevo, balansiranje dvojiškega drevesa.
Priloga: - rešitev naloge Prosti časi - rešitev naloge Presek - Delovni list - Delovni list - rešitve

POTEK UČNE URE

UVODNI DEL: UVAJANJE

ČAS	UČITELJ	UČENEC	UČNE OBLIKE, METODE, TEHNIKE, UČNI PRIPOMOČKI
Uvod 5 min	*Pozdravi učence, zapiše manjkajoče učence in po potrebi prosi dežurnega učenca, da pobriše tablo.* Danes se bomo spoznali s pojmi množica, implementacija množice s tabelo, dvojiško drevo ter iskalno dvojiško drevo. Množica je odličen pripomoček, ki nam lahko zelo olajša delo pri reševanju problemov. Poglejmo si kaj množica sploh je ter kje v svetu se jo uporablja.	Odzdravijo, naštejejo manjkajoče učence, dežurni učenec pobriše tablo.	Frontalno, razlaga, pogovor.
Motivacija 5 min	Pred dvema tednoma smo spoznali zaporedje. Rekli smo, da je to abstraktna podatkovna struktura, kjer je vrstni red elementov pomemben, elementi pa se lahko poljubno ponavljajo. V svetu pa imamo tudi primere, ko nam vrstni red ni pomemben, nam je pa dejstvo, da vsak element nastopa samo enkrat. Bi jih lahko nekaj našteli? Nekaj primerov: • Učenci na zaključnem izletu tvorijo množico, saj v njej nimamo podvajanj, poleg tega pa nas za vsakega dijaka zanima zgolj to, ali se je izleta udeležil, nima pa smisla govoriti o vrstnem redu udeležencev izleta. • Moštvo/ekipa predstavlja množico, pri	Poslušajo. Odgovorijo, naštejejo primere.	Frontalno, razlaga, pogovor.

	čemer so tekmovalci elementi, vrstni red pa ni pomemben. • Koliko pevcev je v pevskem zboru. V vseh teh primerih uporabimo abstraktno podatkovno strukturo množica.		

GLAVNI DEL: OBRAVNAVANJE UČNE SNOVI

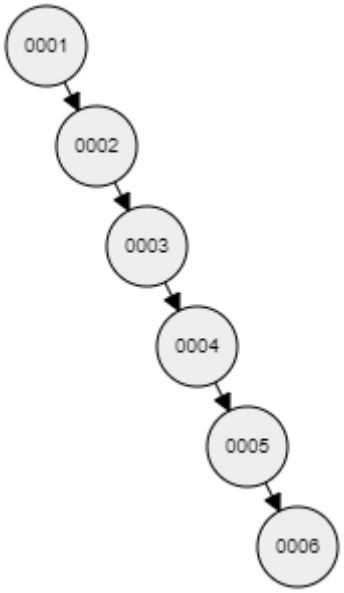
VSEBINSKI POUDARKI	UČITELJ	UČENEC	UČNE OBLIKE, METODE, TEHNIKE, UČNI PRIPOMOČKI
Python set 15 min	Python že vsebuje podatkovni tip set, ustvarimo pa ga z mnozica=set(), kjer kot argument podamo elemente, recimo števila: `mnozica = set([1,3,5,2,4])` Če poskusimo ustvariti množico, kjer se elementi ponavljajo, recimo set([1,1,2,2,3,3,4]), se v njo vsak element pojavi le enkrat. Iz matematike pa vemo, da nad množicami lahko izvajamo operacije unija, presek in razlika. Poglejmo si, kako to naredimo v Pythonu. `m1 = set([1,3,5,7,9])` `m2 = set([1,4,7,10])` unija: `print(m1.union(m2))` presek: `print(m1.intersection(m2))` shranitev rezultata: `m1= m1.intersect(m2)` Sedaj imam za vas nalogo. Rešite jo lahko samostojno ali v skupini. Imamo 4 prijatelje, ki bi se radi dobili na pijači. Ker vsi že vedo, katere dneve so prosti, se morajo le še zmeniti, kateri dan jim skupaj najbolj ustreza. Ana ima čas v ponedeljek, torek in četrtek. Borut ima čas v torek, sredo in četrtek. Cilka ima čas od četrtka do vključno nedelje, Dejan pa od ponedeljka do četrtka. Rešite nalogo tako, da boste uporabili množico za vsakega od prijateljev ter izpišite, na kateri dan so vsi prijatelji prosti. *Učitelj skupaj z učenci preveri rešitev naloge. Rešitev je v prilogi.*	Spremljajo razlago in si zapišejo v zvezek. Rešijo nalogo, preverijo rešitve.	Frontalno, razlaga, individualno, razlaga, reševanje problemov.
Implementacija množice	Poskusimo implementirati množico s tabelo. Kot smo omenili, sta lastnosti množice 2: vrstni red		

s tabelo 10 minut	elementov ni pomemben, ter, da se elementi ne smejo podvajati. Za to bomo morali napisati funkcijo, ki ob vsakem dodajanju novega elementa preveri, ali tak element že ne obstaja v množici oziroma tabeli. Napisali bomo funkcije za kreiranje nove množice, dodajanje elementa, brisanje elementa in izpis elementov. Kreiranje nove množice je enostavno – zgolj kreiramo novo tabelo: `def ustvariMnozico():` `  return []` Pri dodajanju elementa moramo, kot smo omenili, preveriti, ali element še ne obstaja v množici. `def dodaj(mnozica, element):` `  if not jePrisoten(mnozica, element):` `  mnozica.append(element)` Seveda moramo sedaj še napisati funkcijo jePrisoten(), ki vrne True, če je element že v množici. `def jePrisoten(mnozica, element):` `  for el in mnozica:` `  if el == element:` `  return True` `  return False` Manjka nam le še brisanje elementa. `def odstrani(mnozica, element):` `  mnozica.remove(element)` To je to. Program lahko še testiramo, da preverimo, ali dobimo pravilen izpis. `mnozica = ustvariMnozico()` `dodaj(mnozica, 1)` `dodaj(mnozica,2)` `dodaj(mnozica,3)` `dodaj(mnozica,3)` `odstrani(mnozica, 1)` `izpisiMnozico(mnozica) # izpise {2, 3}` Sedaj pa vam predajam naslednjo nalogo. Napišite funkcijo presek(mnozica1, mnozica2), ki izpiše elemente, ki se pojavijo v obeh množicah. Za pomoč: za vsak element iz prve množice	Spremljajo razlago in si zapišejo v zvezek. Pomagajo učitelju pri pisanju funkcij. Rešijo nalogo,	Frontalno, razlaga, individualno, razlaga, reševanje problemov.

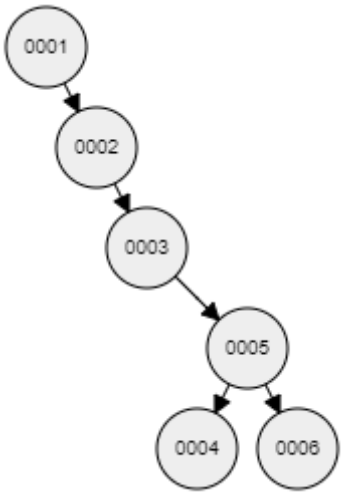
	preverimo, ali obstaja tudi v drugi množici. Če ja, ga dodamo v niz izpis. Spet lahko rešujete ali samostojno ali v skupini, če potrebujete pomoč pa me vprašajte. *Učitelj po potrebi pomaga učencem pri nalogi. Na koncu preverijo rešitev. Rešitev je v prilogi.* Poglejmo si časovne zahtevnosti funkcij jePrisoten(), dodaj() in odstrani(). Mi lahko vi odgovorite? Ker O(n) čas ni ravno super za takšno enostavno podatkovno strukturo, jo bomo implementirali z tako imenovanim dvojiškim iskalnim drevesom.	preverijo rešitve. Odgovorijo: jePrisoten() porabi O(n) časa, saj se mora sprehoditi čez vse elemente tabele. dodaj() prav tako potrebuje O(n), saj kliče funkcijo jePrisoten(), isto pa je tudi za odstrani().	
Dvojiško iskalno drevo 27 min	V povezanem seznamu vsak element kaže na enega naslednjega elementa. Spoznajmo podatkovno strukturo dvojiško drevo, kjer vsak element kaže na 2 naslednja elementa. Zakaj, bomo videli v nadaljevanju. *Učitelj na tablo nariše primer dvojiškega drevesa.* Drevo rišemo od zgoraj dol, pri čemer prvi element imenujemo koren, vozlišči na kateri kaže, pa imenujemo otroka. Vsak od teh otrok ima lahko tudi svoje otroke. Vozlišče brez otrok se imenuje list. V tem drevesu imajo vozlišča poljubne vrednosti. Če pa dodamo pravilo, da mora biti levi otrok po vrednosti manjši od starša, desni pa večji, dobimo dvojiško iskalno drevo. Za vajo mi povejte, ali je katero od naslednjih dvojiških dreves tudi iskalno. *Učitelj na tablo nariše 3 ali 4 dvojiška drevesa, nekatera iskalna, nekatera ne.* Iskalno se imenuje zato, ker omogoča učinkovito iskanje elementov. Iskanje poteka na zelo intuitiven način. Začnemo v korenu; če je iskan element manjši od trenutnega vozlišča. Se pomaknemo v levega otroka, če je večji, pa v desnega. To ponavljamo dokler ne pridem do iskanega elementa ali do lista, ki nima več otrok in	Sledijo razlagi in si zapisujejo. Določijo iskalna dvojiška drevesa na tabli.	Frontalno, razlaga, pogovor, demonstracija.

	tako vemo, da iskanega elementa ni v drevesu. *Učitelj na enem od dvojiških iskalnih dreves na tabli prikaže dodajanje novega elementa.* Sedaj dodajmo še en element, pri čemer mi boste vi povedali, kam ga moram dodati. Dvojiško iskalno drevo lahko implementiramo kar s tabelo, pri čemer je koren drevesa (tj. prvo vozlišče) na indeksu 0, otroci vozlišč pa na indeksih 2*indeks_starša+1 ter 2*indeks_starša+2. Najprej napišimo psevdokodo, nato pa sprogramirajmo dvojiško drevo v Pythonu. Napisali bomo metode ustvari(), dodaj() in jePrisoten(). Z odstranjevanjem elementov se ne bomo ukvarjali, saj bi se s kompleksnostjo logike samo zmedli. V prvi funkciji moramo le vrniti novo tabelo: def ustvari(): vrni prazno tabelo Ker želimo imeti možnost v drevo po potrebi dodati tudi število nič, v funkciji ne moremo napisati le return [], ampak: `def ustvari():` `  return [None]*100` ki vrne tabelo velikosti 100, ki vsebuje prazne element (drugače bi imeli tabelo ničel). V funkciji dodaj() se moramo ustrezno sprehoditi po drevesu do listov, ter dodati element. def dodaj(drevo, element): indeks = 0 while: če je drevo[indeks] prazno: dodaj element na ta indeks če je element v drevo[indeks]: vrni True če je element manjši od drevo[indeks]: indeks = 2*indeks+1 drugače: indeks = 2*indeks+2 Mi lahko pomagate napisati funkcijo v Pythonu? `def dodaj(drevo, element):` `  indeks = 0` `  while True:` `  if drevo[indeks] is None:` `  drevo[indeks] = element` `  return True` `  if element == drevo[indeks]:` `  return True`	Dodajo element v dvojiško drevo. Napišejo funkcijo dodaj() v Pythonu.	

	`` if element < drevo[indeks]: indeks = 2*indeks + 1 else: indeks = 2*indeks + 2 ``` Za funkcijo jePrisoten() je psevdokoda zelo podobna: def jePrisoten(drevo, element): indeks = 0 while True: če je drevo[indeks] prazno: vrni False če je element v drevo[indeks]: vrni True če je element manjši od drevo[indeks]: indeks = 2*indeks+1 če je element večji od drevo[indeks]: indeks = 2*indeks+2 Mi lahko pomagate napisati funkcijo v Pythonu? ``` def jePrisoten(drevo, element): indeks = 0 while True: if drevo[indeks] is None: return False if element == drevo[indeks]: return True elif element < drevo[indeks]: indeks = 2*indeks + 1 else: indeks = 2*indeks + 2 ``` Našo program lahko seveda testiramo: ``` drevo = ustvari() print(drevo) print("Je prisoten 10?", jePrisoten(drevo, 11)) dodaj(drevo, 5) dodaj(drevo, 3) dodaj(drevo, 7) dodaj(drevo, 1) dodaj(drevo, 4) dodaj(drevo, 6) dodaj(drevo, 10) print(drevo) print("Je prisoten 10?", jePrisoten(drevo, 11)) ```	Napišejo funkcijo jePrisoten() v Pythonu.	
Časovna zahtevnost dvojiškega iskalnega drevesa 5 minut	Sedaj si poglejmo časovno zahtevnost dodajanja in iskanja elementa v drevesu. Recimo, da imamo drevo. Pri dodajanju elementa se moramo sprehoditi do listov, pri čemer se ob vsakem premiku v novo vozlišče znebimo pol vozlišč, saj drugi otrok vsebuje približno enako veliko poddrevo. Pri vsakem premiku se zgodi isto – znebimo se bodisi levega, bodisi desnega poddrevesa. Temu, ko se ob	Sledijo razlagi in si zapisujejo.	Frontalno, razlaga, individualno, razlaga, reševanje problemov.

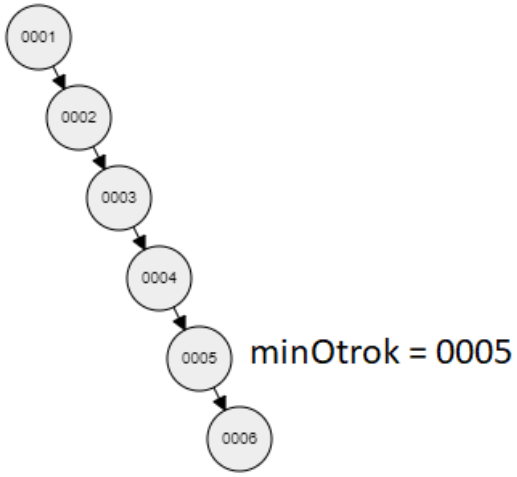
	vsaki iteraciji znebimo pol elementov, imenujemo dvojiški logaritem. Torej imamo O(log(n)) za iskanje pravega mesta, kamor bomo dodali element, za kreiranje tega mesta pa potrebujemo O(1) časa, saj zgolj vpišemo element na mesto v tabeli. Tako za dodajanje elementa v dvojiško iskalno drevo potrebujemo O(log(n)) časa. Enako pa pravzaprav velja tudi za iskanje elementa, saj se tudi pri tem sprehodimo do lista, kar je O(log(n)), ter preverimo, ali je v listu iskani element, za kar porabimo O(1) časa. Tako je tudi za iskanje elementa časovna zahtevnost O(log(n)).		
Uravnotežen je dvojiškega drevesa 10 minut	Obstaja primer drevesa, ki deluje kot navadna tabela. Poglejmo, kakšno drevo nastane, če ustvarimo novo drevo in dodamo elemente 1, 2, 3, 4, 5 in 6.  Katera podatkovna struktura je to? Za njega pa vemo iz poglavja Zaporedje, da ima časovno zahtevnost iskanja O(n). Za tako drevo pravimo, da ni uravnoteženo. To pomeni, da za vsaj eno vozlišče velja, da je razlika globin njegovega levega in desnega poddrevesa večja od 1. Primer: na zgornjem primeru za vozlišče 0004 velja, da je globina levega poddrevesa 0, desnega pa 2. Razlika je 2-0 = 2, kar je večje od 1. Sedaj odprimo debato: kaj bi vi naredili, da bi uravnotežili to drevo? No, v programerskem svetu moramo tako drevo uravnotežiti. To pomeni, da ga zapišemo na rahlo drugačen način, da dobimo časovno zahtevnost iskanja spet O(log(n)), a ohranimo pravili, da vsako vozlišče ima 2 otroka, ter da je levi otrok manjši od desnega.	Sledijo razlagi in si zapisujejo. Odgovorijo: povezan seznam. Učenci podajajo ideje za uravnoteženje drevesa.	

Recimo, vozlišče z vrednostjo 0004 uravnotežimo na naslednji način:

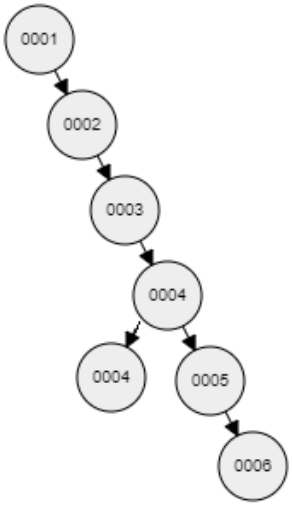


Ker je prej vozlišče 0004 imelo preveč **desnih** otrok, smo uporabili postopek, ki mu pravimo **levo** vrtenje na vozlišču 4. Postopek je sledeč:

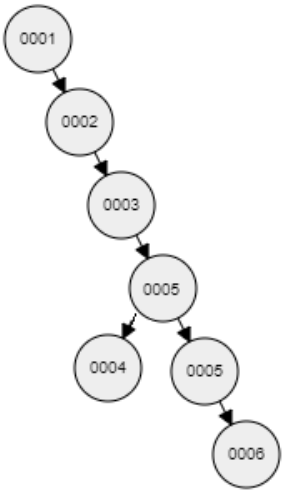
1. Identificiramo najmanjšega otroka v desnem poddrevesu.



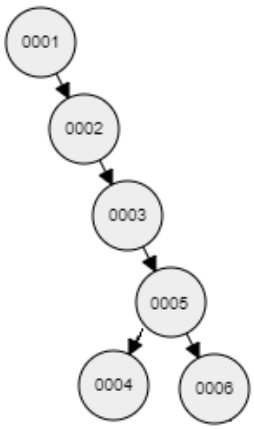
2. Kopiramo vozlišče starš in ga vstavimo v njegovo levo poddrevo. Začasno imamo 2 vozlišči z isto vrednostjo.



3. Vrednost vozlišča minOtrok zapišemo v vozlišče starš.



4. Izbrišemo vozlišče minOtrok iz desnega poddrevesa in po potrebi prevežemo otroke vozlišča minOtrok.



Levo vrtenje uporabimo, ko za neko vozlišče velja, da ima preveč desnih otrok. Sklepamo lahko, da če ima neko vozlišče preveč levih otrok, uporabimo

	desno vrtenje. Na hitro zapišimo postopek tega vrtenja: 1. minOtrok = najmanjši element v levem poddrevesu vozlišča starš. 2. Kopiraj vozlišče starš in ga vstavi v njegovo desno poddrevo. Začasno imamo 2 vozlišči z isto vrednostjo. 3. Vrednost vozlišča minOtrok zapiši v vozlišče starš. 4. Izbriši vozlišče minOtrok iz levega poddrevesa in po potrebi preveži otroke vozlišča minOtrok. To lahko naredimo za vse elemente, ki so neuravnoteženi. Na koncu dobimo uravnoteženo drevo, tj., da je časovna zahtevnost iskanja, dodajanja in brisanja elementov O(log(n)).		

ZAKLJUČNI DEL: ZAKLJUČNO PONAVLJANJE / PREVERJANJE

ČAS	UČITELJ	UČENEC	UČNE OBLIKE, METODE, TEHNIKE, UČNI PRIPOMOČKI
10 min	Za konec vas prosim, da rešite delovni list (priloga 3), na koncu pa bomo pregledali vaše odgovore.	Poslušajo, rešijo delovni list, pregledajo rešitve.	Frontalno, individualno, razlaga, reševanje problemov, pogovor.
2 min	*Se zahvali učencem za sodelovanje, jim da napotke za domačo nalogo. Dežurni učenec naj po potrebi pobriše tablo. Učenci naj ugasnejo računalnike.*	Poslušajo, ugasnejo računalnike, dežurni učenec pobriše tablo.	Frontalno.

Priloge

Priloga 1: Rešitev naloge Prosti časi

```
ana = set(["ponedeljek", "torek", "cetrtek"])
bojan = set(["torek", "sreda", "cetrtek"])
cilka = set(["cetrtek", "petek", "sobota", "nedelja"])
dejan = set(["ponedeljek", "torek", "sreda", "cetrtek"])
prosto = ana.intersection(bojan)
prosto = prosto.intersection(cilka)
prosto = prosto.intersection(dejan)
print(prosto)
```

Priloga 2: rešitev naloge Presek

```
def presek(mnozica1, mnozica2):
    pr = ustvariMnozico()
    for el in mnozica1:
        if jePrisoten(mnozica2, el):
            dodaj(pr, el)
    izpisiMnozico(pr)
```

Priloga 3: delovni list

Delovni list – Množica

1. Zapišite kaj je množica in kaj velja za elemente v njej. Dodajte še primer množice iz realnega sveta.

2. Spoznali smo Pythonov podatkovni tip set, nad katerimi lahko izvajamo operacije union, intersection in difference. Na kratko napišite kaj naredi vsaka funkcija in kako jo uporabimo.

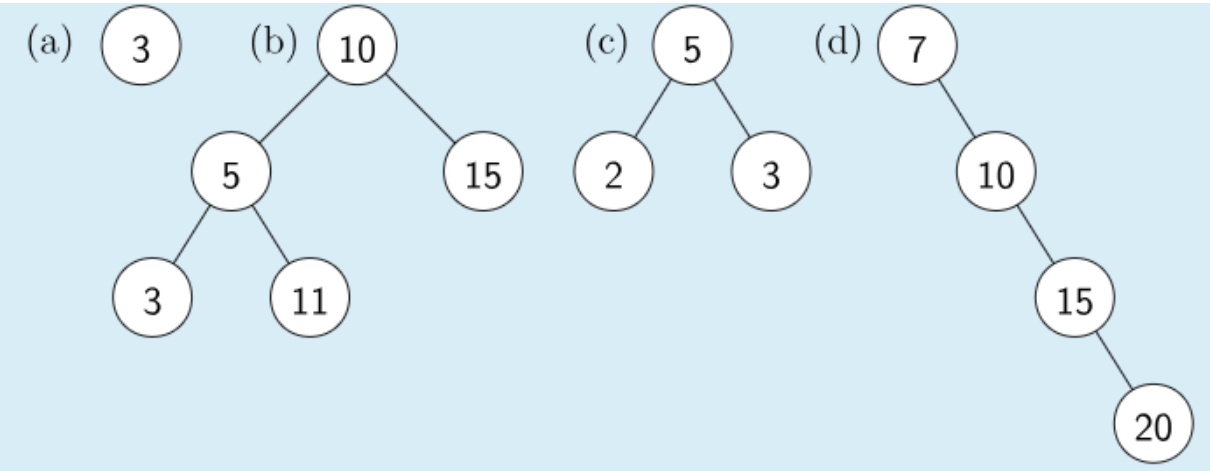
3. Napišite funkcijo steviloRazličnih(tabela), ki izpiše število različnih elementov v prejeti tabeli. V pomoč: tipu set lahko kot parameter podamo kar tabelo, pri čemer bo iz nje naredil množico.

4. Napišite funkcijo skupneCrke(niz1, niz2), ki izpiše črke, ki se ponovijo v obeh prejetih nizih. V pomoč: tipu set lahko kot parameter podamo kar niz, pri čemer bo iz njega naredil množico črk.

5. Spoznali smo, da lahko množico implementiramo s tabelo. Zakaj je to slaba izbira? Kakšne so časovne zahtevnosti osnovnih operacij?

6. Nato smo spoznali dvojiško iskalno drevo. Zapišite, kaj mora veljati za otroke nekega vozlišča..

7. Katera od naslednji dvojiških dreves so iskalna?



8. Vzemite zadnjo drevo iz prejšnje naloge ter ga uravnotežite (opravite levo vrtenje na vozlišču z vrednostjo 10). Za vsak del postopka narišite drevo.

Delovni list – Množica

1. Zapišite kaj je množica in kaj velja za elemente v njej. Dodajte še primer množice iz realnega sveta.

Množica je abstraktna podatkovna struktura za shrambo elementov. Elementi se v njej ne podvajajo. Primer: nogometna ekipa.

2. Spoznali smo Pythonov podatkovni tip set, nad katerimi lahko izvajamo operacije union, intersection in difference. Na kratko napišite kaj naredi vsaka funkcija in kako jo uporabimo.

Union naredi unijo, intersection naredi presek, difference pa razliko 2 množic.

```
m1.union(m2); m1.intersect(m2); m1.differences(m2)
```

3. Napišite funkcijo steviloRazličnih(tabela), ki izpiše število različnih elementov v prejeti tabeli. V pomoč: tipu set lahko kot parameter podamo kar tabelo, pri čemer bo iz nje naredil množico.

```
def steviloRazlicnih(tabela):
    return len(set(tabela))
```

4. Napišite funkcijo skupneCrke(niz1, niz2), ki izpiše črke, ki se ponovijo v obeh prejetih nizih. V pomoč: tipu set lahko kot parameter podamo kar niz, pri čemer bo iz njega naredil množico črk.

```
def skupneCrke(niz1, niz2):
    return set(niz1).intersection(set(niz2))
```

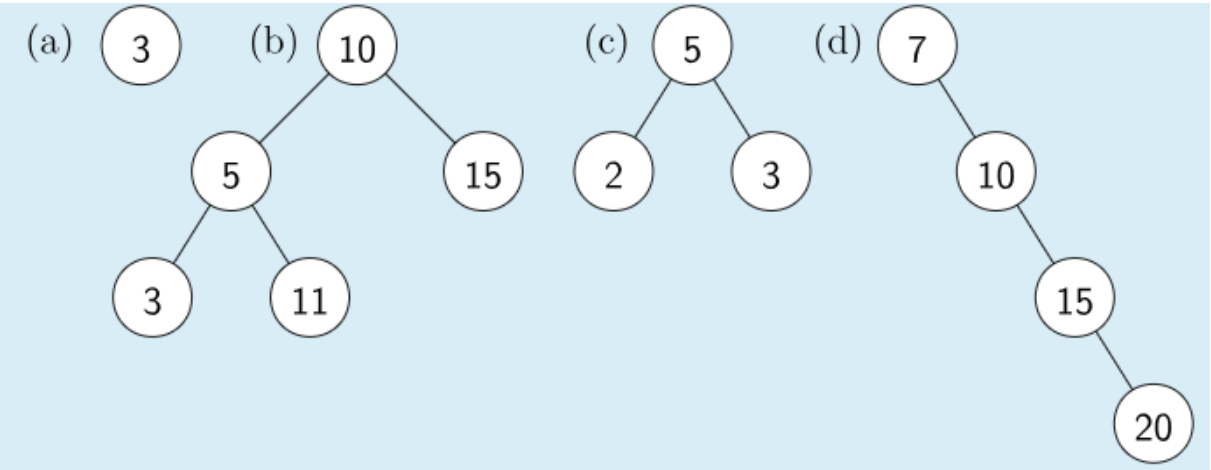
5. Spoznali smo, da lahko množico implementiramo s tabelo. Zakaj je to slaba izbira? Kakšne so časovne zahtevnosti osnovnih operacij?

Ker moramo pri vsakem dodajanju preveriti, ali element že ne obstaja. Dodajanje in brisanje je O(n).

6. Nato smo spoznali dvojiško iskalno drevo. Zapišite, kaj mora veljati za otroke nekega vozlišča.

Levi otrok mora biti manjši, desni pa večji od starša.

7. Katera od naslednji dvojiških dreves so iskalna?



A in D.

8. Vzemite zadnjo drevo iz prejšnje naloge ter ga uravnotežite (opravite levo vrtenje na vozlišču z vrednostjo 10). Za vsak del postopka narišite drevo.